Effective Post-Silicon Validation of System-on-Chips Using Quick Error Detection

David Lin, Ted Hong, Yanjing Li, Eswaran S, Sharad Kumar, Farzan Fallah, Nagib Hakim,
Donald S. Gardner, *Fellow, IEEE*, and Subhasish Mitra, *Fellow, IEEE*

***Abstract*—This paper presents the Quick Error Detection (QED) technique for systematically creating families of post-silicon validation tests that quickly detect bugs inside processor cores and uncore components (cache controllers, memory controllers, and on-chip interconnection networks) of multicore 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 traditional post-silicon validation approaches. QED can be implemented completely in software, without any hardware modification. Hence, it is readily applicable to existing designs. Results using multiple hardware platforms, including the Intel® Core$^{\text{TM}}$ i7 SoC, and a state-of-the-art commercial multicore SoC, along with simulation results using an OpenSPARC T2-like multicore SoC with bug scenarios from commercial multicore SoCs demonstrate: 1) error detection latencies of post-silicon validation tests can be very long, up to billions of clock cycles, especially for bugs inside uncore components; 2) QED shortens error detection latencies by up to nine orders of magnitude to only a few hundred cycles for most bug scenarios; and 3) QED enables up to a fourfold increase in bug coverage.**

***Index Terms*—Electrical bug, logic bugs, post-silicon validation, silicon debug, verification.**

I. Introduction

During post-silicon validation, manufactured integrated circuits (ICs) are tested in actual system environments to detect and fix design flaws (bugs). Design bugs can be broadly classified into two categories.

1) *Logic bugs* that are caused by design errors. Logic bugs include incorrect hardware implementations or incorrect interactions between the hardware implementation and the low-level system software (e.g., firmware).
2) *Electrical bugs* that are caused by subtle interactions between a design and its electrical state. Examples include signal integrity (cross-talk and power-supply noise), thermal effects, and process variations. Electrical bugs often manifest themselves under specific operating conditions, e.g., voltage, frequency, and temperature corners [51].

Post-silicon validation is crucial because pre-silicon verification alone is inadequate. Traditional pre-silicon verification is too slow, and is often incapable of detecting electrical bugs that appear only after ICs are manufactured [51]. However, existing post-silicon validation approaches are *ad hoc*, e.g., insertion of various design for debug (DfD) structures [2], [62] based on heuristics [2], [33], [37]. It has been reported that the costs of post-silicon validation are rising [32]. Furthermore, massive integration of a wide variety of components in complex system-on-chips (SoCs) consisting of multiple processor cores, co-processors/accelerators (e.g., for graphics or cryptography), and uncore components (also referred to as nest or northbridge components) significantly exacerbate the post-silicon validation challenges [3], [42], [57]. *Uncore components* are defined as components in an SoC that are neither processor cores nor co-processors. Examples of uncore components include cache controllers, memory controllers, and on-chip interconnection networks.

Post-silicon validation involves three activities.

1) Detect a bug by applying proper stimuli.
2) Localize/root-cause the bug.
3) Fix the bug using software patches, circuit editing, or silicon respin.

The effort to localize/root-cause bugs from observed failures often dominates the costs of post-silicon validation [2], [5], [30]. Post-silicon validation is difficult because most existing techniques for localizing/root-causing bugs rely on: 1) system-level simulation to obtain the expected or golden system response and 2) system-level failure reproduction, which involves returning the system to an error-free state and rerunning the system with the exact input stimuli (e.g., validation test instructions, validation test inputs, operating conditions such as voltage, temperature and frequency, and interrupts) to reproduce the failure. However, simulation is orders of magnitude slower than actual silicon [47], and failure reproduction is difficult due to Heisenbug effects [23], asynchronous I/Os, and multiple-clock domains.

These problems are further exacerbated by long error detection latencies of bugs. *Error detection latency* is defined as the time elapsed between the occurrence of an error due to a bug and its manifestation as an observable failure. Bugs with error detection latencies longer than a few thousand clock

Manuscript received June 25, 2013; revised November 30, 2013; accepted January 8, 2014. Date of current version September 16, 2014. This paper was recommended by Associate Editor L.-C. Wang.

D. Lin, T. Hong, Y. Li, and F. Fallah are with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305 USA (e-mail: linhai88@stanford.edu; hongted@gmail.com; yanjingl@stanford.edu; farzan.fallah@gmail.com).

S. Mitra is with the Department of Electrical Engineering and Department of Computer Science, Stanford University, Stanford, CA 94305 USA (e-mail: subh@stanford.edu).

E. S and S. Kumar are with Freescale Semiconductor, Noida 201301, India (e-mail: r01001@freescale.com; r2aaju@freescale.com).

N. Hakim and D. S. Gardner are with Intel Corporation, Santa Clara, CA 95054 USA (e-mail: nagib.hakim@intel.com; d.s.gardner@intel.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2014.2334301

cycles are highly challenging because it is extremely difficult to trace too far back in history for debug [53].

This paper presents the Quick Error Detection (QED) technique, which **systematically** transforms existing post-silicon validation tests into new QED family tests with extremely short error detection latencies and improved coverage. The QED family tests target logic and electrical bugs inside processor cores as well as bugs inside uncore components. In this paper, we demonstrate the effectiveness of QED for bugs inside cache controllers, memory controllers, and on-chip interconnection networks, in addition to bugs inside processor cores.

The main contributions of this paper are as follows.

1) We present a detailed description of *QED family tests*. QED family tests are a family of QED tests. Each QED test in the family is created by applying a set of QED transformations, together with specific values of corresponding QED transformation parameters, to the original test. The set of transformations chosen depends on the SoC component targeted (processor cores versus uncore components). The QED transformation parameters provide a systematic way of trading off error detection latency versus intrusiveness. Here, *intrusiveness* is defined as situations where the original test is able to detect a bug that is no longer detected by the QED transformed tests.
2) Using a state-of-the-art commercial multicore SoC hardware platform, we demonstrate the ability of QED family tests to systematically adjust tradeoffs between error detection latency and intrusiveness for an actual difficult logic bug.
3) We present results from multiple hardware platforms, as well as simulation results using realistic bug scenarios, to demonstrate the effectiveness of QED family tests in improving error detection latencies by up to nine orders of magnitude, and coverage by up to fourfold.

Our previous papers [28], [36] introduced the QED technique, and presented a list of bug scenarios obtained by collecting and analyzing "difficult" bugs from the bug databases of commercial multicore SoCs. This paper extends our previous papers in following ways.

1) We introduce QED family tests in detail.
2) We reinforce the results in [28] and [36] with more hardware results (including new logic bug results on a hardware platform) and extensive simulation results using an OpenSPARC T2-like SoC [48].

The rest of this paper is organized as follows. Section II presents a list of bug scenarios obtained by analyzing the bug reports of several state-of-the-art commercial multicore SoCs. In Section III, we show how traditional post-silicon validation tests can result in very long error detection latencies, and can also introduce excessive intrusiveness. Section IV presents our technique for systematically creating QED family tests for post-silicon validation. In Section V, we present a case study for an extremely difficult logic bug in a state-of-the-art commercial multicore SoC hardware platform. In Sections VI and VII, we present simulation results using an OpenSPARC T2-like multicore SoC and experimental results using the Intel® Core$^{\mathrm{TM}}$ i7 hardware platform. Related work is discussed in Section VIII, followed by the conclusion in Section IX.

TABLE I
(a) BUG ACTIVATION CRITERIA. (b) BUG EFFECTS

(a)

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* cycles.
	5. A sequence of load and/or store instructions in *X* clock cycles.
Processor cores	6. Data forwarding between pipeline stages.
	7. Two branch instructions in *X* clock cycles.
Other	8. A randomly chosen clock cycle.

(b)

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 cores	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.
** Include bugs inside cache controllers, memory controllers, and on-chip interconnection networks.

II. BUG SCENARIOS

To advance post-silicon validation research, it is crucial to have a list of bug scenarios that researchers can use to understand the limitations of existing techniques, and to evaluate new approaches. Toward this goal, we compiled a list of realistic bug scenarios by analyzing the bug reports (primarily logic bugs) of "difficult" bugs (i.e., bugs that took very long times to debug, as indicated in the bug reports) that occurred in several state-of-the-art commercial multicore SoCs.

We performed extensive analysis of the bug reports, and worked closely with validation teams to abstract the bug descriptions into bug scenarios using high-level descriptions while removing product-specific details. As a result, several actual bugs are abstracted into a single bug scenario. Each bug scenario consists of a bug activation criterion [Table I(a)] and a bug effect [Table I(b)].

Bug activation criterion refers to a set of conditions that must be satisfied to activate a bug scenario. In Table I(a), criteria 1–4 correspond to bugs inside cache controllers, criterion five corresponds to bugs inside cache controllers, memory controllers, and on-chip interconnection networks, and criteria 6–7 correspond to bugs inside processor cores. Activation criterion eight corresponds to scenarios where a bug is activated due to a random external (asynchronous) event, e.g., an external interrupt generated when the user pushes a button. These asynchronous events are difficult to capture in a simulation environment (e.g., it is difficult to simulate random user inputs in simulation), therefore, we abstracted this activation criterion as a randomly chosen clock cycle since these asynchronous events appear random to the simulated system.

Since we abstracted the activation criteria from "informal" bug reports, it is possible that all activation conditions might

TABLE II
SUMMARY OF LOGIC BUG COLLECTIONS

Bugs	Source	Bug description	Components targeted	Subsumed by this paper?
This paper	Bug reports for industrial SoCs	High level, implementation -independent	Processor core and uncore components	N.A.
[26]	Research chip	High level, implementation -independent	Processor core	Yes
[59]	Class project	Implementation -dependent	Processor core	N.A.
[61]	Class project	High level, implementation -independent	Processor core	Yes
[17], [18]	Errata pages	High level, implementation -independent	Cache	Yes
[15]	RTL errata	RTL-specific	Processor core	N.A.

not have been completely captured. For activation criteria 1–5 and 7, we include a positive integer parameter X that is adjusted to create a family of bug activation criteria from a single bug activation criterion. The minimum value that X can take is 2 (i.e., two store or load operations in two clock cycles; or two branch instructions in two clock cycles), but there is no maximum value for X.

Bug effect refers to the incorrect behavior resulting from bug activation. In Table I(b), effects A–E correspond to bugs inside cache controllers, effect F corresponds to bugs inside memory controllers, effect G corresponds to bugs inside interconnection networks, and effects H–J correspond to bugs inside processor cores. Bug effect D has a positive integer parameter Y that can be adjusted to create a family of bug effects. The minimum value that Y can take is 1 (i.e., delayed by one clock cycle), but there is no maximum value for Y.

The descriptions in Table I(a) and (b) allow us to implement each bug scenario using RTL or micro-architectural simulators (details in Section VI). One can create families of bug scenarios by adjusting the parameters X and Y in Table I(a) and (b). For example, pairing bug activation criterion two using $X = 2$, with bug effect A produces the bug scenario 2A.

Two stores in 2 cycles to the same cache line result in cache coherence message for that line to be dropped.

While there is on-going work on understanding electrical bug behaviors [21], [41], there exists little consensus on what constitutes accurate logic bug models [29]. Earlier researchers have analyzed logic bugs found in research chips, class projects, and errata pages [15], [17], [18], [26], [59], [61]. Bug scenarios resulting from Table I(a) and (b) subsume bugs in [17], [18], [26], and [61]. It is difficult to compare bugs in [15] and [59] versus bugs in Table I(a) and (b). This is because [15] provides RTL-specific bug examples, and [59] focuses on implementation-dependent root-cause analysis, e.g., missing inputs and incorrect signal sources. Table II summarizes our bug scenarios and earlier efforts to collect logic bugs.

III. LONG ERROR DETECTION LATENCY CHALLENGE

During post-silicon validation, a variety of tests such as random instruction tests, architecture-specific focused tests, instruction traces, and end-user applications are run on manufactured ICs to detect bugs [3], [32]. Given the long runtimes of such tests, it is generally difficult to run such tests during the pre-silicon phase. We provide an example in Fig. 1 (based on bug scenario 3C) to illustrate the long error detection latencies as well as the intrusiveness that may be incurred by traditional post-silicon validation tests. The example shown in Fig. 1 is from a real bug found during post-silicon validation of a state-of-the-art commercial multicore SoC. Consider a multicore SoC where each processor core can execute one instruction per clock cycle, and each processor core has a private L1 cache using write-through policy [25] (not shown in Fig. 1). The SoC has a shared L2 cache using write-through and write-allocate policy [25]. The following shows bug scenario 3C with $X = 2$.

Two stores in 2 clock cycles to adjacent cache lines result in the next store operation to not allocate a cache line.

Bug scenario 3C with $X = 2$ is only activated when two store operations occur to adjacent cache lines within two clock cycles. Therefore, as shown in Fig. 1(a), when Core 3 performs a store operation to a memory location A followed by a store to memory location B in 2 clock cycles, bug scenario 3C is activated. When executing the store to memory location B, if B is not already cached, a new cache line needs to be allocated for B in the shared cache (i.e., the cache controller needs to pick a free cache line for memory location B or evict an existing cache line if a free cache line is not available). However, due to bug scenario 3C, a new cache line is not allocated for B. Instead, an existing cache line is erroneously used (i.e., one that caches the value of an *arbitrary* memory location, in this example it is the cache line corresponding to some arbitrary memory location C). The store operation erroneously overwrites the existing cache line caching memory location C, corrupts the cached value, and marks it as modified. Note that, because of the write through policy of the shared cache, the value of B is stored correctly in the main memory.

After a very long time, Core 7 loads the corrupted value corresponding to memory location C from the shared cache (since the cache line is in modified state in the shared cache) and uses this corrupted value in its computations. Consequently, Core 7 produces incorrect results. As shown in Fig. 1(a), tests that rely on end result checks, which check for expected output values upon test completion, result in very long error detection latencies.

As shown in Fig. 1(b), the corrupted value read by Core 7 can result in a livelock/deadlock when used by code that performs locking. Techniques for detecting livelocks/deadlocks [7], [12] are inadequate in shortening error detection latency in this example because the error detection latency is already very long when the deadlock occurs (i.e., long after the cached value corresponding to memory location C is corrupted).

Self-checking tests that focus on bugs inside processor cores are not sufficient to reduce error detection latency and may also introduce excessive intrusiveness. Such tests check the results of instructions by inserting self-checking instructions that compare the results against a golden model (i.e., from simulation) [4] or against results of other instructions executed on the processor cores [52], [63].

Fig. 1(c) shows an example where a self-checking test detects the bug with a very long error detection latency. Since

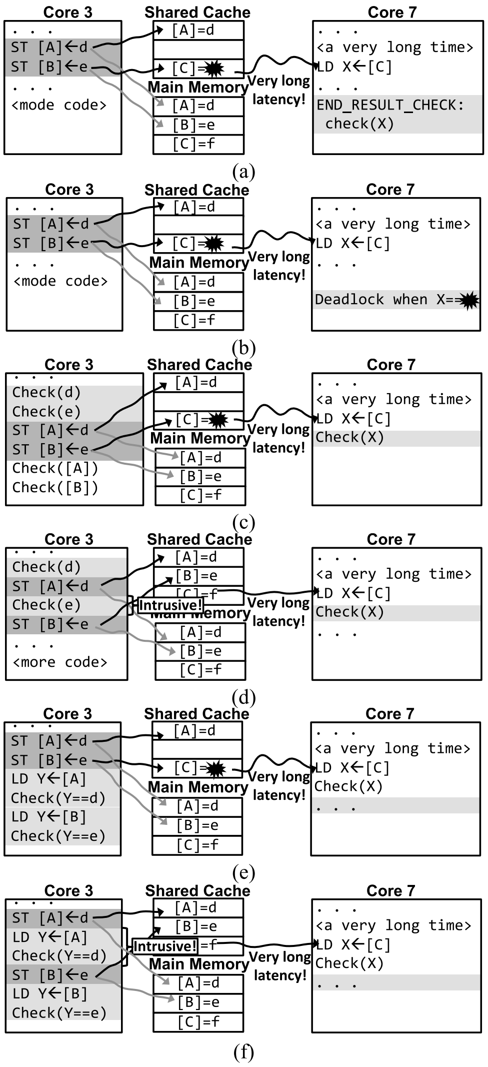


Fig. 1. Examples of long error detection latency and intrusiveness of traditional post-silicon validation tests. (a) End result checks with long error detection latency. (b) Deadlock detection with long error detection latency. (c) Self-checking test with long error detection latency. (d) Self-checking test with excessive intrusiveness. (e) Store readback test with long error detection latency. (f) Store readback test with excessive intrusiveness.

the bug only corrupts the cache value corresponding to an arbitrary memory location *C* in the shared cache, Core 3's self-checks do not detect the error when they check memory locations *A* and *B*, (since both are correct). It takes a very long time before Core 7 performs a load from memory location *C* and detects the error.

Self-checking tests can also introduce excessive intrusiveness, for example, by inserting a check instruction between the store to memory location *A* and store to memory location *B* [Fig. 1(d)]. This disrupts the activation criterion of two stores to adjacent cache lines within two clock cycles. Therefore, the bug is not activated and cannot be detected.

Another self-checking test variant, referred to as store readback test [Fig. 1(e)], performs a load operation on the same core that performed the store operation (Core 3 in this case) to check if the loaded value matches the stored value. As shown in Fig. 1(e), the error detection latency is still very long since neither memory location *A* nor *B* is corrupted. It takes a very long time before Core 7 performs a load from memory location *C* and detects the error.

TABLE III
QUALITATIVE COMPARISON OF POST-SILICON VALIDATION TESTS

Techniques	Error detection latency	Configurable intrusiveness	Simulation required for expected results?	Simulation required for indirect branch targets?
End result check	Can be very long	No	Sometimes	No
Deadlock detection [7], [12]	Can be very long	No	No	Sometimes*
Self-checking tests [4], [52], [63]	Can be very long	No	Sometimes	Sometimes*
Store readback tests [63]	Can be very long	No	No	Sometimes*
QED family tests	Short: bounded and configurable (Section IV.D)	Yes, systematic (Section IV.D)	No	Sometimes*

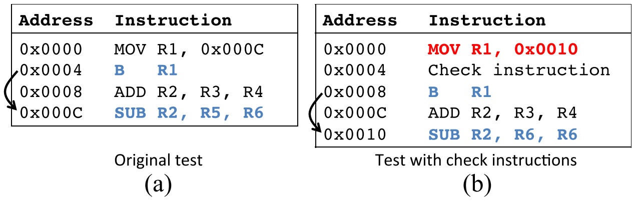
* When applied to test where only the test's binary executable is available, these techniques may require simulation to obtain the target addresses of indirect branch instructions.

Similar to Fig. 1(d), a store readback test can also introduce intrusiveness. For example, as shown in Fig. 1(f), such a test may choose to insert load and check instructions between the two store operations; as a result, the activation criterion may be disrupted, and the bug may not be activated and detected.

QED overcomes the long error detection latency challenge by creating tests that quickly detect bugs and simultaneously improve coverage. QED family tests also provide a systematic way to adjust the intrusiveness (with trade-offs versus error detection latency). Furthermore, QED family tests do not require simulation to obtain expected (i.e., golden) results. Table III provides a qualitative comparison of traditional post-silicon validation tests versus QED family tests (details in Section IV).

As shown in Table III, in situations where only the test's binary executable is available, techniques such as deadlock detection, self-checking tests, store readback tests, and QED family tests may require simulation to obtain the target addresses of indirect branch instructions (i.e., branch instruction where the target address is stored in a register). This is necessary because these techniques insert extra check instructions in the middle of the test that may displace the addresses of existing instructions in the test. Therefore, the target addresses of branch instructions must be changed to branch to the new addresses (after the incorporation check instructions). An example is shown in Fig. 2. Fig. 2(a) shows the addresses and instructions in the original test. The original test contains an indirect branch instruction at address 0x0004, which branches to the SUB instruction at target address 0x000C (the target address is stored in register R1). However, after the insertion of a check instruction at address 0x0004 in Fig. 2(b), the address of the SUB instruction is displaced to 0x0010. Therefore, the indirect branch instruction at address 0x0008 must branch to address 0x0010 (instead of address 0x000C). To identify the target address of the indirect branch instruction, simulation may be required to identify the value of R1 when the indirect branch instruction executes.

This is not necessary for direct branch instructions because the target address of direct branch instruction is encoded in the instruction itself and can be readily identified when the



```
Address  Instruction
0x0000   MOV R1, 0x000C
0x0004   B   R1
0x0008   ADD R2, R3, R4
0x000C   SUB R2, R5, R6
```

Original test

(a)

```
Address  Instruction
0x0000   MOV R1, 0x0010
0x0004   Check instruction
0x0008   B   R1
0x000C   ADD R2, R3, R4
0x0010   SUB R2, R6, R6
```

Test with check instructions

(b)

Fig. 2. Example where simulation may be required to determine the target address of indirect branch instruction. (a) Original test. (b) Test with check instructions.

instruction is disassembled. This is also not necessary if the techniques are applied at the source code level (C or assembly code) since the target addresses are known via labels in the source code. Finally, this may not be required for end result checks if the check instructions are only added at the end of the original test (hence existing instructions in the test are not displaced by the check instructions).

IV. Quick Error Detection

Given a post-silicon validation test (referred to as an "original" test in this paper) and a target error detection latency constraint (expressed as the number of clock cycles), QED systematically transforms original tests into new *QED tests* using a variety of *QED transformations*. The resulting QED tests have bounded error detection latencies. The target error detection latency constraint is determined by the bug localization technique used (e.g., trace buffers or simulation). In general, error detection latency should be less than 1,000 cycles because trace buffers record on the order of 1,000 cycles of history [2], [49]. In addition, if an intrusiveness constraint is available (e.g., if known *a priori* that a specific sequence of branch instructions or load/store instructions has a high probability of activating bugs), such constraints can be provided (i.e., as comments in the test instructions) so that the QED transformations do not modify the sequence and do not introduce excessive intrusiveness. However, if such intrusiveness constraints are not available, the intrusiveness can be systematically adjusted using the QED transformation parameters (details in Section IV-D).

From a single original test, one can create a family of QED tests (referred to as QED family tests) by selecting various QED transformations (details in Sections IV-A–C) and QED transformation parameters (details in Section IV-D) for each test in the family. Each of the tests in the family can have different tradeoffs between error detection latency, amount of intrusiveness, and the SoC components that it targets (i.e., processor cores versus uncore components).

While the EDDI-V, CFCSS-V, CFTSS-V, and PLC software-only QED transformations presented here are inspired by software implemented hardware fault tolerance (SIHFT) techniques [39] from fault-tolerant computing, there are important differences between transformations for post-silicon validation and transformations for fault-tolerant computing.

1) During post-silicon validation, reducing error detection latency is extremely important, because debug time rather than test execution time is the major bottleneck [30]. Therefore, some test execution time overhead may be acceptable during post-silicon validation if error detection latency is significantly reduced. This is because some overhead in test execution time (e.g., on the order of minutes to hours) but significantly shortened error detection latency may result in significant improvements in debug time (e.g., on the order of days).
2) Transformations for post-silicon validation tests must not cause excessive intrusiveness, which can degrade coverage of the post-silicon validation tests. For example, as shown by the bug example in Section III, if a transformation causes the test to no longer activate a bug, then the bug cannot be detected and the coverage of the test is degraded. This is not a primary concern in fault-tolerant computing, especially when transient errors are targeted.
3) During post-silicon validation, test inputs may be known *a priori* [9]. This presents a unique opportunity to optimize QED transformations for the corresponding test inputs in order to improve error detection latency and reduce intrusiveness. For example, if it is known *a priori* the number of iterations that a loop will execute, the loop can be unrolled to satisfy QED transformation parameters for error detection latency and intrusiveness (details in Section IV-D).

Sections IV-A–C present four software-only QED transformations: Error Detection using Duplicated Instructions for Validation (EDDI-V), Proactive Load and Check (PLC), Control Flow Checking using Software Signatures for Validation (CFCSS-V), and Control Flow Tracking using Software Signatures for Validation (CFTSS-V). These software-only QED transformations do not require any hardware modifications, and are readily applicable to existing post-silicon validation flows. QED also can be augmented with small hardware support (details in [28]). Section IV-D presents details for the QED transformation parameters *Inst_min* and *Inst_max*.

As stated above, multiple QED family tests can be created from a single original test. For example, given an original test *T*, the following QED family tests: *T_eddiv*, *T_cfcssv*, *T_cftssv*, and *T_plc*, can be created by transforming *T* using the EDDI-V, CFCSS-V, CFTSS-V, or PLC transformations, respectively. Furthermore, multiple QED transformations can be used to create a single QED family test. For example, *T* can be transformed using both the EDDI-V and CFCSS-V transformations to create *T_eddiv_cfcssv*, or *T* can be transformed using the CFTSS-V and PLC transformations to create *T_cftssv_plc*. For each of the *T_eddiv*, *T_cfcssv*, *T_cftssv*, *T_plc*, *T_eddiv_cfcssv, T_cftssv_plc* tests, we can select a range of values for the QED transformation parameters *Inst_min* and *Inst_max* (details in Section IV-D) and vary the "window" for QED transformations (details in Section IV-D) to create multiple QED family tests. An example demonstrating how QED family tests are created from a single original test *T* is shown in Fig. 3. The EDDI-V, CFCSS-V, CFTSS-V, PLC, EDDI-V + CFCSS-V, and CFTSS-V + PLC transformations are used. For each transformation, we select a range of values for *Inst_min,* from *Inst_min* = 1 to *Inst_min* = 1,000; and a range of values for *Inst_max*, from *Inst_max* = 1 to *Inst_max* = 1,000, to create QED family tests. We also vary the window for each test in the family.

A. Error Detection Using Duplicated Instruction for Validation (EDDI-V)

As shown in Section III, post-silicon validation tests can result in extremely long error detection latencies. Furthermore,

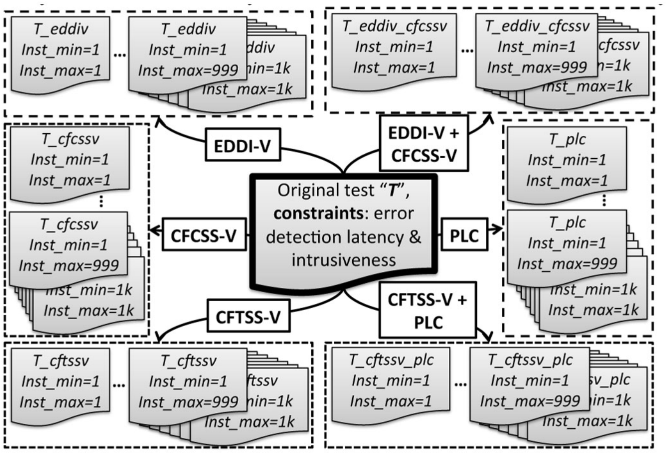
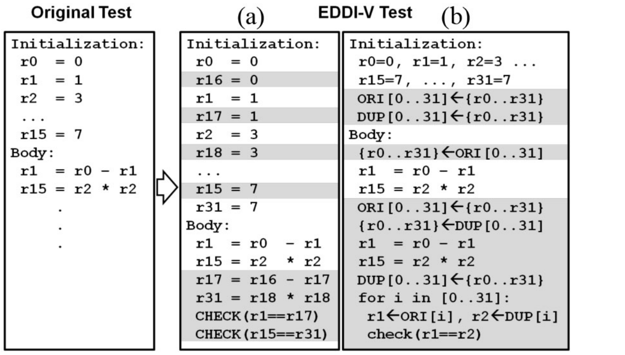


Fig. 3. Example of QED family tests. Here, a single original test "*T*" is supplied as the input, along with the constraints on error detection latency and intrusiveness (if available). We select the EDDI-V, CFCSS-V, CFTSS-V, PLC, EDDI-V + CFCSS-V, and CFTSS-V + PLC transformations, along with a range of values for *Inst_min*, from *Inst_min* = 1 to *Inst_min* = 1,000 (1 k), and a range of values for *Inst_max*, from *Inst_max* = 1 to *Inst_max* = 1,000 (1 k) to create multiple tests in the family. We also vary the window for each test in the family.

such tests can also result in excessive intrusiveness that may adversely impact test coverage. In contrast, EDDI-V (which extends the EDDI technique used in fault-tolerant computing [44]) bounds error detection latency (details in Section IV-D) for bugs that occur inside processor cores, and provides configurability in the amount of intrusiveness introduced (details in Section IV-D).

EDDI-V is implemented by first reserving half of the registers and memory space for the instructions from the original test (referred to as "original" instructions in this paper), while the other half is used by the duplicated instructions created by the EDDI-V transformation (details later). The registers and memory for the original and the duplicated instructions are initialized to the same values. For example, in Fig. 4(a), the original test uses 16 registers (r0–r15). Therefore, the EDDI-V transformation reserves 16 registers (r16–r31) and initializes them to the same values as those in r0–r15. In situations where there are insufficient registers (i.e., if the original test needs to use all of the available registers), we reserve memory to temporarily store register values.[1] For example, in Fig. 4(b), the original test needs to use all 32 registers available in the system (r0–r31). Therefore, the EDDI-V transformation reserves two memory blocks, *ORI[0..31]* and *DUP[0..31]*, to store the original and duplicated register values, respectively. Then, as shown in Fig. 4(b), before executing a block of original instructions, the original register values are loaded from the memory block *ORI* back into the registers. After a block of original instructions has executed, the register values are stored back into the memory block *ORI*. Then the duplicated register values are loaded into the registers from the memory block *DUP*. The duplicated instructions inserted by EDDI-V are executed, and the register values are stored back into the memory block *DUP*.

As shown in Fig. 4, the EDDI-V transformation then creates duplicated and check instructions. For every arithmetic, logical, shift, or move instruction in the original test, a corresponding duplicated instruction is created that operates on the corresponding duplicated registers reserved for EDDI-V (or the duplicated register values, if registers are stored in memory). A check instruction is created that compares the destination register values of the original and duplicated instructions. Any mismatch in the comparison indicates an error is detected. Similarly, for every load and store instruction, a corresponding duplicated instruction is created that operates on the duplicated registers (or register values) and the memory reserved for EDDI-V. A check instruction is created that compares the values stored and loaded by the original instruction and the corresponding values stored and loaded by the duplicated instruction. Any mismatch in the comparison indicates an error is detected. The frequency that duplicated instructions and check instructions are inserted in the test is determined by the QED transformation parameters *Inst_min* and *Inst_max* (details in Section IV-D).



```
Initialization:
 r0  = 0
 r1  = 1
 r2  = 3
 ...
 r15 = 7
Body:
 r1  = r0 - r1
 r15 = r2 * r2
      .
      .
      .
```

```
Initialization:
 r0  = 0
 r16 = 0
 r1  = 1
 r17 = 1
 r2  = 3
 r18 = 3
 ...
 r15 = 7
 r31 = 7
Body:
 r1  = r0  - r1
 r15 = r2  * r2
 r17 = r16 - r17
 r31 = r18 * r18
 CHECK(r1==r17)
 CHECK(r15==r31)
```

```
Initialization:
 r0=0, r1=1, r2=3 ...
 r15=7, ..., r31=7
 ORI[0..31]←{r0..r31}
 DUP[0..31]←{r0..r31}
Body:
 {r0..r31}←ORI[0..31]
 r1  = r0 - r1
 r15 = r2 * r2
 ORI[0..31]←{r0..r31}
 {r0..r31}←DUP[0..31]
 r1  = r0 - r1
 r15 = r2 * r2
 DUP[0..31]←{r0..r31}
 for i in [0..31]:
  r1←ORI[i], r2←DUP[i]
  check(r1==r2)
```

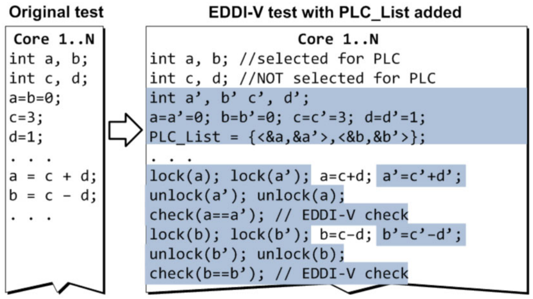
Fig. 4. Examples of the EDDI-V transformation with *Inst_min* = *Inst_max* = 2. (a) With half of all registers reserved for EDDI-V. (b) With register values stored temporarily in memory blocks *ORI* and *DUP*.

EDDI-V can be enhanced using diversity techniques (e.g., based on the principles of [45]) to ensure that the instructions inserted by QED execute differently as compared to the instructions from the original test. Such diversity techniques reduce the likelihood that the original instructions and the duplicated instructions are affected in identical ways by logic bugs and electrical bugs. Furthermore, our results in Sections VI and VII also show that EDDI-V inherently introduces timing diversity between executions of the original instructions and the duplicated instructions, especially for difficult logic bugs and electrical bugs.

B. PLC

PLC is a QED transformation technique that bounds error detection latency for bugs inside the uncore components (e.g., cache controllers, memory controllers, interconnection networks). Unlike EDDI-V, the PLC transformation does not solely rely on (identical or diverse) reexecution of instructions in the original test. Instead, it inserts special PLC operations at very fine granularities across memory (and I/O) spaces using targeted instructions. These PLC operations perform loads on all threads executing on all processor cores from a selected set of variables and perform self-consistency checks on those variables. The PLC transformation is compatible with the EDDI-V transformation; hence, bugs inside processor cores can also be detected with short error detection latencies.

[1] However, one must be careful about possible intrusiveness due to additional store and load instructions.



```
Core 1..N
int a, b;
int c, d;
a=b=0;
c=3;
d=1;
. . .
a = c + d;
b = c – d;
. . .
```

```
Core 1..N
int a, b; //selected for PLC
int c, d; //NOT selected for PLC
int a’, b’ c’, d’;
a=a’=0; b=b’=0; c=c’=3; d=d’=1;
PLC_List = {<&a,&a’>,<&b,&b’>};
. . .
lock(a); lock(a’); a=c+d; a’=c’+d’;
unlock(a’); unlock(a);
check(a==a’); // EDDI-V check
lock(b); lock(b’); b=c–d; b’=c’–d’;
unlock(b’); unlock(b);
check(b==b’); // EDDI-V check
```

Fig. 5. PLC transformation. Step 1: Initialization.

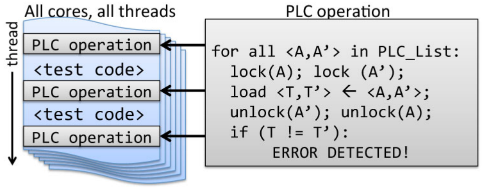
In the next subsections, we provide a detailed overview of the steps required to implement the software-only PLC transformation for a multicore and multithreaded SoC, such as the OpenSPARC T2 SoC. However, PLC can be further enhanced with hardware support (details are beyond the scope of this paper).

1) Step 1 (Initialization): Given an original test, we first perform the EDDI-V transformation to bound error detection latencies for bugs inside processor cores. Then a list 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 in the original test 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. 5). The values of the pointers are obtained either via source code labels (for statically allocated variables) or through function calls to the memory allocation function (for dynamically allocated variables). The pointer values are used to determine the memory addresses of variables during PLC operations.

For multithreaded tests, 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 four operations: store to an original variable in the *PLC_List*, store to the corresponding EDDI-V variable, load from the original variable in the *PLC_List*, and load from the corresponding EDDI-V variable. We achieve this by locking the original variable and the corresponding EDDI-V variable pair during store and load operations to the pair (Fig. 5).

2) Step 2 (PLC Operation Insertion): The PLC transformation inserts PLC operations in each thread running on each processor core. Each PLC operation (Fig. 6) performs the following functions. It iterates once over all tuples in *PLC_List*. For each tuple in the list, it locks the tuple and loads the values pointed to by 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. Any mismatch indicates an error is detected. 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 exactly the same data values.

The 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, the 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 a bug that creates an error in a different variable (e.g., the example provided in Section III). The intervals for which the PLC operations are inserted in the test are determined by the QED transformation parameters *Inst_min* and *Inst_max* (details later).



```
for all <A,A’> in PLC_List:
  lock(A); lock (A’);
  load <T,T’> ← <A,A’>;
  unlock(A’); unlock(A);
  if (T != T’):
      ERROR DETECTED!
```

Fig. 6. PLC transformation. Step 2: PLC operation insertion.

3) Variable Selection Strategies for PLC: There are 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 the PLC operations must load and check all variables in the *PLC_List*, which can take a long time.
2) One can create a family of tests from an original test, where each test in the family selects only a subset of variables (from the original test) for its *PLC_List*. In this case, a single test in the family may not be able to detect all bugs in a system. However, our results in Section VI demonstrate that the **entire family** of PLC-based QED family tests is highly effective in improving coverage and also shortening error detection latencies.
3) If the inputs to a test are known *a priori* (which is often the case for post-silicon validation tests [9]), one can perform profiling to determine variables with store-to-load or load-to-load latencies (defined as the number of clock cycles elapsed between a store/load instruction to a variable and a subsequent load to the same variable) longer than the desired error detection latency bounds, and include only those variables in the *PLC_List.*

C. CFCSS-V and CFTSS-V

CFCSS-V and CFTSS-V are two QED transformations that target bugs that affect a processor core's control flow. CFCSS-V is inspired by CFCSS [46] used in fault-tolerant computing. Like CFCSS, CFCSS-V checks the control flow of a test program during runtime against the control flow graph constructed during the test program's compile time [46]. However, the major difference between CFCSS for fault-tolerant computing and CFCSS-V for post-silicon validation is that CFCSS-V supports configurability to trade off error detection latency for intrusiveness. To achieve this, instead of checking control flow between basic blocks [46], CFCSS-V checks control flow between "blocks of instructions" that may contain arbitrary number of instructions, as determined by the QED transformation parameters *Inst_min* and *Inst_max* (details in Section IV-D). By adjusting *Inst_min* and *Inst_max*, the number of instructions in a "block of instructions" can range from a single instruction (i.e., short error detection latency, but

```
ST [TEMP_VARIABLE], R1
LI R1, SOFTWARE_SIGNATURE
ST [CFTSS_V_SIGNATURE], R1
LD R1, [TEMP_VARIABLE]
```

Fig. 7. Pseudo assembly code for the CFTSS-V operation inserted at the beginning of each "block of instructions."

high intrusiveness) to multiple basic blocks (i.e., longer error detection latency, but low intrusiveness).

CFCSS-V is implemented by splitting the original test into "blocks of instructions" as determined by *Inst_min* and *Inst_max*. Each "block of instructions" is assigned a unique integer (i.e., the software signature). Algorithms for assigning signatures are found in [38], [46], and [54]. Then, the control flow graph between the blocks of instructions is constructed following the algorithm presented in [46]. Finally, code that checks the control flow during runtime against the control flow graph is inserted at the beginning of each "block of instructions" using the technique presented [46].

CFTSS-V is variant of CFCSS-V that tracks the execution of instructions using special software signatures inserted into the test code, but does not perform control flow checking. The CFTSS-V transformation is performed by first declaring a global variable *CFTSS_V_SIGNATURE* stored in memory that holds the current runtime signatures of the program. Next, the instructions in the test are divided into "blocks of instructions" as determined by the QED transformation parameters *Inst_min* and *Inst_max* (details in Section IV-D). Then a unique integer number (i.e., the software signature) is assigned to each "block of instructions." The algorithm for assigning the software signature can be found in [38], [46], and [54].

The CFTSS-V transformation then inserts CFTSS-V operations to the beginning of each "block of instructions." Each CFTSS-V operation stores the assigned software signature of the "block of instructions" to the global variable *CFTSS_V_SIGNATURE*. Fig. 7 lists the pseudo assembly code for the CFTSS-V operation. In Fig. 7, [*TEMP_VARIABLE*] is a designated memory location used to store the saved value of R1. This is needed because the CFTSS-V transformation must not alter the value of the registers in the original test. *SOFTWARE_SIGNATURE* is the unique software signature assigned to the "block of instructions" using the signature assignment algorithm found in [46], and [*CFTSS_V_SIGNATURE*] is the designated memory location to hold the current runtime signature of the test. When a failure occurs (e.g., livelock or deadlock) the content of [*CFTSS_V_SIGNATURE*] is used to determine the last "block of instructions" that was executed by the processor core before the failure.

D. QED Transformation Parameters: Inst_min and Inst_max

QED transformations support systematic tradeoffs between error detection latency and intrusiveness. This is achieved by two QED transformation parameters: *Inst_min* and *Inst_max*, which correspond to the minimum and maximum number of original instructions executed before any instructions inserted by the QED transformations execute (*Inst_min* must be less than or equal to *Inst_max* by definition). Increasing *Inst_min* reduces intrusiveness, and vice-versa. Decreasing *Inst_max* decreases the error detection latency, and vice-versa. Note that, unlike *Inst_max*, which can always be satisfied, *Inst_min* is a "soft constraint": although we make a best effort to satisfy *Inst_min*, there are some cases in which this cannot be done. For example, *Inst_min* cannot be satisfied if the original code has fewer than *Inst_min* instructions. Furthermore, an excessively large *Inst_min* may degrade the coverage benefits of QED. For example, if *Inst_min* is excessively large, an error caused by a bug may get masked before it is detected by QED.

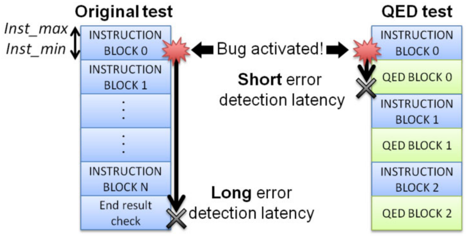


Fig. 8. Error detection latency of the original test versus the QED test.

During post-silicon validation, test program inputs may be known *a priori* [9]. Therefore, we can utilize code analysis techniques to ensure that the QED transformation parameters *Inst_min* and *Inst_max* can be satisfied even if the original test contains loops, conditional branches, and synchronization primitives such as locks. For example, a small loop may contain fewer instructions than the desired *Inst_min*. In this case, the loop is unrolled so that multiple iterations are executed without intervening branches. This way, larger blocks consisting of more than *Inst_min* instructions can be constructed. Likewise, it may not be possible to divide a loop body containing more than *Inst_max* instructions into blocks each with less than *Inst_max* and more than *Inst_min* instructions. In this case, the loop is unrolled until the unrolled loop body can be divided into blocks of instructions that satisfy the *Inst_max* and *Inst_min*. For conditional branch instructions, we consider each branch (including the branches of any nested conditional branch instructions) separately, and divide the instructions of each branch into blocks of instructions that satisfy both *Inst_min* and *Inst_max*. For locks, the original and duplicated code blocks are enclosed by the by the same set lock and unlock instructions.

As illustrated in Fig. 8, instructions from the original test are divided into "blocks of instructions" (determined by *Inst_min* and *Inst_max*). The error detection latency is bounded by the sum of two terms.

1) The time it takes to execute a block of original instructions (i.e., bounded by *Inst_max*).
2) The time it takes to execute the corresponding QED block (i.e., EDDI-V instruction duplication and check, PLC operation, CFCSS-V, or CFTSS-V operation).

This can provide a great reduction in error detection latency compared to the original test, which may detect errors only after a visible failure (e.g., program crash) or using its original checks (if available, e.g., end result checks that compare actual program outputs to expected outputs), both of which can be unbounded. QED transformations also create a family of tests (i.e., QED family tests) from a single original test by systematically selecting different *Inst_min* and *Inst_max* for each test in the family. This provides systematic tradeoffs between the error detection latency and the amount of intrusiveness.

For a selected pair of *Inst_min* and *Inst_max*, multiple QED family tests can be created by changing the window of instructions that are included in a "block of instructions."

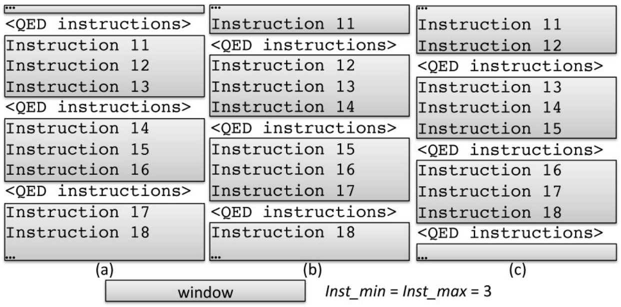


Fig. 9. Three possible "windows" for *Inst_min* = *Inst_max* = 3. (a) Window 1. (b) Window 2. (c) Window 3.

TABLE IV
COMPARISON OF QED TRANSFORMATIONS

	EDDI-V	PLC	CFCSS-V / CFTSS-V
Pros	Short error detection latency and high coverage for bugs inside processor cores.	Short error detection latency and high coverage for bugs inside uncore components.	Small amount of code change. Small increase in test runtime.
Cons	May have long error detection latencies for bugs inside uncore components.	May have very long test runtime (can be minimized with hardware support).	May not detect bugs that do not affect a processor's control flow.
Types of bugs targeted	Bugs inside processor cores (also some bugs inside uncore components, but may have long error detection latencies for uncore bugs).	Bugs inside uncore components such as cache controllers, memory controllers, on-chip interconnection networks.	Bugs that affect a processor's control flow (i.e., branch to wrong address livelock, and deadlock).

We define the window as a consecutive sequence of instructions that are grouped together into a "block of instructions." For example, Fig. 9 illustrates three different windows for *Inst_min* = *Inst_max* = 3. In Fig. 9, each "block of instructions" for test (a), (b), and (c) contains three instructions. However, the specific instructions that are grouped into a window are different for all three tests. For example, in test (a), the instructions {Instruction 11, Instruction 12, Instruction 13} are grouped into a window, where as in test (b) the instructions {Instruction 12, Instruction 13, Instruction 14} are grouped into window, and in test (c), the instructions {Instruction 13, Instruction 14, Instruction 15} are grouped into a window.

E. Summary and Comparison of QED Transformations

The above subsections presented four software-only QED transformations. Table IV provides a qualitative comparison of the QED transformations presented above, focusing on the pros and cons of each transformation and the types of bugs targeted by each of the transformations.

V. CASE STUDY: LOGIC BUG IN MULTICORE SOC HARDWARE

In this section, we present a case study to demonstrate the effectiveness of QED in detecting logic bugs in a complex multicore SoC hardware platform. The SoC contains six out-of-order superscalar processor cores. Each core has private L1 and L2 caches. The processor cores are connected to each other by a coherent interconnect fabric, which also connects the processor cores to other uncore components, such as shared L3 caches, memory controllers, network controllers, and I/O controllers. The name and model number of the SoC is omitted for confidentiality reasons. Fig. 10 shows a picture of the SoC in the validation lab during post-silicon validation.

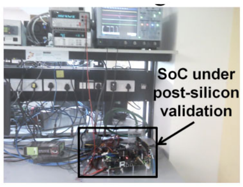


Fig. 10. Picture of the SoC during post-silicon validation.

The SoC has a logic bug that is very difficult to debug for the following reasons (the specific details of the bug are not disclosed due to confidentiality reasons).

1) The bug does not affect any architectural states, i.e., the bug does not modify any registers or memory locations. The only effect of the bug is to cause the processor core to deadlock, i.e., the processor core will not execute any instructions.
2) Originally, the bug was detected only by a ten-second timeout timer. This corresponds to error detection latency on the order of 15 billion clock cycles.
3) A very specific instruction sequence is required to activate the bug. If the test is modified (e.g., by truncating/removing instructions from the test or by modifying instructions in the test), excessive intrusiveness may be introduced, and the test will not be able to activate the bug.

We were not informed *a priori* the specific instruction sequence required to activate this bug. We obtained the original test (as a binary file) that was originally used to activate this bug. From the original test, we systematically created QED family tests by selecting various QED transformations and by systematically varying *Inst_min* for each test in the family. The QED transformations do not require any information about the bug, and we did not use any design-specific details (e.g., micro-architectural information) when we created the QED family tests. The following subsections detail the results of QED family tests created using the CFCSS-V, CFTSS-V, and EDDI-V transformations. PLC-based QED family tests were not used because the original test does not target uncore components.

A. CFCSS-V and CFTSS-V QED Family Tests Results

We created the CFCSS-V and the CFTSS-V QED family tests by varying *Inst_min* in order to systematically adjust tradeoffs between error detection latency and intrusiveness (determined by whether the deadlock on the processor core is preserved). We started with *Inst_min* = 1,000 instructions (the maximum desired error detection latency); for each test in the QED family tests, we reduced *Inst_min* in order to improve error detection latency (at the cost of possibly increased intrusiveness). For *Inst_min* $\leq$ 20, we also varied the transformation "windows" (Section IV-D) to create multiple tests for each *Inst_min*. Next, we ran each test in the family on the SoC platform to determine whether the deadlock in the processor core still occurred with the given *Inst_min*. For *Inst_min* $\leq$ 20,

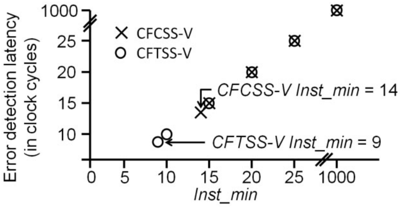


Fig. 11. Error detection latencies versus *Inst_min*. CFTSS-V detected the bug in nine clock cycles and CFCSS-V detected the bug in 14 clock cycles. For the CFTSS-V QED test, the bug is not activated when *Inst_min* = 8. For the CFCSS-V QED test, the bug is not activated when *Inst_min* = 13. For *Inst_min* = 20, the bug is considered as activated if at least one of the tests created using the transformation "window" method for that particular *Inst_min* activated the bug.

TABLE V
ERROR DETECTION LATENCIES FOR QED FAMILY TESTS

Test	Error detection latency (cycles)
Original test	~15 x 10^9
CFCSS-V QED family tests	14
CFTSS-V QED family tests	9

the bug is considered as activated if at least one of the tests created using the transformation "window" method for that particular *Inst_min* value has activated the bug. If the deadlock occurred, we recorded the "block of instructions" that CFCSS-V or CFTSS-V determined was the last block executed by the processor core before the deadlock occurred.

The results are presented in Fig. 11, where the horizontal axis represents *Inst_min* and the vertical axis represents error detection latency. CFTSS-V-based QED family tests detected the bug within nine instructions after it was activated (corresponding to an error detection latency of approximately nine clock cycles). This is nine orders of magnitude improvement in error detection latency compared to the original test. CFCSS-V-based QED family tests detected the bug within 14 instructions after it was activated. This is slightly longer than CFTSS-V-based QED family tests because the CFCSS-V transformation inserts more instructions into the test as compared to the CFTSS-V transformation.

Fig. 11 shows the systematic tradeoffs between error detection latency and intrusiveness using *Inst_min*. For this bug, CFTSS-V transformation with *Inst_min* smaller than nine instructions introduced excessive intrusiveness; therefore the bug is not activated and not detected when *Inst_min* is less than nine instructions. For the CFCSS-V transformation, *Inst_min* smaller than 14 instructions result in excessive intrusiveness. Adjusting *Inst_min* is different from truncating the test program (i.e., removing instructions from the test program to shorten error detection latency) since truncating the test program may remove instructions that are necessary to activate the bug. In contrast, adjusting *Inst_min* does not remove any instructions and ensures that the test program still contains the instructions required to activate the bug.

Table V summarizes the error detection latencies of the original test and QED family tests. We worked with the validation team to ensure that QED family tests indeed detected the difficult bug in question.

B. EDDI-V QED Family Tests Result

We also performed experiments to evaluate the effectiveness of using *Inst_min* to systematically adjust the intrusiveness of the EDDI-V transformation. Note that, since this bug does not impact any registers or memory locations, the EDDI-V only tests do not detect this bug (hence error detection latencies are not reported). However, we can still characterize the intrusiveness of the EDDI-V only tests by observing whether the deadlock still occurs for a given *Inst_min*.

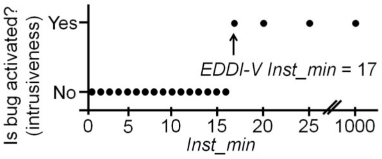


Fig. 12. Intrusiveness versus *Inst_min* tradeoff of EDDI-V QED family tests. For each *Inst_min*, the bug is considered as activated if at least one of the tests created using the transformation "window" method for that particular *Inst_min* activated the bug.

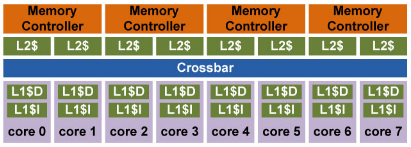


Fig. 13. OpenSPARC T2-like multicore SoC [48].

We created the EDDI-V-based QED family tests by sweeping the *Inst_min* parameter of the EDDI-V transformation. By sweeping the values of *Inst_min*, we can determine the values of *Inst_min* that preserve the deadlock in the processor core. We started with *Inst_min* = 1, which has the most intrusiveness but the smallest error detection latency, and increase *Inst_min* by 1 for each test in the family (i.e., *Inst_min* = 1, 2, 3...). The results are presented in Fig. 12. For each *Inst_min*, the bug is considered as activated if at least one of the tests created using the transformation "window" method for that particular *Inst_min* value has activated the bug. As shown in Fig. 12, the minimum *Inst_min* value that still preserves the deadlock is 17. This value is greater than the nine instructions for the CFTSS-V or the 14 instructions for the CFCSS-V-based QED family test because the EDDI-V transformation inserts more instructions in the test, and may cause more intrusiveness.

VI. OPENSPARC T2 SIMULATION RESULTS

To demonstrate the effectiveness of QED for a wide variety of bugs, we evaluated QED by simulating the bug scenarios in Section II using a micro-architectural simulator. We used Multifacet's general execution-driven multiprocessor simulator (GEMS) [40] to simulate an OpenSPARC T2-like SoC [48], a 500 million-transistor design with eight processor cores, 64 hardware threads, private L1 data and instruction caches, crossbar-based interconnects, eight-way banked L2 cache using directory-based cache coherence protocol, and four memory controllers (Fig. 13).

Table I(a) and (b) together produce 80 different bug classes (the cross product of eight bug activation criteria and ten bug effects in Table I(a) and (b)). The QED transformations do not need any information about the bug scenarios. For each activation criterion with a parameter *X*, we varied *X* from 2 to 1,000 clock cycles to create a family of activation criteria. We limited the minimum sweep range to two clock cycles because it is not possible for the simulated processor core to execute more than two load, store or branch instructions in one clock cycle [40]. We limited the maximum sweep range of X to 1,000 clock cycles because after X = 800 clock cycles, the number of times activation criteria 1–5 and 7 are satisfied does not significantly change as X is increased (100% of the load/store instructions satisfy activation criteria 1–5 and 100% of branch instructions satisfy activation criterion 7). This gives a total of nine different X values (i.e., X = 2, 5, 10, 20, 50, 100, 200, 500, 1,000 clock cycles to distribute the X values between 2 and 1,000). For bug effect D, we varied Y (the number of clock cycles delayed) from 1 clock cycle to 1,000 clock cycles to create a family of ten bug effects (i.e., Y = 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000 clock cycles for Y values between 1 and 1,000). This gives us a total of 19 different bug effects (bug effects A, B, C, E, F, G, H, I, and J plus ten bug effects resulting from bug effect D by varying Y). We chose the maximum Y value to be 1,000 cycles to ensure it is significantly larger than the maximum cache latency (16 cycles in our system). For each X value, there are a total of 152 distinct bug scenarios (i.e., cross product of eight bug activation criteria with 19 bug effects). This results in a total of 1,368 bug scenarios. QED transformation is agnostic (i.e., not specifically tailored) to the bug scenarios and is also agnostic to the X and Y parameters.

In this section, the bug scenarios correspond to logic bugs, which are always present in the system (i.e., they are not injected). A specific set of activation criteria has to be satisfied in order for a logic bug to cause an effect in the system. Therefore, the bug scenarios were simulated in the following manner: for each experiment, only one bug scenario is simulated. For a selected bug scenario, we modified the source code of the simulated system to include two additional routines: one that continuously monitors the simulated system for the activation criterion, and another that inserts the bug effect in the system (initially disabled). Whenever the activation criterion is satisfied, the routine that inserts the bug effect in the system is enabled to create the bug effect in the system. 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. The bug scenario affects all of the cores, L2 caches, and memory controllers (i.e., no specific component was preselected).

The results are summarized in Figs. 14 and 15 for FFT, LU, RADIX, and OCEAN programs from the SPLASH-2 benchmark suite [64] and a proprietary industrial post-silicon validation test targeting memory bugs (details omitted for confidentiality). We performed analysis on the SPLASH-2 tests by tabulating the percentage of various instructions in each test. Since the majority of the bug scenarios in Section II are related to uncore components (which communicate with the processor cores using only load/store instructions), we counted the percentage of load/store instructions out of the total number of instructions for each of the tests in the SPLASH-2 benchmark suite. We picked tests that have a low, medium, and high percentage of load/store instructions. The SPLASH-2 benchmark suite contains tests with percentage of load/store instruction that ranged from 10% to 19%. RADIX has the maximum percentage of load/store instructions (19%) out of all tests in the benchmark suite. LU and OCEAN have medium percentage of load/store instructions (15% and 13%). FFT has minimum percentage of load/store instructions (10%) out of all tests in the SPLASH-2 benchmark suite.

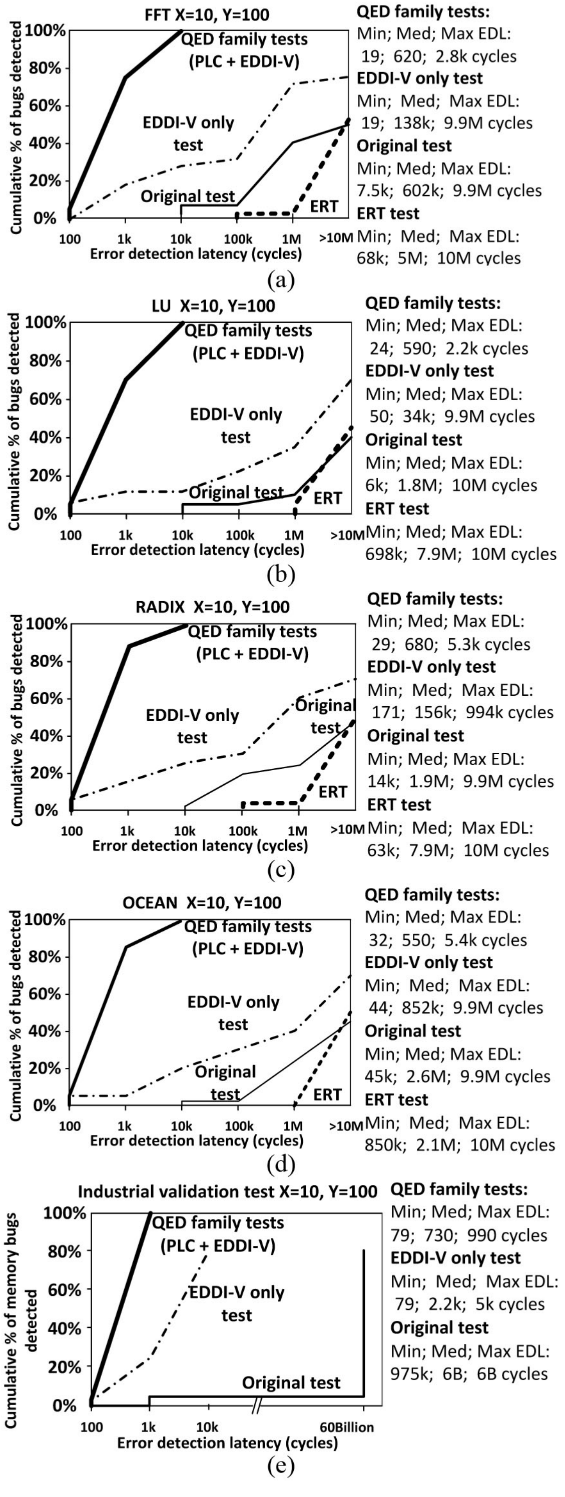


Fig. 14. Plot showing error detection latencies (EDL) and coverage of post-silicon validation tests for X = 10 and Y = 100. The vertical axis represents the **absolute** cumulative percentage (not normalized percentage) of bug scenarios detected with respect to the list of bug scenarios in Section II. The horizontal axis represents the error detection latencies. (a) FFT. (b) LU. (c) RADIX. (d) OCEAN from the SPLASH-2 benchmark. (e) Industrial validation test.

In Fig. 14, the vertical axes represent the **absolute** percentage of bug scenarios from Section II detected (not normalized to number of bugs detected by QED family tests), and the horizontal axes represent the error detection latencies in clock cycles. In Fig. 14, we set $X = 10$ clock cycles and $Y = 100$ clock cycles for the bug scenarios because these values gave us "close" representations (on our simulation platform) of the "difficult" bugs found in the bug databases. However, to show that QEDs improvements in error detection latencies and coverage are not dependent on the X and Y parameters, we also measured the error detection latencies and coverage by varying the X and Y parameters, which are reported in Fig. 15. In both Figs. 14 and 15, the constraints on error detection latency and intrusiveness are set using the QED transformation parameters (i.e., *Inst_min* = *Inst_max* = 10 lines of C source code).

In Fig. 15, the vertical axes represent the error detection latencies in clock cycles. The horizontal axes represent the X-values for the bug scenarios. In Fig. 15, for each X-value, there are 152 bug scenarios (obtained using different combinations of: bug activation criteria, bug effects, and the Y parameter as discussed above). Fig. 15 shows the minimum, median (represented by the square dot), and maximum error detection latencies. The results in Figs. 14 and 15 correspond to eight-threaded (single thread per core) versions of the tests. 64-threaded versions of the tests were not used because of the slow simulation speeds and because our simulator does not accurately simulate systems with 64 hardware threads [40]. In Figs. 14 and 15 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 QED family tests (PLC + EDDI-V) in Figs. 14 and 15, we performed the PLC- and the EDDI-V-based QED transformations at the C source code level with *Inst_min* = *Inst_max* = 10 lines of C source code. We split all the heap variables in the original test into 1,024 groups. This is done by aggregating a list that contains 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 family, each performing PLC operations for only the variables in its corresponding group. Register variables are not included for PLC because errors in these variables can be quickly detected by EDDI-V alone. Each test in the family is run one after another. For each bug scenario, the error detection latency reported in Figs. 14 and 15 corresponds to the error detection latency when the bug scenario was detected for the first time by the QED family tests. For the industrial test, there is a single QED family test consisting of all variables allocated in memory. Since this test targets memory bugs only, Figs. 14(e) and 15(e) only considers memory bug scenarios (i.e., activation criteria 1–5 in Table I(a) and effects A–G in Table I(b)). Figs. 14 and 15 also report results for EDDI-V only tests obtained by transforming the original tests using the only the EDDI-V transformation on the C source code.

Table VI shows the runtimes of the original tests, the EDDI-V only tests and the QED family tests with PLC and EDDI-V. For each row in Table VI, the runtime reported in each column is normalized to the runtime of the original test (i.e., the second column). The reported runtimes of QED family tests are the combined runtimes for all tests in the family (i.e., 1,024 tests in the family for FFT, LU, RADIX, OCEAN, and one test in the family for industrial validation test). The runtimes of QED family tests are significantly longer than

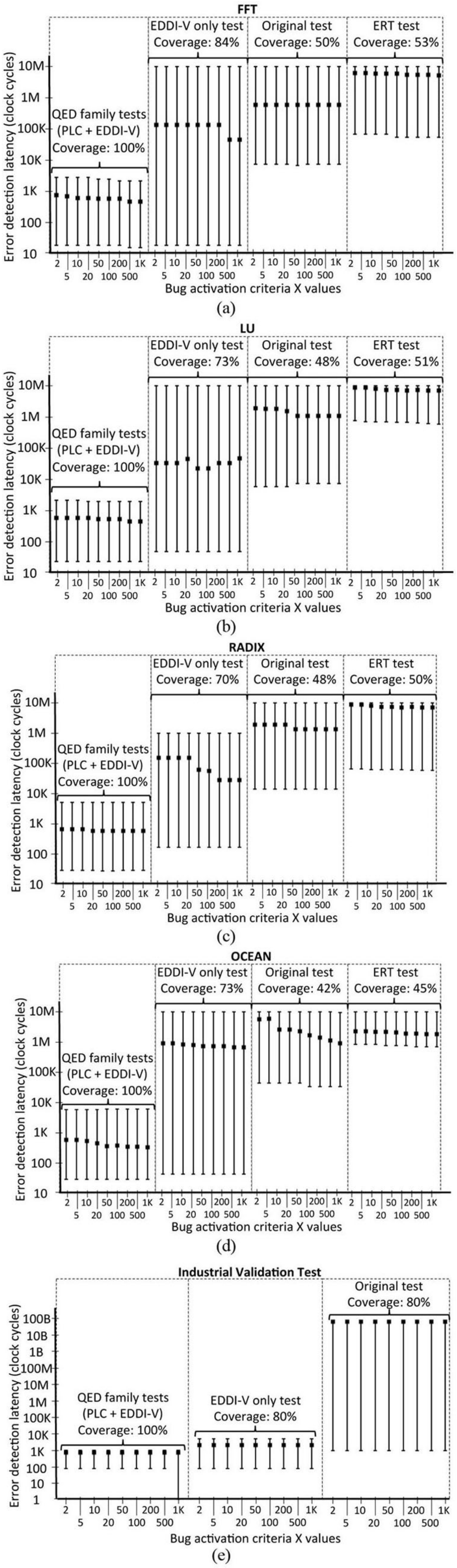


Fig. 15. Graph showing the minimum, median (represented by the square dot), and maximum error detection latencies by varying both X and Y parameters of the bug activation criteria and bug effects for (a) FFT, (b) LU, (c) RADIX, (d) OCEAN from the SPLASH-2 benchmark, and (e) industrial validation test. The coverage numbers reported are **absolute** (not normalized) coverage.

TABLE VI
RUNTIMES NORMALIZED TO CORRESPONDING ORIGINAL TEST

	Original test	EDDI-V only	QED family test (PLC + EDDI-V)	ERT
FFT	1	≈ 2	≈ 28,672	≈ 28,672
LU	1	≈ 3	≈ 32,768	≈ 32,768
RADIX	1	≈ 3	≈ 30,720	≈ 30,720
OCEAN	1	≈ 4	≈ 33,792	≈ 33,792
Industrial validation test	1	≈ 3	≈ 35	N.A.

the original tests. This raises the question: are the significant coverage (and error detection latency) benefits of QED family tests due to QED check instructions or due to longer runtimes? It may be possible that QED family tests have more opportunities to activate the bug scenarios due to longer runtimes, and result in improved coverage and error detection latency. 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 QED family tests. This is done by keeping most of the QED family tests intact but removing the error reporting code for when an EDDI-V or PLC check detects an error. These tests are referred to as equivalent runtime tests (ERT tests). The error detection latency and coverage of ERT tests are shown in Figs. 14 and 15. ERT test was not created for the industrial validation 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 QED technique shorten error detection latencies by several orders of magnitude for all bug scenarios in Section II. Error detection latencies of original tests can be extremely long: several millions or billions of cycles. The EDDI-V only tests that target bugs inside processor cores have long error detection latencies for uncore bugs. In contrast, QED family tests with PLC and EDDI-V have error detection latencies of only a few hundred cycles for most bug scenarios. Comparison against the ERT results indicates that these benefits come from the QED operations and not from the longer runtimes of QED family tests.

Observation 2: Post-silicon validation tests created using our QED technique continue to detect bug scenarios detected by the original tests. There is not a single bug scenario that the original tests (or the EDDI-V only tests or the ERT tests) detected but the QED family tests with EDDI-V and PLC did not.

Observation 3: QED family tests with PLC and EDDI-V detect up to twofold more bug scenarios that would otherwise remain undetected by the original tests, the EDDI-V only tests, or the ERT tests. Our experiments confirmed that each bug scenario was activated at least once by the original tests, the QED family tests, the EDDI-V only test, and the ERT tests.

VII. INTEL® CORE$^{\text{TM}}$ I7 HARDWARE RESULTS

Fig. 16 shows a quad-core Intel® Core$^{\text{TM}}$ i7 hardware platform used for the evaluation of QED. The BIOS of the DX58SO motherboard is used to vary the operating voltage and frequency of the processor. A custom-designed temperature controller is used to keep the chip package at a fixed temperature. A debug tool attached to the system's debug port is used to control and observe system states (e.g., register and memory contents, operating voltage, and frequency values).

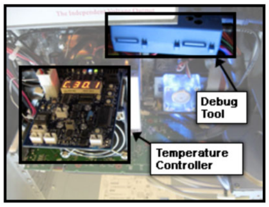


Fig. 16. Quad-core Intel® Core$^{\text{TM}}$ i7 system with a DX58SO motherboard, a temperature controller, and a debug tool.

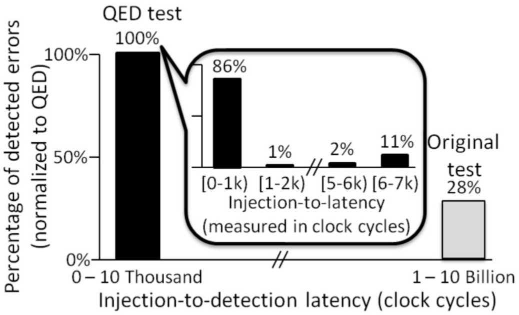


Fig. 17. Histogram showing the distribution of measured injection-to-detection latencies for the Linpack test, which consists of 75 injection-to-detection latencies obtained from vulnerability window injections that resulted in errors detected by but not crashes. For confidentiality reasons, the percentage of detected errors are normalized to QED.

A. Error Detection Latency Results

We performed error injection experiments to measure the error detection latencies of QED tests. The difficulty in measuring error detection latencies in a hardware platform lies in not being able to identify the exact point in time when an error occurs. To overcome this challenge, we create a vulnerability window during which conditions are set so that errors may occur. The start of this window is a lower bound on when an error (if any) actually occurs, which allows us to obtain the injection-to-detection latency, the time between the start of the vulnerability window and error detection. Injection-to-detection latency is an upper bound (i.e., pessimistic) for error detection latency. A vulnerability window is created by using the debug tool to temporarily switch the system from a condition in which the system runs without error (i.e., a reliable operating condition), to a condition in which errors may occur (i.e., an unreliable operating condition), and back. Any error must have occurred during the window of vulnerability, which lasts for no more than a few hundred million cycles.

We identified conditions under which the system would operate with and without errors by sweeping frequency, voltage, and temperature values. In our system, the reliable and unreliable operating conditions (voltage, frequency) pair were chosen to be (1.02 V, 1.6 GHz) and (1.02 V, 3.2 GHz), respectively, and the package temperature was fixed at 30°C. The reliable operating condition was chosen with large frequency margins to ensure that the system operates without error, while the frequency of the unreliable operating condition was chosen to be only slightly faster than the frequency that the processor can reliably support at 1.02 V. By fixing the voltage at 1.02 V and reducing the frequency by a single step of 133 MHz below 3.2 GHz, no more than two QED checks detected error(s) during each of 10 two-hour test runs (using the Linpack test described below). Moreover, at 1.02 V and two steps (i.e., 266 MHz) below 3.2 GHz, no errors were detected for the entire duration of 10 two-hour test runs.

We obtained 75 injection-to-detection latency values; the distributions are shown in Fig. 17. In Fig. 17, the vertical axis represents the percentage of detected errors. This is normalized to QED because of confidentiality reasons. The horizontal axis represents the error detection latencies. The validation test used in this experiment is the Linpack benchmark [19]. The Linpack test used in our experiment executes a main loop for two hours, and each main loop iteration performs the same operations. We transformed the original Linpack program into a QED test by performing the EDDI-V transformation at the source code level with *Inst_min* = *Inst_max* = 10 lines of source code.

In Fig. 17, the results of the QED test are when we take into account the QED checks. Results of the original test shown in Fig. 17 were obtained by ignoring the QED checks and only taking into account the original test's end result checks. This allows us to compare injection-to-detection latencies obtained by using end result checks only, and injection-to-detection latencies obtained by using QED checks, with respect to the same error(s).

With QED tests, injection-to-detection latencies are all very short, ranging from fewer than 1,000 clock cycles to $\sim$ 6,000 clock cycles, as shown in Fig. 17 (actual error detection latencies are even shorter because injection-to-detection latency is only an upper bound). For QED tests, 86% of the injection-to-detection latencies are fewer than 1,000 clock cycles. 24% of the injection-to-detection latencies of QED tests are longer than 1,000 because we performed QED transformation only on the C source code of the tests. We did not perform QED transformation on any system library functions (e.g., functions for performing memory allocation, printing, reading from files, and writing to files). As a result, errors injected when the system was executing operating system library functions were not detected by QED until the system finished executing the operating system library functions and returned to executing the test itself (which is transformed by QED).

For the original test, 72% of the same 75 data points did not result in an error in the final program output (when compared to pregenerated golden results), indicating masked errors. Note that, we did not observe any case where end result checks detected an error but QED checks did not. For the remaining 28%, although incorrect program results were detected by end result checks, injection-to-detection latencies were on the order of billions of clock cycles (even after we subtracted the latency overhead introduced by QED instrumentation, including both the duplicated and check statements).

The C source code-level EDDI-V QED transformation used for the experiments in Figs. 17 and 18 does result in longer test execution times (approximately twofold). However, the overall debug time, and hence productivity, can improve drastically due to significantly shorter error detection latencies.

The following observations can be made from these results.

Observation 4: QED significantly reduces error detection latencies by six orders of magnitude compared to the original test with end result checks. With QED, error detection latencies are reduced from billions of clock cycles to a few thousand clock cycles or less. For 86% of the cases, the error detection latencies of QED tests are less than 1,000 clock cycles. The remaining error detection latencies are longer than 1,000 clock cycles because we performed the QED transformation only at the C source code-level, and not inside any library functions. Errors that occur during the execution of a library function are not detected by QED instrumentation at the C source code-level until that library function is exited.

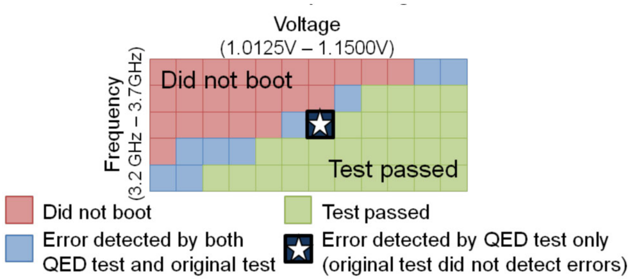


Fig. 18. Linpack Shmoo plot. The voltage and frequency operating point labeled with a star represents unique error detection by the QED test: the original test did not detect any errors, whereas the QED test detected errors quickly.

Observation 5: QED detects errors that would otherwise remain undetected by the original test. QED results in significant (fourfold) improvement in coverage.

Observation 6: QED detects all errors detected by the original test; the incorporation of QED transformation does not adversely impact the ability of the test to detect errors.

B. Electrical Bug Coverage Results

Shmoo plots for the original Linpack (with end result checks) and the EDDI-V-based QED Linpack test are presented in Fig. 18. Each frequency and voltage operating point is classified as follows.

1) Did not boot—the system could not boot.
2) Test passed—both the original test and the QED test did not detect an error.
3) Error detected by both the QED test and original test—an error was detected by a check (a QED check or an end result check), or a system crash occurred.
4) Error detected by the QED test only—the QED test detected an error. Whereas the original test did not detect an error.

From the Shmoo, we make the following observations.

Observation 7: QED improves coverage while significantly improving error detection latency. This is demonstrated by the voltage and frequency operating point in Fig. 18 (labeled with a star) where the original test passed, but the QED test detected errors very quickly. QED tests are valid tests, i.e., they do not bring the system into illegal states. Therefore, the errors detected by QED tests are actual errors in the system. Moreover, there exists no point on the Shmoo plot in Fig. 18 for which the QED test passed but the original test did not. This empirically establishes the fact that the QED test continues to activate and detect errors that are detected by the original test.

VIII. Related Work

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

A. Automatic Post-Silicon Validation Stimuli Generation

In Section III, we demonstrated that self-checking tests [4], [52], [63] can incur very long error detection latencies, as well as introduce excessive intrusiveness, especially for bugs inside uncore components. In contrast, QED shortens error detection latencies for bugs inside processor cores as well as bugs inside uncore components. QED also provides systematic ways to adjust the intrusiveness.

Furthermore, QED can be applied to automatically generated functional tests [8], [10], [31], [34], [50], [55] to improve the error detection latencies and coverage of such tests. QED can also improve the error detection latencies of tests that rely on multipass checking, where tests are run multiple times to determine the expected results and to detect errors.

B. Post-Silicon Debug Techniques

Many existing post-silicon debug techniques can benefit from QEDs short error detection latencies and improved coverage. These include post-silicon debug techniques that rely on failure reproduction [65], simulation [13], [35], trace buffers [2], [49] formal methods [16], [27], [66], or emulator/accelerator-based debugging [32], [51]. QED is also compatible with techniques for enhancing the observability of on-chip signals during post-silicon validation [2], [6], [18], [43], [58]. By significantly reducing error detection latency and improving coverage, QED reduces the number of clock cycles that signals need to be captured and analyzed for debugging.

C. Transformations for Fault-Tolerant Computing

As discussed in Section IV, there are important differences between transformations for post-silicon validation and transformations for fault-tolerant computing.

1) Transformations for post-silicon validation must not introduce excessive intrusiveness, which can degrade the coverage of post-silicon validation tests.
2) During post-silicon validation, the test program inputs may be known *a priori*, this enables special opportunities for QED transformations to improve both error detection latencies and coverage (e.g., Section IV-D) while reducing test runtime overhead (e.g. Section IV-B3).
3) During post-silicon validation, reducing error detection latency is very important because debug time, rather than test execution time, dominates the overall costs of post-silicon validation [30]. Some test runtime overhead can be tolerated if error detection latencies are significantly improved.

D. Memory Scrubbing for Fault-Tolerant Computing

Memory scrubbing [1], [56] is used in fault-tolerant computing to detect and correct errors inside memory arrays. PLC is inspired by memory scrubbing. However, in addition to the differences between transformations for post-silicon validation and transformations for fault-tolerant computing discussed above, PLC and memory scrubbing are different because of the following.

1) Scrubbing 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 during post-silicon validation; therefore, memory scrubbing can result in very long error detection latencies.

E. Assertions for Post-Silicon Validation

The use of assertions during post-silicon validation suffers from several challenges. A design can have numerous assertions; those assertions have to be carefully crafted, and it is difficult to keep them up-to-date and to validate their correctness [9], [60]. While there exist techniques to automatically generating assertions for a given design [20], [24], [60], it may be difficult to implement all of such assertions in hardware. Reconfigurable logic can ease the implementation of assertions in hardware [2], [11], [21], but one has to be careful about selecting the relevant set of assertions to implement in hardware for effective post-silicon debug [42]. Furthermore, assertions may depend on signals located in different regions of an IC, such assertions must be decomposed into components that only use nearby signals. In contrast, QED provides a systematic way of performing extensive checks. The checks inserted by QED can be automatically generated and validated.

IX. Conclusion

QED is a structured approach to post-silicon validation. It overcomes major post-silicon validation challenges by systematically creating post-silicon validation tests that detect bugs very quickly, i.e., with very short error detection latencies and also with improved coverage. Results using a state-of-the-art commercial multicore SoC hardware platform demonstrate nine orders of magnitude improvement in error detection latency. On the same hardware platform, we also demonstrated the ability of QED family tests to systematically adjust tradeoffs between error detection latencies and intrusiveness. Results using an Intel® Core$^{\text{TM}}$ i7 hardware platform demonstrate six orders of magnitude improvement in error detection latencies and simultaneously fourfold improvement in coverage for electrical bugs. Simulation results using a list of difficult bug scenarios on an OpenSPARC T2-like SoC also demonstrate several orders of magnitude improvement in error detection latencies and up to twofold improvement in coverage. Such short error detection latencies can significantly improve post-silicon validation productivity. QED does not require any hardware changes and is readily applicable to existing designs.

Several opportunities exist to further enhance existing validation methodologies using QED. Examples include the following.

1) Automated bug localization/root-causing by analyzing which QED checks passed and which failed.
2) Systematic techniques for synthesizing hardware support for QED to further reduce intrusiveness and improve error detection latency.

3) Effective debugging techniques using QED in emulation environments.
4) Use of QED for system-level identification of failing ICs and for root-causing no-trouble-found (NTF) ICs [14].
5) QED techniques for analog and mixed-signal design blocks.

Acknowledgment

The authors would like to thank E. Rentschler of AMD, W. Kadry, R. Morad, A. Nahir, and A. Ziv of IBM, J. Keshava, R. Mohammed, and K. Tiruvallur of Intel, and M. Mizuno, K. Nose, and K. Yamaguchi of Renesas for their valuable inputs. They also sincerely thank all of the reviewers for their valuable comments.

References

[1] J. A. Abraham, E. S. Davidson, and J. H. Patel, "Memory system design for tolerating single event upsets," *IEEE Trans. Nucl. Sci.*, vol. 30, no. 6, pp. 4339–4344, Dec. 1983.

[2] M. Abramovici, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proc. IEEE/ACM Des. Autom. Conf.*, San Francisco, CA, USA, 2006, pp. 7–12.

[3] A. Adir *et al.*, "A unified methodology for pre-silicon verification and post-silicon validation," in *Proc. IEEE/ACM Des. Autom. Test Eur. Conf.*, Grenoble, France, 2011, pp. 1–6.

[4] A. Aharon *et al.*, "Test program generation for functional verification of PowerPC processors in IBM," in *Proc. IEEE/ACM Des. Autom. Conf.*, San Francisco, CA, USA, 1995, pp. 279–285.

[5] M. E. Amyeen, S. Venkataraman, and M. W. Mak, "Microprocessor system failures debug and fault isolation methodology," in *Proc. IEEE Int. Test Conf.*, Austin, TX, USA, 2009, pp. 1–10.

[6] (2014, Jan. 7). *ARM CoreSight* [Online]. Available: http://www.arm.com/products/system-ip/coresight/

[7] A. A. Bayazit and S. Malik, "Complementary use of runtime validation and model checking," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Washington, DC, USA, 2005, pp. 1049–1056.

[8] A. Benso, A. Bosio, S. D. Carlo, G. Di Natale, and P. Prinetto, "March test generation revealed," *IEEE Trans. Comput.*, vol. 57, no. 12, pp. 1704–1713, Dec. 2008.

[9] B. Bentley and R. Gray, "Validating the Intel Pentium 4 processor," *Intel Technol. J.*, vol. 5, no. 1, pp. 1–8, Feb. 2001.

[10] P. Bernardi, E. E. S. Sanchez, M. Schillaci, and G. Squillero, "An effective technique for the automatic generation of diagnosis-oriented programs for processor cores," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 3, pp. 570–574, Feb. 2008.

[11] M. Boule, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proc. IEEE Int. Symp. Quality Electron. Des.*, San Jose, CA, USA, 2007, pp. 613–620.

[12] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144–156, May 1983.

[13] K.-H. Chang, I. L. Markov, and V. Bertacco, "Automated post-silicon debugging and repair," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, San Jose, CA, USA, 2007, pp. 91–98.

[14] Z. Conroy, G. Richmond, X. Gu, and B. Eklow, "A practical perspective on reducing ASIC NTFs," in *Proc. IEEE Int. Test Conf.*, Austin, TX, USA, 2005, pp. 1–7.

[15] K. Constantinides, O. Mutlu, and T. Austin, "Online design bug detection: RTL analysis, flexible mechanisms, and evaluation," in *Proc. IEEE/ACM Int. Symp. Microarchitecture*, Como, Italy, 2008, pp. 282–293.

[16] F. M. De Paula, A. J. Hu, and A. Nahir, "nuTAB-BackSpace: Rewriting to normalize non-determinism in post-silicon debug traces," in *Proc. Int. Conf. Comput. Aided Verif.*, Berkeley, CA, USA, 2012, pp. 513–531.

[17] A. DeOrio, A. Bauserman, and V. Bertacco, "Post-silicon verification for cache coherence," in *Proc. IEEE Int. Conf. Comput. Des.*, Lake Tahoe, CA, USA, 2008, pp. 348–355.

[18] A. DeOrio, I. Wagner, and V. Bertacco, "DACOTA: Post-silicon validation of the memory subsystem in multi-core designs," in *Proc. IEEE Int. Symp. High-Perform. Comput. Arch.*, Raleigh, NC, USA, 2009, pp. 405–416.

[19] J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present and future," *Concurr. Comput. Pract. Exp.*, vol. 15, no. 9, pp. 803–820, Jul. 2003.

[20] M. D. Ernst *et al.*, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, nos. 1–3, pp. 35–45, Dec. 2007.

[21] M. Gao, H.-M. Chang, P. Lisherness, and K.-T. Cheng, "Time-multiplexed online checking: A feasibility study," in *Proc. IEEE Asian Test Symp.*, Sapporo, Japan, 2008, pp. 371–376.

[22] M. Gao, P. Lisherness, and K.-T. Cheng, "Post-silicon bug detection for variation induced electrical bugs," in *Proc. IEEE Asia South Pacific Des. Autom. Conf.*, Yokohama, Japan, 2011, p. 273.

[23] J. Gray, "Why do computers stop and what can be done about it?" Tandem Computer, Cupertino, CA, USA, Tech. Rep. 85.7, PN 87614, Jun. 1985.

[24] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants," in *Proc. IEEE/ACM Des. Autom. Conf.*, Anaheim, CA, USA, 2005, pp. 775–778.

[25] J. L. Hennessy and D. A. Patterson, "Memory hierarchy design," in *Computer Architecture: A Quantitative Approach,* 5th ed., San Mateo, CA, USA: Morgan Kaufmann, 2012, pp. 72–131.

[26] R. C. Ho, C. H. Yan, M. A. Horowitz, and D. L. Dill, "Architecture validation for processors," in *Proc. ACM/IEEE Int. Symp. Comput. Arch.*, Santa Margherita Ligure, Italy, 1995, pp. 404–413.

[27] R. C. Ho *et al.*, "Post-silicon debug using formal verification waypoints," in *Proc. Des. Validation Conf.*, 2009.

[28] T. Hong *et al.*, "QED: Quick error detection tests for effective post-silicon validation," in *Proc. IEEE Int. Test Conf.*, Austin, TX, USA, 2010, pp. 1–10.

[29] (2009). *ITRS* [Online]. Available: http://www.itrs.net/Links/2009ITRS/Home2009.htm

[30] D. Josephson, "The good, the bad, and the ugly of silicon debug," in *Proc. IEEE/ACM Des. Autom. Conf.*, San Francisco, CA, USA, 2006, pp. 3–6.

[31] Y. Katz, M. Rimon, and A. Ziv, "Generating instruction streams using abstract CSP," in *Proc. IEEE/ACM Des. Autom. Test Eur. Conf.*, Dresden, Germany, 2012, pp. 15–20.

[32] J. Keshava, N. Hakim, and C. Prudvi, "Post-silicon validation challenges: How EDA and academia can help," in *Proc. IEEE/ACM Des. Autom. Conf.*, Anaheim, CA, USA, 2010, pp. 3–7.

[33] H. F. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *Proc. IEEE/ACM Des. Autom. Test Eur. Conf.*, Munich, Germany, 2008, pp. 1298–1303.

[34] A. Krstic, W.-C. Lai, K.-T. Cheng, L. Chen, and D. Dey, "Embedded software-based self-test for programmable core-based designs," *IEEE Des. Test Comput.*, vol. 19, no. 4, pp. 18–27, Aug. 2002.

[35] A. Krstic, L.-C. Wang, K.-T. Cheng, and T. M. Mak, "Diagnosis-based post-silicon timing validation using statistical tools and methodologies," in *Proc. IEEE Int. Test Conf.*, 2003, pp. 339–348.

[36] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra, "Quick detection of difficult bugs for effective post-silicon validation," in *Proc. IEEE/ACM Des. Autom. Conf.*, San Francisco, CA, USA, 2012, pp. 561–566.

[37] X. Liu and Q. Xiu, "Trace signal selection for visibility enhancement in post-silicon validation," in *Proc. IEEE/ACM Des. Autom. Test Eur. Conf.*, Nice, France, 2009, pp. 1338–1343.

[38] D. J. Lu, "Watchdog processors and structural integrity checking," *IEEE. Trans. Comput.*, vol. 31, no. 7, pp. 681–685, Jul. 1982.

[39] M. N. Lovellette *et al.*, "Strategies for fault-tolerant space-based computing: Lessons learned from the ARGOS testbed," in *Proc. Aerospace Conf.*, 2002, pp. 5-2109–5-2119.

[40] M. M. K. Martin *et al.*, "Multifacet's general execution-drive multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Comput. Arch. News*, vol. 33, no. 4, pp. 92–99, Nov. 2005.

[41] R. McLaughlin, S. Venkataraman, and C. Lim, "Automated debug of speed path failures using functional tests," in *Proc. IEEE VLSI Test Symp.*, Santa Cruz, CA, USA, 2009, pp. 91–96.

[42] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," in *Proc. IEEE/ACM Des. Autom. Conf.*, Anaheim, CA, USA, 2010, pp. 12–17.

[43] M. H. Neishaburi and Z. Zilic, "Hierarchical embedded logic analyzer for accurate root-cause analysis," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst.*, Vancouver, BC, Canada, 2011, pp. 120–128.

[44] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 63–75, Mar. 2002.

[45] N. Oh, S. Mitra, and E. J. McCluskey, "ED^{4}I: Error detection by diverse data and duplicated instructions," *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 180–199, Feb. 2002.

[46] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.

[47] K. Olukotun, M. Heinrich, and D. Ofelt, "Digital system simulation: Methodologies and examples," in *Proc. IEEE/ACM Des. Autom. Conf.*, San Francisco, CA, USA, 1998, pp. 658–663.

[48] (2014, Jan. 7). *OpenSPARC: World's First Free 64-bit Microprocessor* [Online]. Available: http://www.opensparc.net

[49] S.-B. Park, T. Hong, and S. Mitra, "Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA)," *IEEE. Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1545–1558, Oct. 2009.

[50] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS—A microprocessor functional BIST method," in *Proc. IEEE Int. Test Conf.*, 2002, pp. 590–598.

[51] P. Patra, "On the cusp of a validation wall," *IEEE Des. Test Comput.*, vol. 24, no. 2, pp. 193–196, Mar. 2007.

[52] R. Raina and R. Molyneaux, "Random self-test method applications on PowerPC microprocessor cache," in *Proc. ACM/IEEE Great Lakes Symp. VLSI*, Lafayette, LA, USA, 1998, pp. 222–229.

[53] K. Reick, "Post-silicon debug—DAC workshop on post-silicon debug: Technologies, methodologies, and best-practices," in *Proc. IEEE/ACM Des. Autom. Conf.*, 2012.

[54] J. P. Shen and M. A. Schuette, "On-line self-monitoring using signatured instruction streams," in *Proc. IEEE Int. Test Conf.*, 1983, pp. 275–282.

[55] S. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self-test and design validation," in *Proc. IEEE Int. Test Conf.*, Washington, DC, USA, 1998, pp. 990–999.

[56] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented EDAC protection against SEUs," *IEEE Trans. Rel.*, vol. 49, no. 3, pp. 273–284, Sep. 2000.

[57] E. Singerman, Y. Abarbanel, and S. Baartmans, "Transaction based pre-to-post silicon validation," in *Proc. IEEE/ACM Des. Autom. Conf.*, New York, NY, USA, 2011, pp. 564–568.

[58] (2014, Jan. 7). *Tektronix, Clarus SoC Post-Silicon Validation Solution* [Online]. Available: http://www.tek.com/embedded-instrumentation/

[59] D. Van Campenhout, T. Mudge, and J. P. Hayes, "Collection and analysis of microprocessor design errors," *IEEE Des. Test Comput.*, vol. 17, no. 4, pp. 51–60, Oct.–Dec. 2000.

[60] S. Vasudevan *et al.*, "GoldMine: Automatic assertion generation using data mining and static analysis," in *Proc. IEEE/ACM Des. Autom. Test Eur.*, 2010, pp. 626–629.

[61] M. N. Velev, "Collection of high-level microprocessor bugs from formal verification of pipelined and superscalar designs," in *Proc. IEEE Int. Test Conf.*, 2003, pp. 138–147.

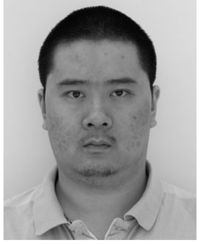
[62] B. Vermeulen and S. K. Goel, "Design for debug: Catching design errors in digital chips," *IEEE Des. Test Comput.*, vol. 19, no. 3, pp. 37–45, May 2002.

[63] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *Proc. IEEE Int. Conf. Comput. Des.*, Lake Tahoe, CA, USA, 2008, pp. 307–314.

[64] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. ACM/IEEE Int. Symp. Comput. Arch.*, Santa Margherita Ligure, Italy, 1995, pp. 24–36.

[65] Y. Yang, N. Nicolici, and A. Veneris, "Automated data analysis solutions to silicon debug," in *Proc. IEEE/ACM Des. Autom. Test Eur.*, Nice, France, 2009, pp. 982–987.

[66] C. S. Zhu, G. Weissenbacher, and S. Malik, "Post-silicon fault localisation using maximum satisfiability and backbones," in *Proc. IEEE/ACM Formal Methods Comp.-Aided Des.*, 2011, pp. 63–66.



David Lin received the B.S. degree in electrical engineering from the California Institute of Technology, Pasadena, CA, USA, in 2009, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2011.

He is a Research Assistant in Prof. Subhasish Mitra's Robust Systems Group at Stanford University. His current research interests include post-silicon validation, verification, debug, and computer architecture.

David Lin is a recipient of the Richard and Naomi Horowitz Stanford Graduate Fellowship.

Ted Hong received the B.S. degree in electrical and computer engineering from the University of California, Berkeley, Berkeley, CA, USA, in 2005, and the M.S. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2007.

He was a Research Assistant under Prof. S. Mitra at the Robust Systems Group, Stanford University. He is currently a Senior Hardware Engineer with Oracle Corporation, Santa Clara, CA, USA. His current research interests include test, post-silicon validation, variability modeling, and circuit simulation and optimization.

Yanjing Li received the B.S. degree in electrical and computer engineering and the M.S. degree in mathematical sciences from Carnegie Mellon University, Pittsburgh, PA, USA, in 2006, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, USA, in 2013.

She has been a Research Scientist at Intel Corporation, Santa Clara, CA, USA, since 2011, and a Visiting Scholar at Stanford University, since 2013. Her current research interests include robust system design, energy-efficient systems, system validation and test, computer architecture, and system software.

Dr. Li was the recipient of the European Design and Automation Association Outstanding Dissertation Award in 2013, the IEEE International Test Conference Best Student Paper Award in 2010, the IEEE VLSI Test Symposium Best Paper Award in 2010, and the two Intel Divisional Recognition Awards in 2012 and 2013.

Eswaran S received the B.E. degree from Madurai Kamaraj University, Madurai, India, in 2000.

He is with Freescale Semiconductor, Noida, India, and has been involved in various roles of post silicon validation for the networking group at the India center. He is currently a Senior Member of the technical staff with the networking division. His current research interests include various hardware and software techniques for silicon validation of large multicore SoCs.

Sharad Kumar received the B.E. degree from Delhi University, Delhi, India, in 1997, and the M.S. degree from Michigan State University, East Lansing, MI, USA, in 2000.

He has been with Freescale Semiconductor, Noida, India (formerly the semiconductor division of Motorola), in various design engineering roles from 2001. Since 2006, he has been involved in driving post silicon validation for the networking group at the India center. He is currently a Senior Manager with the networking division. His current research interests include various hardware and software techniques for silicon validation of large multicore SoCs.

Farzan Fallah received the Ph.D. degree in electrical engineering and computer science from Massachusetts Institute of Technology, Cambridge, MA, USA.

He is the Founder and the President of Idelan Inc. San Jose, CA, USA. He was previously a Scientist at the Department of Electrical Engineering, Stanford University, Stanford, CA, USA. He has published over 70 papers and holds 22 granted or pending patents.

Dr. Fallah was the recipient of the Best Paper Award at the Design Automation Conference in 1998, the Best Paper Award at the VLSI Design Conference in 2005, and 17 Industrial Contribution Awards while with Fujitsu Laboratories of America.

Nagib Hakim received the M.S. and the Ph.D. degrees in electrical engineering from Columbia University, New York, NY, USA, in 1986 and 1992, respectively.

He joined Intel Corporation, Santa Clara, CA, USA. His work in CAD for process development and design includes statistical circuit modeling and optimization, SER prediction, and power/performance analysis. He is currently a Principal Engineer with the System Validation Enabling Division, focusing on methodologies to accelerate post-silicon electrical validation. His current research interests include accurate large-scale circuit modeling and emulation, as well as model to silicon correlation.

Dr. Hakim was the recipient of the Mahboob Khan Outstanding Industry Liaison Award in 2012.

Donald S. Gardner (M'77–SM'08–F'12) received the B.S. degrees in physics and electrical engineering from the University of Southern California, Los Angeles, CA, USA, in 1981, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 1983 and 1988, respectively.

From 1989 to 1991, he was a Research Scientist with Hitachi Research Laboratories, Japan. Since 1991, he has been a Research Engineer and a Principal Engineer at Intel Corporation, Santa Clara, CA, USA. From 2001, he was a Visiting Scholar at Stanford University. He developed a process used to fabricate the world's first microchip with copper-based interconnections and pioneered techniques for the study of metallic films using *in situ* mechanical stress measurements. He also invented an Al alloy/Ti metallization for interconnections that was widely used by industry in microchips. His current research interests include power delivery technology, energy storage devices, magnetic materials, passive devices including inductors and capacitors, new materials integration, and nanostructure design. He holds 86 issued patents, including for inductors using magnetic materials, high-frequency voltage converters, reflowed copper for interconnections, layered aluminum interconnections, and embedded decoupling capacitors. He has authored over 160 articles in journals and conference proceedings.

Dr. Gardner is a member of the Electrochemical Society, American Vacuum Society, and Materials Research Society. He was the recipient of the Intel's highest technical award "For Fundamentally Changing Platform Power Delivery with Integrated Voltage Regulators and Magnetic Inductors on CMOS" and an award for the "World's First Integrated Voltage Regulator," which resulted in significant energy savings in computers, the Intel Division Recognition Award in 2010 for a new energy storage device, and five Best Paper and Poster Awards at international conferences including from the IEEE International Test Conference and IEEE VLSI Test Symposium. He had been a Tutorial Speaker and a member of a panel of experts of the IEEE Magnetics Conference and IEEE VLSI Interconnection conference. He is a member of the Technical Advisory Committee of the IEEE Magnetics Society and has been a Symposium Organizer, Editor, and a Program Chair of the IEEE International Magnetic Conference, MMM Conference, MRS, E-MRS, and IEEE V-MIC.

Professor Subhasish Mitra directs the Robust Systems Group in the Department of Electrical Engineering and the Department of Computer Science of Stanford University, where he is the Chambers Faculty Scholar of Engineering. Before joining Stanford, he was a Principal Engineer at Intel Corporation.

Prof. Mitra's research interests include robust system design, VLSI design, CAD, validation and test, and emerging nanotechnologies. His X-Compact technique for test compression has been key to cost-effective manufacturing and high-quality testing of a vast majority of electronic systems, including numerous Intel products. X-Compact and its derivatives have been implemented in widely-used commercial Electronic Design Automation tools. His work on carbon nanotube imperfection-immune digital VLSI, jointly with his students and collaborators, resulted in the demonstration of the first carbon nanotube computer, and it was featured on the cover of NATURE. The National Science Foundation presented this work as a Research Highlight to the United States Congress, and it also was highlighted as "an important, scientific breakthrough" by the BBC, Economist, EE Times, IEEE Spectrum, MIT Technology Review, National Public Radio, New York Times, Scientific American, Time, Wall Street Journal, Washington Post, and numerous other organizations worldwide.

Prof. Mitra's major honors include the ACM SIGDA/IEEE CEDA A. Richard Newton Technical Impact Award in Electronic Design Automation, which honors an outstanding technical contribution published at least 10 years before the presentation of the award, the Presidential Early Career Award for Scientists and Engineers from the White House, the highest United States honor for early-career outstanding scientists and engineers, and the Intel Achievement Award, Intel's highest corporate honor. He and his students published several award-winning papers at major venues: IEEE/ACM Design Automation Conference, IEEE International Solid-State Circuits Conference, IEEE International Test Conference, IEEE Transactions on CAD, IEEE VLSI Test Symposium, Intel Design and Test Technology Conference, and the Symposium on VLSI Technology. At Stanford, he has been honored several times by graduating seniors "for being important to them during their time at Stanford."

Prof. Mitra has served on numerous conference committees and journal editorial boards. Recently, he served on the Defense Advanced Research Projects Agency's (DARPA) Information Science and Technology Board as an invited member. He is a Fellow of the IEEE.