QED Post-Silicon Validation and Debug

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

HAI LIN

AUGUST 2015

This dissertation is online at: http://purl.stanford.edu/kq850gw1530

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Subhasish Mitra, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Gill, III

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

ABSTRACT

During post-silicon validation and debug, manufactured integrated circuits (ICs) are tested in actual system environments to detect and fix design flaws (bugs). Traditional pre-silicon verification is inadequate; as a result, many critical bugs are detected only after ICs are manufactured (i.e., during post-silicon validation and debug). However, post-silicon validation and debug is challenging because traditional techniques are *ad hoc* (e.g., insertion of various Design for Debug structures based on various heuristics), and the associated costs are rising faster than design costs. These challenges are further magnified by the slowdown of silicon CMOS scaling, as ICs incorporate tremendous complexity to meet increasing demands for improvements in performance and energy efficiency. Examples include the use of multiple processor cores, co-processors, hardware accelerators, uncore components (defined as components in an SoC that are neither the processor cores nor the co-processors / accelerators; examples of uncore components include cache controllers, memory controllers, and interconnection networks), and power management units.

This dissertation presents the Quick Error Detection (QED) technique to overcome post-silicon validation and debug challenges. QED is essential because long error detection latency, the time elapsed between the occurrence of an error caused by a bug and its manifestation as an observable failure, severely limits the effectiveness of

existing post-silicon validation and debug approaches. Experimental results collected using several state-of-the-art commercial hardware platforms, as well as results obtained from simulations of various bug scenarios that occurred in commercial multi-core System-on-Chips (SoCs), demonstrate the effectiveness and practicality of QED:

1. QED improves error detection latencies by up to 9 orders of magnitude, from billions of clock cycles to very few clock cycles (generally fewer than 1,000 clock cycles for most bug scenarios).

2. QED enables up to 4-fold improvement in bug coverage (i.e., QED detects bugs that may be missed by traditional post-silicon validation approaches).

3. Symbolic Quick Error Detection (Symbolic QED) localizes difficult logic bugs automatically in a few hours (less than 7 hours for most bug scenarios), without requiring any additional hardware. Localizing a bug involves identifying a *bug trace* (defined as a sequence of inputs, e.g., instructions, that activates and detects the bug) and identifying the hardware design block where the bug is (possibly) located. This was demonstrated for an open-source multi-core SoC consisting of 500 millions transistors. In contrast, it might take days or weeks (or even months) of manual work, per bug, when traditional techniques are used.

QED is effective for bugs inside processor cores, co-processors / software-programmable accelerators (which are components in an SoC that can be programmed using software to perform a specific set of functions, examples include graphic processing unit and digital signal processor), non-programmable hardware accelerators

(which are components in a SoC that are designed to perform a pre-defined set of functions, but cannot be programmed using software, examples include accelerators for video or audio compression), and uncore components such as cache controllers, memory controllers, and interconnection networks. QED has been successfully used in industry during post-silicon validation and debug of a commercial multi-core SoC.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my research advisor, Professor Subhasish Mitra, for his support and guidance throughout my graduate career. His dedication to research and his patience with me have guided and helped me immensely throughout my graduate career.

I would also like to sincerely thank my associate dissertation readers, Professor Oyekunle A. Olukotun and Professor John Gill for reading my dissertation and providing invaluable feedback, and to Professor Clark Barrett and Professor Roger T. Howe for agreeing to be on my oral exam committee on such a short notice.

I am also extremely thankful to my academic collaborators for their guidance, support, and thoughtful discussions: Professor Clark Barrett from New York University; Professor Deming Chen and Keith Campbell from the University of Illinois at Urbana-Champaign.

I am grateful to the many industrial collaborators throughout my graduate career, who have provided valuable data, insights, and guidance: Don Gardner, Nagib Hakim, Jagannath Keshava, Rahima Mohammed, Helia Naeimi, and Keshavan Tiruvallur from Intel; Eric Rentschler, Greg Lewis, Roy McCubbin from AMD; Sharad Kumar and Eswaran S from Freescale; Chen-Yong Cher, Rainer Dorsch, Rebecca Gott, Wisam Kadry, Ronny Morad, and Avi Ziv from IBM; and Masayuki Mizuno, Koichi Nose, and Koichi Yamaguchi from Renesas.

I would also like to express immense gratitude for the financial support from the Richard and Naomi Horowitz Fellowship and the Semiconductor Research Corporation.

I would like to thank my friends and colleagues from my research group, the Robust Systems Group, specifically: Eric Cheng, Hyungmin Cho, Farzan Fallah, Ted Hong, and Yanjing Li, with whom I shared a significant portion of my graduate career.

Finally, I would like to thank my family for their continued support.

This page was intentionally left blank.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

There is an explosive growth in our dependence on electronic systems. Electronic systems are widely used in banking, commerce, finance, health care, education, scientific research, communication, and entertainment. Unfortunately, malfunctions in electronic systems may have serious consequences, such as loss of productivity, loss of financial gains, or even loss of human lives.

One major source of malfunctions in electronic systems is due to 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 issues (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) [Patra 07].

Traditional pre-silicon verification relies on simulation, emulation or formal methods to find logic bugs. However, traditional pre-silicon verification is too slow, and is often incapable of detecting electrical bugs that appear only after ICs are manufactured

[Patra 07]. As a result, critical design bugs escape pre-silicon verification and are detected only during post-silicon validation and debug [Adir 10, 11, Friedler 14, Foster 07, Keshava 10, Mitra 10, Wisam 13].

1.1 The Post-Silicon Validation and Debug Challenge

During *post-silicon validation and debug*, manufactured integrated circuits (ICs) are tested in actual system environments to detect and fix design bugs. Post-silicon validation and debug is crucial because pre-silicon verification alone is inadequate to detect and debug difficult logic bugs as well as electrical bugs. However, existing post-silicon validation and debug approaches are *ad hoc*. Examples of these *ad hoc* techniques include insertion of various Design-for-Debug (DfD) structures (e.g., [Abramovici 06, Deutsch 14, Vermeulen 02]) based on various heuristics (e.g., [Abramovici 06, Ko 08, Liu 09]). It has been reported that the costs of post-silicon validation and debug are rising [Abramovici 06, Friedler 14, Yerramilli 06]. Furthermore, silicon CMOS technology scaling that has traditionally fueled increases in computing performance and energy efficiency has changed dramatically. While the feature size still scales (Moore's Law), the energy efficiency advantages of (Dennard) scaling are greatly diminished. It is well known that the classical Dennard scaling of silicon CMOS circuits has slowed down [Bohr 09]. This fundamental change in technology scaling is forcing designers to find new ways to achieve improvements in energy-efficiency. This comes at the price of drastically increased complexity. For example, on System-on-Chips (*SoCs*), these complexities include multiple processor cores and co-processors (e.g., Graphics

Processing Unit); various uncore components such on-chip cache controllers, on-chip memory controllers, and on-chip interconnection networks; accelerators such as those for multimedia and cryptography; and adaptive thermal, power, and reliability management features. These increasing levels of complexity further exacerbate the difficulties in validating designs [Adir 11, Mitra 10, Singerman 11]. *Uncore components* (also referred to as nest or northbridge components) are defined as components in an SoC that are neither processor cores nor co-processors. Without scalable ways of taming these increasing levels of complexity, systems can end up being highly vulnerable to design bugs that might jeopardize correct operations and introduce security vulnerabilities [Bose 12]. Such effects are already visible in today's systems [Kubicki 07, Shankland 05, Shimpi 11].

Post-silicon validation and debug involves three activities:

1. Detect the bug by applying proper stimuli. Examples of these stimuli include end-user applications, games, random instruction tests (i.e., random instructions automatically generated by a test generator), and architecture-specific focused tests (i.e., tests specifically written by the designers to test certain features of the design).
2. Localize / root-cause the bug, which involves identifying a sequence of inputs (e.g., instructions) that activates and detects the bug (also referred to as a *bug trace*), and identifying the hardware design block where the bug is (possibly) located.

3. Fix the bug using software patches, circuit editing (e.g., using Focused Ion Beams [Josephson 06]), or silicon re-spin.

Several industrial sources have indicated that the effort to localize bugs from observed system failures (e.g., deadlocks, crashes, wrong / erroneous test results) dominates the overall costs of post-silicon validation and debug [Abramovici 06, Amyeen 09, Friedler 14, Josephson 06, Keshava 10]. For example, it might take days to weeks to localize and debug a single logic bug [Amyeen 09, Keshava 10, Reick 12]. New techniques are essential to reverse this trend.

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 cycles are highly challenging because it is extremely difficult to trace too far back in history for debug [Abramovici 06, Friedler 14, Reick 12], especially for large designs with multiple cores, cache controllers, memory controllers, etc.

Traditional post-silicon validation and debug techniques often rely on trace buffers for generating bug traces. *Trace buffers* are small memory buffers that record the logic values of a selected set of signals. Typically, trace buffers can record only a few (~1,000) clock cycles of history (or a longer history at the cost of recording fewer signals) [Abramovici 06, De Paula 11, Deutsch 14]. However, when dealing with

extremely long error detection latencies (especially for multi-core chips with many signals to record), trace buffer techniques can quickly become ineffective.

Some existing post-silicon validation and debug techniques may also rely on design-specific assertions to detect errors caused by bugs. However, manual assertion creation is difficult, and it is even more difficult to create assertions that can be efficiently implemented in hardware [Abramovici 06, Mitra 10, Vasudevan 10]. While reconfigurable logic can ease the implementation of assertions in hardware [Abramovici 06], it is difficult to select the "right" set of assertions to implement that can quickly detect bugs. This is especially true for automatically-created assertions [El Mandouh 12, Hangal 05, Li 10, Vasudevan 10], which may result in numerous assertions (not all of which can quickly detect bugs), and one must pick which subset to implement in hardware for post-silicon validation and debug.

Many existing post-silicon bug localization practices rely on failure reproduction, which involves returning the system to an error free state and re-executing the failure-causing stimuli. As explained in [De Paula 11, 12], failure reproduction is very difficult for complex ICs due to non-deterministic behaviors such as interrupts, I/O functionalities, interactions between multiple processor cores, and operating system functionalities (e.g., context switches).

The sheer design size also poses major challenges. System-level simulations are several orders of magnitude slower than actual silicon [Adir 11, Keshava 07, Schelle 10]. The use of formal analysis and Boolean Satisfiability techniques for post-silicon

validation and debug (e.g., [De Paula 08, 12, Zhu 11]) can also be severely limited by design size (as will shown in Chapter 6).

1.2 Contributions of This Dissertation

This dissertation presents the Quick Error Detection (*QED*) technique for post-silicon validation and debug. QED **systematically** transforms existing post-silicon validation tests into new QED tests with extremely short error detection latencies and improved coverage. QED tests are effective for both logic and electrical bugs inside processor cores, uncore components, as well as bugs related to power-management features. QED is also effective for non-programmable accelerators in SoCs. Unlike processor cores and programmable co-processors / accelerators (e.g., GPU), non-programmable accelerators implement a pre-defined set of functions and are not programmed using software. The details of QED for non-programmable hardware accelerators are presented in [Campbell 15].

In addition, QED also enables the automatic localization of logic bugs in complex multi-core SoCs.

The main contributions of this dissertation are:

1. Identify that long error detection latency is the main challenge for effective post-silicon validation and debug. Long error detection latency makes it extremely difficult to localize and root-cause bugs.
2. Demonstrate why traditional post-silicon validation techniques incur very long error detection latencies (on the order of millions or even billions of clock cycles) and also have low bug detection coverage.
3. The software-only QED technique for systematically creating post-silicon validation tests with very short error detection latencies and improved bug detection coverage. Results demonstrate that QED achieves up to 9 orders of magnitude improvement in error detection latency and up to 4-fold improvement in bug detection coverage.
4. The Fast QED technique for creating QED tests with small hardware support. Fast QED preserves the improved error detection latency and coverage benefits of software-only QED, while at the same time, significantly improves the runtime of QED tests by up to 4 orders of magnitude. Fast QED makes use of small hardware structures that incurs less than 0.4% area overhead impact, and negligible power and performance impact.
5. The Symbolic QED technique for quickly and automatically localizing logic bugs in complex SoCs. On a complex multi-core SoC with 500-million transistors, results demonstrate that Symbolic QED automatically localizes

logic bugs in only a few hours, compared to days or even weeks of (manual) work required when using traditional post-silicon validation and debug approaches. Such quick localization is possible only with the short error detection latencies and high coverage enabled by QED.

6. A list of difficult bug scenarios abstracted from bug reports of actual bugs found during the post-silicon validation and debug of multiple state-of-the-art commercial multi-core SoCs. Such a list of bug scenarios is important to quantitatively evaluate existing and new validation and debug techniques. These bugs are considered "difficult" because they took very long times to debug (e.g., multiple weeks to multiple months) as indicated in the bug reports.

1.3 Outline of This Dissertation

The rest of this dissertation is organized as follows. Chapter 2 demonstrates how traditional post-silicon validation tests can result in extremely long error detection latencies and low bug coverage. Chapter 3 presents a list of difficult bug scenarios obtained by analyzing the bug reports of several state-of-the-art commercial multi-core SoCs. Such a list of bug scenarios is important to quantitatively evaluate existing and new validation and debug techniques. Chapter 4 presents the Quick Error Detection technique for systematically creating post-silicon validation tests with very short error detection latency and high coverage. While the QED technique can be software-only, which is present in Chapter 4, some hardware support can significantly improve the

benefits of QED, which is demonstrated in Chapter 5 where the Fast QED technique is presented. Enabled by such quick detection, Chapter 6 presents the Symbolic QED technique, which quickly and automatically localizes logic bugs in a design. Appendix 1 provides further discussions on the Symbolic QED technique for localizing logic bugs.

CHAPTER 2. ERROR DETECTION LATENCY CHALLENGE

This chapter presents examples of actual bugs caught during post-silicon validation and debug of commercial SoCs to illustrate long error detection latencies and excessive intrusiveness that may be incurred by traditional post-silicon validation tests.

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 [Adir 11, Keshava 10]. Given the long runtimes of such tests, it is generally difficult to run such tests during the pre-silicon phase. An example is provided in Figure 2.1-Figure 2.6 (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 Figure 2.1-Figure 2.6 is from a real bug found during post-silicon validation of a state-of-the-art commercial multi-core SoC. Consider a multi-core SoC where each processor core can execute 1 instruction per clock cycle, and each processor core has a private L1 cache using write-through policy

[Hennessy 12] (not shown in Figure 2.1-Figure 2.6). The SoC has a shared L2 cache using write-through and write-allocate policy [Hennessy 12]. 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 2 clock cycles. Therefore, as shown in Figure 2.1, 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 Figure 2.1, tests that rely on end result checks, which check for expected output values upon test completion, result in very long error detection latencies.

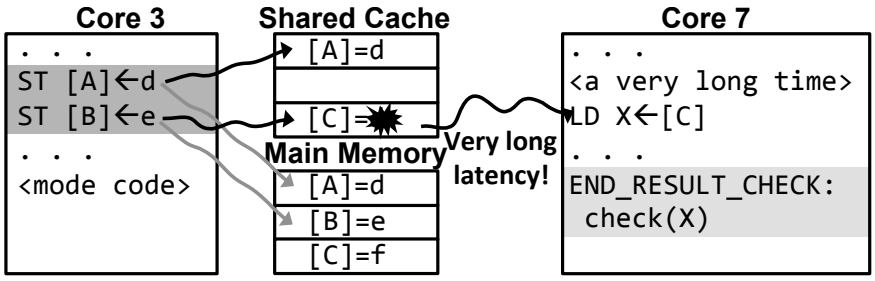


Figure 2.1 An example of end result check test with long error detection latency.

As shown in Figure 2.2, 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 [Chandy 83] 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).

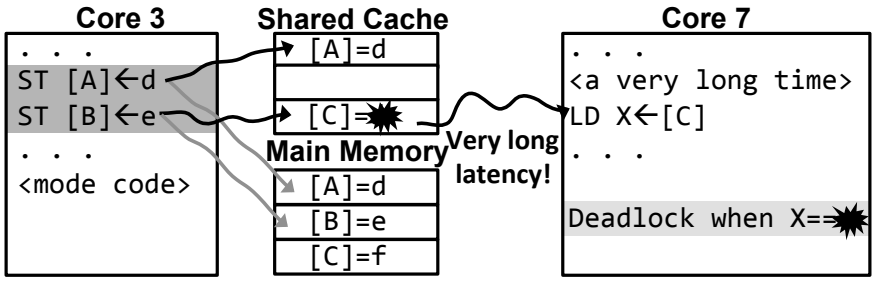


Figure 2.2 An example of deadlock detection test with long error detection latency.

Self-checking tests that focus on bugs inside processor cores are not sufficient to reduce error detection latency for bugs inside uncore components and may also introduce excessive intrusiveness. Here, *intrusiveness* is defined as situations where a bug is no longer activated or detected due to excessive checking operations. Such tests check the results of instructions by inserting self-checking instructions that compare the results against a golden model (i.e., from simulation) [Aharon 95] or against results of other instructions executed on the processor cores [Raina 98, Wagner 08].

Figure 2.3 shows an example where a self-checking test detects the bug with a very long error detection latency. Since 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.

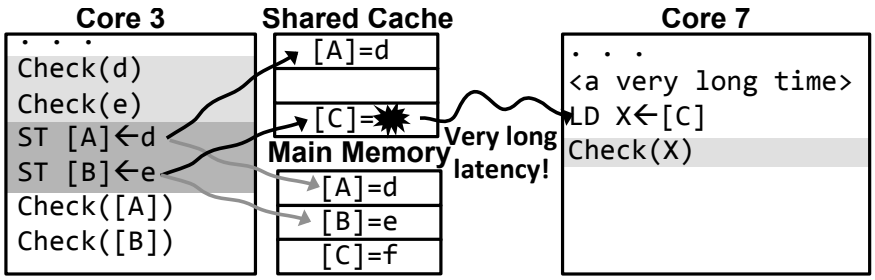


Figure 2.3 An example of a self-checking test with long error detection latency.

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* (Figure 2.4). This disrupts the activation criterion of two stores to adjacent cache lines within 2 clock cycles. Therefore, the bug is not activated and cannot be detected.

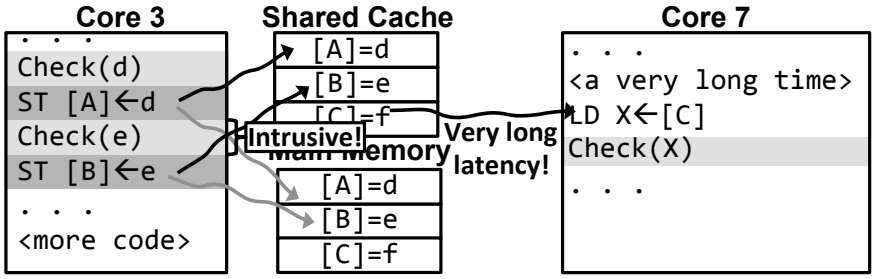


Figure 2.4 An example of self-checking test with excessive intrusiveness.

Another self-checking test variant, referred to as *store readback test* (Figure 2.5), 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 Figure 2.5, 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.

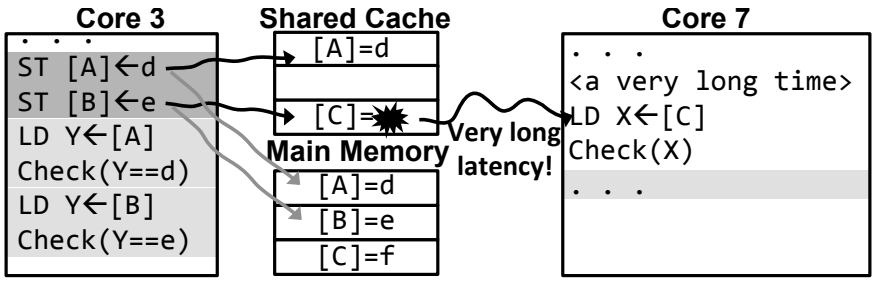


Figure 2.5 An example of store readback test with long error detection latency.

Similar to Figure 2.4, a store readback test can also introduce intrusiveness. For example, as shown in Figure 2.6, 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.

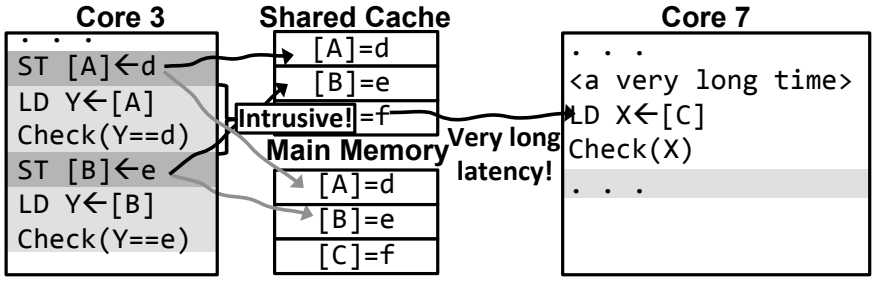


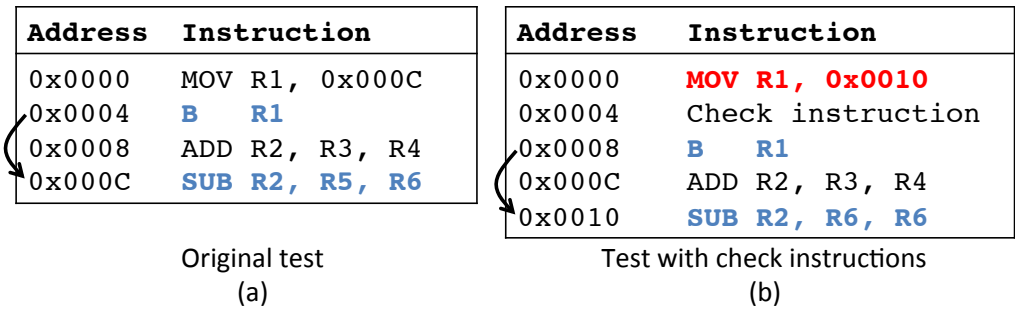
Figure 2.6 An example of store readback test with excessive intrusiveness

In contrast to the above technique, the QED technique (details in Chapters 4-5) 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 vs. error detection latency). Furthermore, QED family tests do not require simulation to obtain expected (i.e., golden) results. Table 2.1 provides a qualitative comparison of traditional post-silicon validation tests vs. QED family tests (details in Chapter 4).

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 of check instructions). An example is shown in Figure 2.7. Figure 2.7a 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 Figure 2.7b, 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.

Simulation is not necessary to obtain the target addresses of direct branch instructions. This is because the target address of direct branch instruction is encoded in the instruction itself and can be readily identified when the 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).



Original test (a)

Test with check instructions (b)

Figure 2.7 Example where simulation may be required to determine the target address of indirect branch instruction.

Table 2.1 Qualitative Comparison of Post-Silicon Validation Test Techniques.

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 [Chandy 83]	Can be very long	No	No	Sometimes*
Self-checking tests [Aharon 95, Raina 98, Wagner 08]	Can be very long	No	Sometimes	Sometimes*
Store readback tests Wagner 08]	Can be very long	No	No	Sometimes*
This dissertation: QED family tests (Chapter 4)	**Short: bounded and configurable (Chapter 4)**	**Yes, systematic (Chapter 4)**	**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.

CHAPTER 3. DIFFICULT BUG SCENARIOS

This chapter presents a list of realistic bug scenarios abstracted from actual bugs found during the post-silicon validation and debug of multiple commercial multi-core SoCs. These bug scenarios can be used to evaluate new post-silicon validation and debug techniques as well as to understand the limitations of existing post-silicon validation and debug techniques. Note that QED, Fast QED, and Symbolic QED techniques presented in this dissertation are not designed to target only the bug scenarios presented in this chapter. These techniques do not require any a priori knowledge about the bug scenarios.

To advance validation and debug 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" that occurred in several state-of-the-art commercial multi-core SoCs. These bugs are considered

"difficult" because they took very long times to debug (e.g., multiple weeks to multiple months) as indicated in the bug reports. 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.

This chapter presents a list of difficult bug scenarios inside both processor cores and uncore components (primarily cache controllers, memory controllers, and interconnection networks) of SoCs. As discussed above, these bugs are abstracted from actual bugs found during the post-silicon validation and debug of several commercial multi-core SoCs. The names and model numbers of these SoCs are not disclosed due to confidentiality reasons. Each bug scenario consists of a bug activation criterion (Table 3.1) and a bug effect (Table 3.2).

Bug activation criterion refers to a set of conditions that must be satisfied to activate a bug scenario. In Table 3.1, criteria 1-4 correspond to bugs inside cache controllers, criterion 5 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 8 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 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., 2 store or load operations in 2 clock cycles; or 2 branch instructions in 2 clock cycles), but there is no maximum value for *X*.

Bug effect refers to the incorrect behavior resulting from bug activation. In Table 3.2, 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 1 clock cycle), but there is no maximum value for *Y*.

Table 3.1 Bug Activation Criteria.

Location	Description
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.

* Where activation criterion is satisfied.
** Include bugs inside cache controllers, memory controllers, and on-chip interconnection networks.

Table 3.2 Bug Effect.

Location	Description
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.

The descriptions in Table 3.1 and Table 3.2 allow one to implement each bug scenario using RTL or micro-architectural simulators. One can create families of bug scenarios by adjusting the parameters *X* and *Y* in Table 3.1 and Table 3.2. For example,

pairing bug activation criterion 2 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.*

3.1 Power Management Related Bug Scenarios

With massive integration of complex SoCs, power management features are becoming increasingly complex and very challenging to validate [Bojan 07, Keshava 10]. The slowdown of silicon CMOS (Dennard) scaling [Bohr 09] implies that increasingly complex power management is required to meet energy efficiency targets. Table 3.3 and Table 3.4 present a list of bug scenarios obtained by analyzing reports of actual "difficult" bugs (primarily logic bugs) in the power management features of commercial multi-core SoCs for servers and mobile applications. The bugs are considered difficult because they took very long times to debug (e.g., several weeks for a single bug). These bug scenarios supplement the bug scenarios in Table 3.1 and Table 3.2.

Each power-management related bug scenario is decomposed into a bug activation criterion (Table 3.3), and a bug effect (Table 3.4). In Table 3.4, bug effects A-C correspond to bugs inside cache controllers, D-F correspond to bugs inside memory controllers, G-I correspond to bugs inside on-chip interconnection networks, and J-L

correspond to bugs inside processor cores. A single bug scenario is formed as an ordered pair of an activation criterion and a bug effect.

Table 3.3 Activation criteria for power management bug scenarios.

ID	Description
1	When exiting from power-saving state.

Table 3.4 Bug effects for power management bug scenarios.

Type	ID	Description
Uncore components	A	The value of the next load operation from data cache is corrupted to all 0.
	B	Next load operation from data cache delayed (1 clock cycle) by cache controller.
	C	Data cache drops the next load operation.
	D	The value of the next load operation from main memory is corrupted to all 0.
	E	Next load operation from main memory delayed (1 clock cycle) by memory controller.
	F	Next load request to main memory is dropped.
	G	Next load operation is delayed for 1 clock cycle by the interconnection network.
	H	Next load operation is corrupted to all 0 by the interconnection network.
	I	Next load operation is dropped by the interconnection network.
Processor cores	J	Processor jumps to a random address.
	K	Next instruction is corrupted to NOP
	L	The value of the next register read is corrupted to all 0.

3.2 Comparison Against Earlier Efforts in Collecting Logic Bugs

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

Table 3.5 summarizes bug scenarios presented in this chapter and earlier efforts to collect logic bugs.

Table 3.5 Summary of logic bug collections.

Bugs	**Source**	**Bug description**	**Components targeted**	**Subsumed by this dissertation?**
This dissertation	Bug reports for industrial SoCs	High level, implementation-independent	Processor core and uncore components	N.A.
[Ho 95]	Research chip	High level, implementation-independent	Processor core	Yes
[Van Campenhout 00]	Class project	Implementation-dependent	Processor core	N.A.
[Velev 03]	Class project	High level, implementation-independent	Processor core	Yes
[DeOrio 08, 09]	Errata pages	High level, implementation-independent	Cache	Yes
[Constantinides 08]	RTL errata	RTL-specific	Processor core	N.A.

CHAPTER 4. QUICK ERROR DETECTION (QED)

This chapter presents the software-only QED technique to overcome challenges in post-silicon validation and debug. Results from multiple hardware platforms, as well as simulation results, demonstrate that QED improves error detection latencies by up to 9 orders of magnitude and improves bug detection coverage by up to 4-folds. The software-only QED technique does not require any hardware modification, and therefore, is readily applicable to existing post-silicon validation and debug flows.

To overcome the challenges in post-silicon validation and debug, we created the Quick Error Detection (QED) technique. Given a post-silicon validation test (referred to as an "original" test in this dissertation) 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 may be determined by the bug localization technique used (e.g., trace buffers or simulation).

For example, for typical on-chip trace buffers that record on the order of 1,000 cycles of history [Abramovici 06, Park 09], error detection latency should be set to less than 1,000 cycles. In addition, if an intrusiveness constraint is available (e.g., if it is known *a priori* that a specific sequence of branch instructions or load / store instructions has a high probability of activating bugs, and thus should not be modified), such constraints can be provided so that the QED transformations do not modify the sequence. One way of providing such constraints is by adding comments next to those instructions to indicate that they have a high probability of activating bugs. An example of this is shown in Figure 4.1. However, if such intrusiveness constraints are not available, the intrusiveness can be systematically adjusted using the QED transformation parameters (details in Sec. 4.4).

```
...
ADD   R3, R3, 1
MUL   R1, R7, R6
SUB   R2, R1, R5
// QED_DO_NOT_MODIFY_BEGIN
STORE R1, 0x3000_0000
STORE R2, 0x3000_0010
STORE R3, 0x3000_0020
LOAD  R1, 0x3000_0030
LOAD  R2, 0x3000_0040
LOAD  R3, 0x3000_0050
CMP   R4, 10
BNE   label
// QED_DO_NOT_MODIFY_END
...
```

A sequence of instructions that **should not** be modified

Figure 4.1 Adding comments next to a sequence of instructions that has a high probability of activating bugs. QED transformations will not insert or modify any instructions in the shaded box.

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 4.1-4.3) and QED transformation parameters (details in Sec. 4.4) for each test in the family. Each of the tests in the family can have different tradeoffs between error detection latency, the SoC components that it targets (i.e., processor cores vs. uncore components), and the amount of intrusiveness introduced. 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. The details of QED family tests are presented in Sec. 4.5).

This chapter presents the software-only QED technique for creating effective post-silicon validation tests. These software-only QED transformations do not require any hardware modifications, and are readily applicable to existing post-silicon validation and debug flows. Although this chapter presents the software-only QED transformations, the resulting QED tests are effective for bugs inside processor cores as well as bugs inside uncore components of SoCs (as demonstrated in Sec. 4.8, Sec. 4.9, and Sec. 4.10). QED also can be augmented with small hardware support (details in Chapter 5). QED is also effective for bugs inside non-programmable hardware accelerators. The details of QED for non-programmable hardware accelerators are presented in [Campbell 15].

Major benefits of the software-only QED include:

1. QED significantly improves error detection latency. Results from multiple hardware platforms, including the Intel® Core ™ i7 SoC, as well as simulation results using realistic bug scenarios (Chapter 3) on a complex

multi-core SoC demonstrate that QED improve error detection latencies by up to 9 orders of magnitude when compare against traditional post-silicon validation tests.

2. QED significantly improves bug coverage. Results from multiple hardware platforms, including the Intel® Core ™ i7 SoC, as well as simulation results using realistic bug scenarios (Chapter 3) on a complex multi-core SoC demonstrate that QED achieves up to 4-fold improvement in coverage when compared against traditional post-silicon validation tests.
3. QED provides a systematic approach to adjust tradeoffs between error detection latency and intrusiveness. This is demonstrated using a state-of-the-art commercial multi-core SoC hardware platform with an actual difficult logic bug.

The rest of this chapter is organized as follows. Sections 4.1, 4.2 and 4.3 in this chapter present four software-only QED transformations: Error Detection by 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). Sec. 4.4 presents details for the QED transformation parameters *Inst_min* and *Inst_max*. Section 4.5 presents the details of QED family test. A case study for an extremely difficult logic bug in a state-of-the-art commercial multi-core SoC hardware platform is presented in Sec. 4.7. Sections 4.9 and 4.10 present simulation results using an OpenSPARC T2-like multi-core SoC and

experimental results using the Intel® Core™ i7 hardware platform. Related work is discussed in Sec. 4.11, followed by summary and discussions in Sec. 4.12.

4.1 Error Detection by Duplicated Instruction for Validation (EDDI-V)

As shown in Chapter 2, post-silicon validation tests can result in extremely long error detection latencies. Furthermore, 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 [Oh 02a]) bounds error detection latency (details in Section 4.4) for bugs that occur inside processor cores, and provides configurability in the amount of intrusiveness introduced (details in Sec. 4.4 and 4.5).

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 dissertation), 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 Figure 4.2a, the original test uses 16 registers (r0 to r15). Therefore, the EDDI-V transformation reserves 16 registers (r16 to r31) and initializes them to the same values as those in r0 to 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

[1] One must be careful about intrusiveness due to additional store and load instructions. The intrusiveness is controlled by the QED transformation parameters *Inst_min* and *Inst_max* explained in Sec. 4.4.

example, in Figure 4.2b, the original test needs to use all 32 registers available in the system (r0 to 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 Figure 4.2b, 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 Figure 4.2, 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 Sec. 4.4).

EDDI-V can be enhanced using diversity techniques (e.g., based on the principles of [Oh 02b]) 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, results in Sec. 4.7 and 4.9 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.

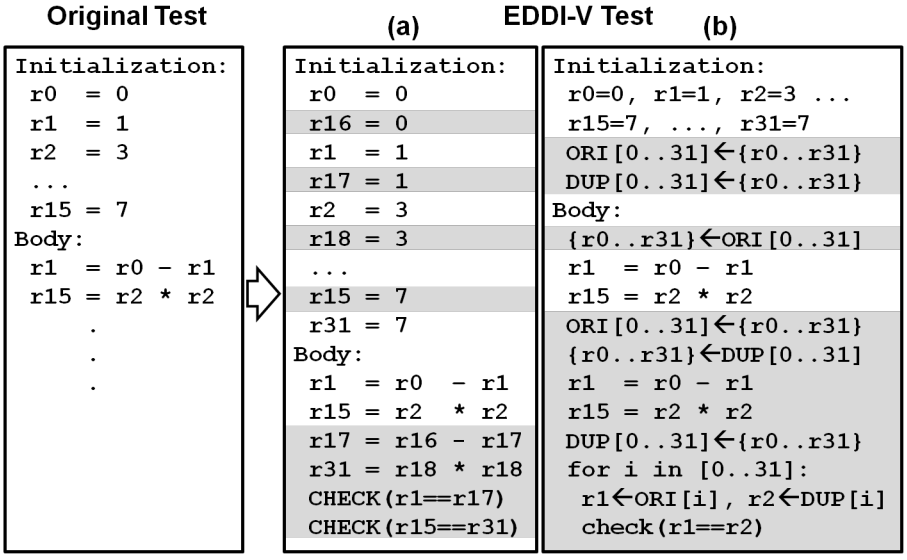


Figure 4.2 Examples of the EDDI-V transformation with *Inst_min*=*Inst_max*=*2*: (a) with half of all registers reserved for EDDI-V, and (b) with register values stored temporarily in memory blocks *ORI* and *DUP*.

4.2 Proactive Load and Check (PLC)

Proactive Load and Check (*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) re-execution 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 operations execute on all threads. This is because the bugs are not known *a priori* and can affect various pathways between processor cores and uncore components. For example, a bug may exist in a point-to-point interconnect that is used only by processor core 2 to read / write from cache bank 1. As a result, the bug may not be detected unless processor core 2 reads / writes to cache bank 1 via the point-to-point interconnect. The PLC transformation is compatible with the EDDI-V transformation; hence, bugs inside processor cores can also be detected with short error detection latencies.

In the next subsections, we provide a detailed overview of the steps required to implement the software-only PLC transformation for a multi-core and multi-threaded SoC, such as the OpenSPARC T2 SoC. However, PLC can be further enhanced with hardware support (details are presented in Chapter 5).

A. 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 Figure 4.3). 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 multi-threaded 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 (Figure 4.3).

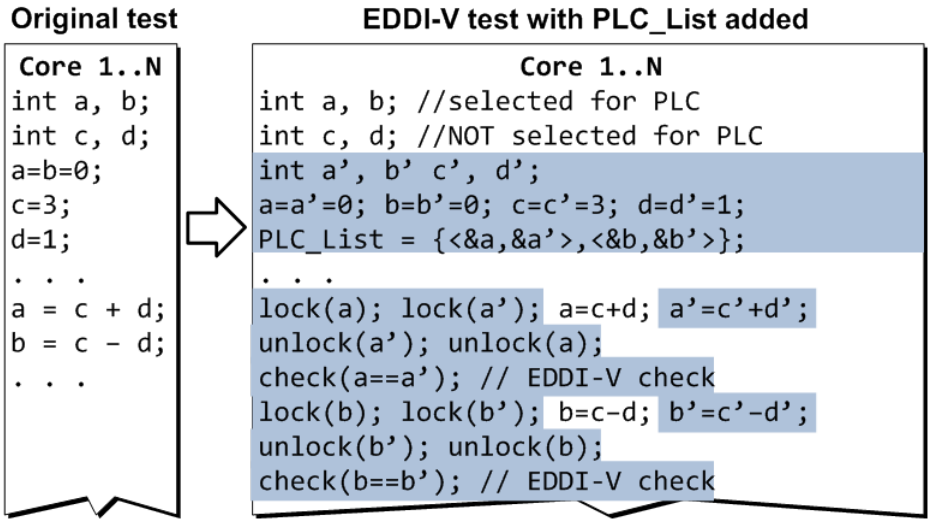


Figure 4.3 PLC transformation, Step 1: initialization.

B. Step 2: PLC Operation Insertion

The PLC transformation inserts Proactive Load and Check operations in each thread running on each processor core. Each *Proactive Load and Check (PLC) operation* (Figure 4.4) 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 Chapter 2). 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).

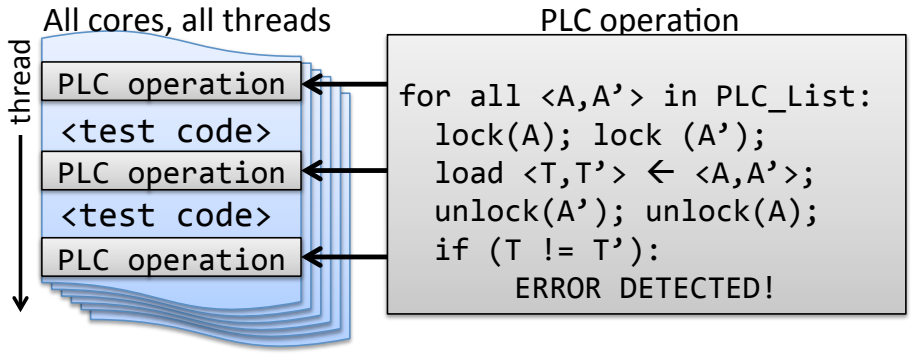


Figure 4.4 PLC transformation, Step 2: PLC operation insertion.

C. 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, results in Section 4.7 and 4.9 empirically demonstrate that the **entire family** of PLC-based *QED family tests* (details in Sec. 4.5) 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 [Bentley 01]), 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.*

4.3 Control Flow Checking Using Software Signatures for Validation (CFCSS-V) and Control Flow Tracking Using Software Signatures for Validation (CFTSS-V)

Control Flow Checking using Software Signatures for Validation (CFCSS-V) and Control Flow Tracking using Software Signatures for Validation (CFTSS-V) are two QED transformations that target bugs that affect a processor core's control flow. CFCSS-V is inspired by CFCSS [Oh 02c] 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 [Oh 02c]. 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 [Oh 02c], 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 Sec. 4.4). 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 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 software signatures are found in [Lu 82, Oh 02c, Shen 83]. Then, the control flow graph between the blocks of instructions is constructed following the algorithm presented in [Oh 02c]. 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 [Oh 02c].

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 `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 Sec. 4.4). 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 [Lu 82, Oh 02c, Shen 83].

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 *SIGNATURE*. Figure 4.5 lists the pseudo assembly code for the CFTSS-V operation. In Figure 4.5, *[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 Figure 4.5, and *[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 *[SIGNATURE]* is used to determine the last "block of instructions" that was executed by the processor core before the failure.

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

Figure 4.5 Pseudo assembly code for the CFTSS-V operation inserted at the beginning of each "block of instructions."

4.4 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.

During post-silicon validation, test program inputs may be known *a priori* [Bentley 01]. 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 Figure 4.6, 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.

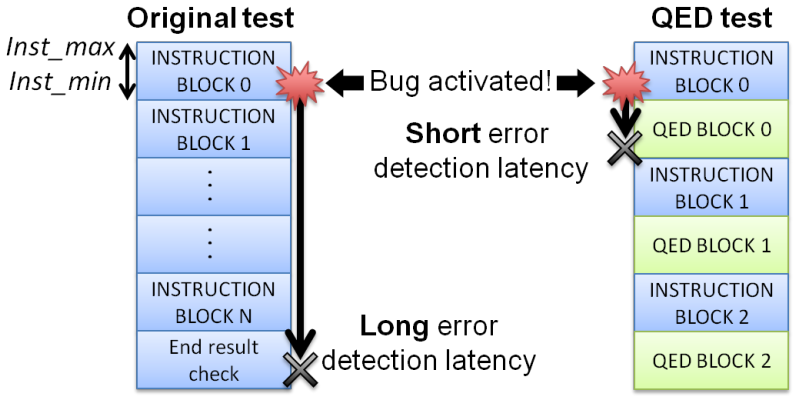


Figure 4.6 Error detection latency of the original test vs. the QED test.

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." We define the *window* as a consecutive sequence of instructions that are grouped together into a "block of instructions." For example, Figure 4.7 illustrates three different windows for *Inst_min* = *Inst_max* = 3. In Figure 4.7 each "block of instructions" for test (a), (b) and (c) contains 3 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.

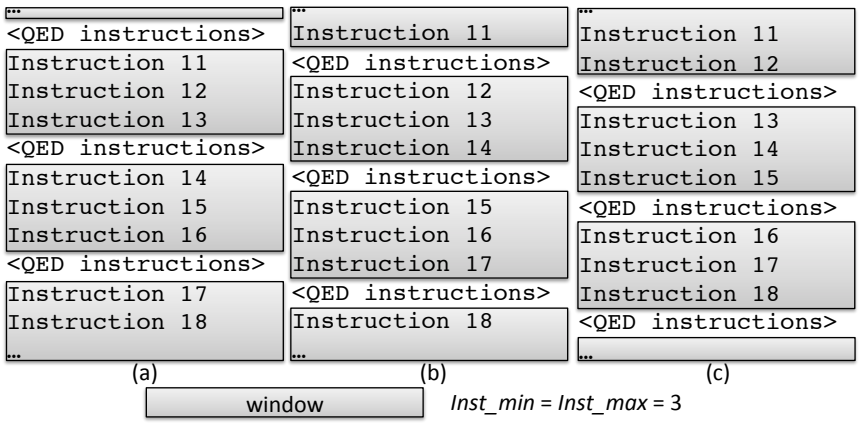


Figure 4.7 An example of three possible "windows" for *Inst_min=Inst_max=3*.

4.5 QED Family Tests

Multiple QED tests can be created from a single original test. The resulting tests are collectively referred to as *QED family tests*. 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 Sec. 4.4 of this chapter) and vary the "window" for QED transformations (details in Sec. 4.4 of this chapter) to create multiple QED family tests. An example demonstrating how QED

family tests are created from a single original test *T* is shown in Figure 2.1. 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.

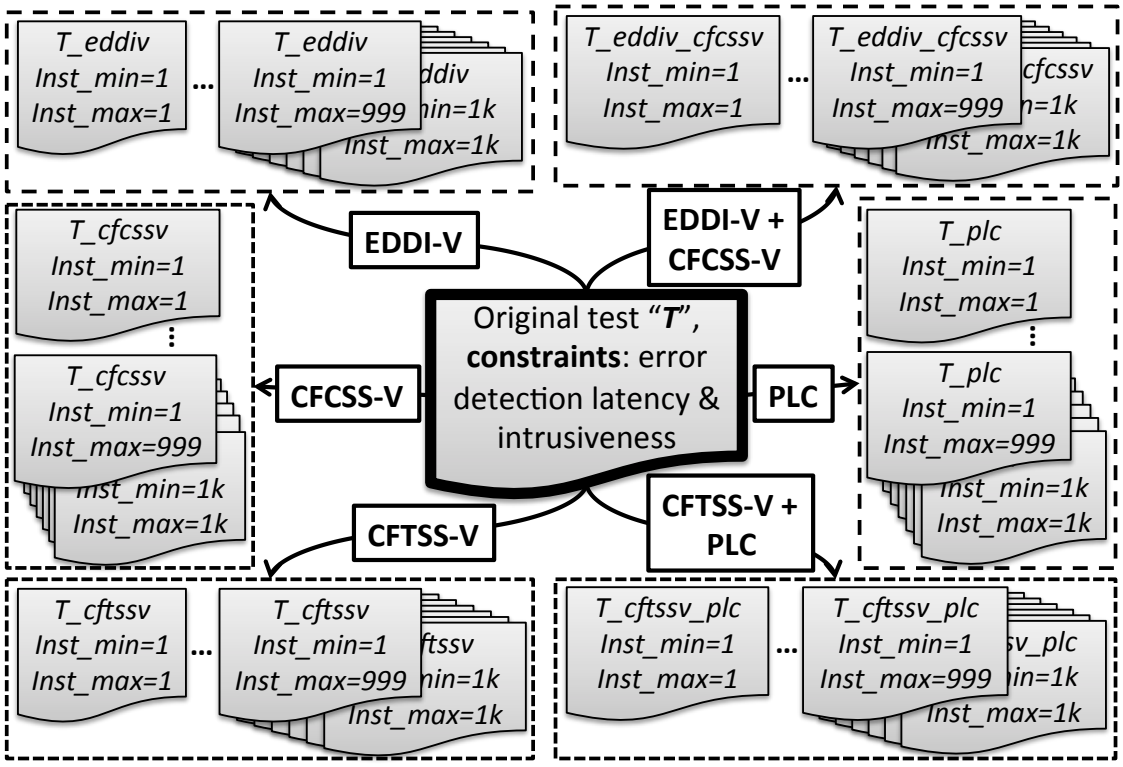


Figure 4.8 A conceptual example of QED family tests. Here, a single original test "*T*" is supplied as the input, along with the constraints on error detection latency and intrusiveness (if available). We select the EDDI-V, CFCSS-V, CFTSS-V, PLC, EDDI-V + CFCSS-V, and CFTSS-V + PLC transformations, along with a range of values for *Inst_min*, from *Inst_min*=1 to *Inst_min*=1,000 (1k), and a range of values for *Inst_max*, from *Inst_max*=1 to *Inst_max*=1,000 (1k) to create multiple tests in the family. We also vary the window for each test in the family.

4.6 Summary and Comparison of Software-Only QED Techniques

The above sections presented four software-only QED transformations. Table 4.1 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.

Table 4.1 Comparison of QED transformations presented in this chapter.

	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).

4.7 Post-Silicon Validation and Debug vs. Fault Tolerant Computing Discussion

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 [Lovellette 02] 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 [Josephson 06]. 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 Chapter 2, 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* [Bentley 01]. This presents an 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 Sec. 4.4).

4.8 Case Study: Logic Bug in a Commercial Multi-Core SoC

In this section, we present a case study to demonstrate the effectiveness of QED in detecting logic bugs in a complex multi-core SoC hardware platform. The SoC contains 6 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. Figure 4.9 shows a picture of the SoC in the validation lab 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 10-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 or any other information about the bug itself (other than the fact that the bug causes a deadlock). 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.

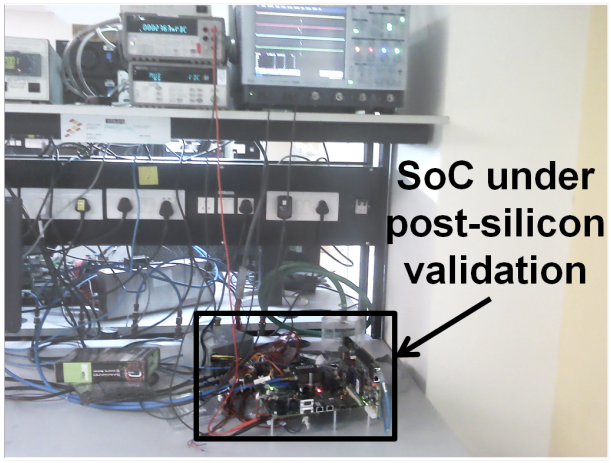


Figure 4.9 Picture of the SoC during post-silicon validation and debug.

A. CFCSS-V and CFTSS-V 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* ≤ 20, we also varied the transformation "windows" (Sec. 4.5) 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* ≤ 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* 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 Figure 4.10, 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 9 instructions after it was activated (corresponding to an error detection latency of approximately 9 clock cycles). This is 9 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.

Figure 4.10 shows the systematic tradeoffs between error detection latency and intrusiveness using *Inst_min*. For this bug, CFTSS-V transformation with *Inst_min* smaller than 9 instructions introduced excessive intrusiveness; therefore the bug is not activated and not detected when *Inst_min* is less than 9 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.

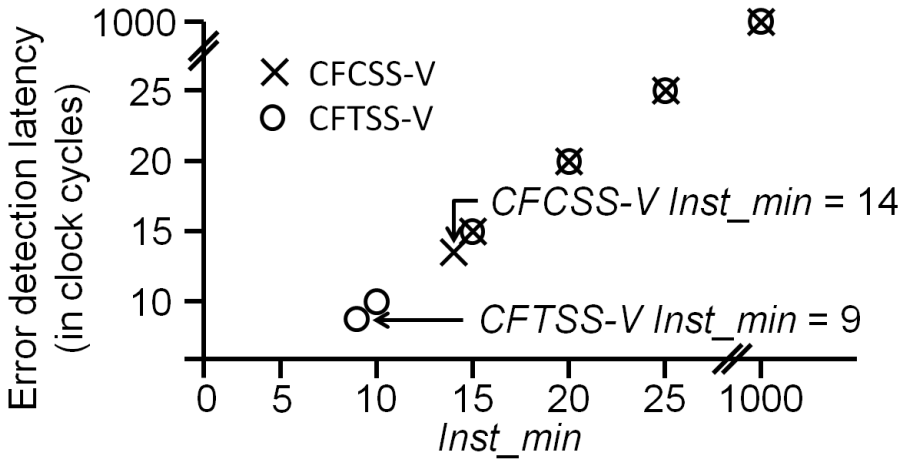


Figure 4.10 Error detection latencies vs. *Inst_min*. CFTSS-V detected the bug in 9 clock cycles and CFCSS-V detected the bug in 14 clock cycles. For the CFTSS-V QED test, the bug is not activated when *Inst_min* ≤ 8. For the CFCSS-V QED test, the bug is not activated when *Inst_min* ≤ 13. For *Inst_min* ≤ 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 4.2 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.

Table 4.2 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

B. EDDI-V Results

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*.

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 Figure 4.11. 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 Figure 4.11, the minimum *Inst_min* value that still preserves the deadlock is 17. This value is greater than the 9 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.

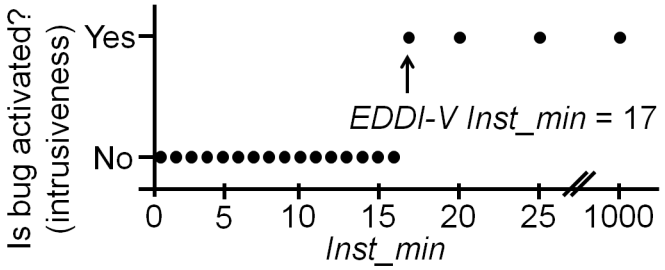


Figure 4.11 Intrusiveness vs. *Inst_min* tradeoff of EDDI-V QED family tests. For each *Inst_min*, the bug is considered as activated if at least one of the tests created using the transformation "window" method for that particular *Inst_min* activated the bug.

4.9 OpenSPARC T2 SoC Simulation Results

To demonstrate the effectiveness of QED for a wide variety of bugs, we evaluated QED by simulating the bug scenarios resulting from Table 3.1 and Table 3.2 using a micro-architectural simulator. We used Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) [Martin 05] to simulate an OpenSPARC T2-like SoC [OpenSPARC], a 500 million-transistor design with 8 processor cores, 64 hardware threads, private L1 data and instruction caches, crossbar-based interconnects, 8-way

banked L2 cache using directory-based cache coherence protocol, and 4 memory controllers (Figure 4.12).

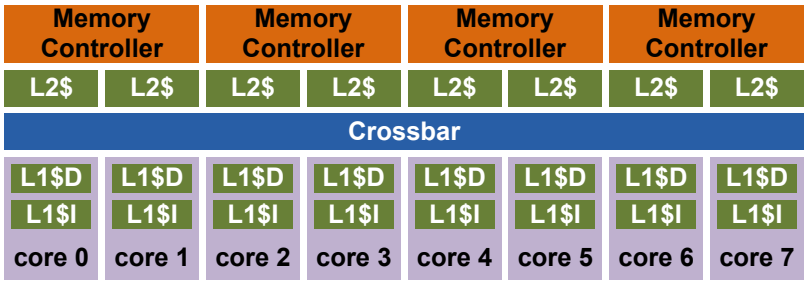


Figure 4.12 Block diagram of an OpenSPARC T2-like multi-core SoC.

Table 3.1 and Table 3.2 presents 80 different bug classes (the cross product of 8 bug activation criteria and 10 bug effects in Table 3.1 and Table 3.2) relating to the processor cores, cache controllers, memory controllers, and interconnection network. 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 2 clock cycles because it is not possible for the simulated processor core to execute more than 2 load, store or branch instructions in one clock cycle [Martin 05]. 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 9 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 10 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 10 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 8 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.

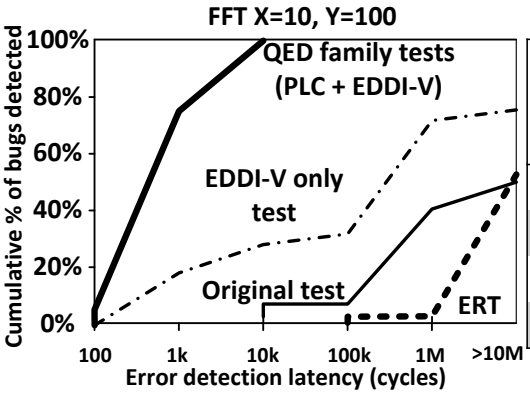
In this section, the bug scenarios correspond to logic bugs, which are always present in the system (i.e., they are not injected). A specific set of activation criteria has to be satisfied in order for a logic bug to cause an effect in the system. Therefore, the bug scenarios were simulated in the following 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 pre-selected).

The results are summarized in Figure 4.13 to Figure 4.22 for FFT, LU, RADIX and OCEAN programs from the SPLASH-2 benchmark suite [Woo 95] 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 presented Chapter 3are 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.

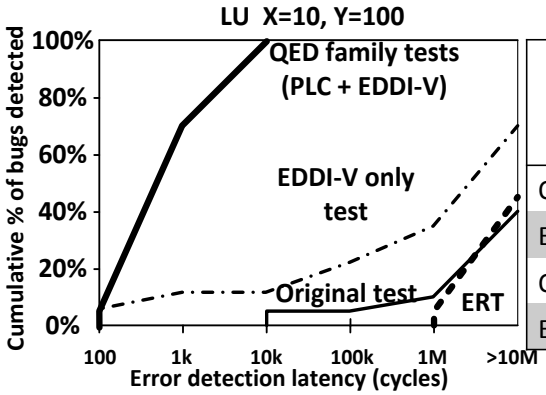
In Figure 4.13 to Figure 4.17, the vertical axes represent the absolute percentage of bug scenarios from Chapter 3 detected (not normalized to number of bugs detected by QED family tests), and the horizontal axes represent the error detection latencies in clock cycles. In Figure 4.13 to Figure 4.17, 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 QED's 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 Figure 4.18 to Figure 4.22. In Figure 4.13 to Figure 4.22, 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).



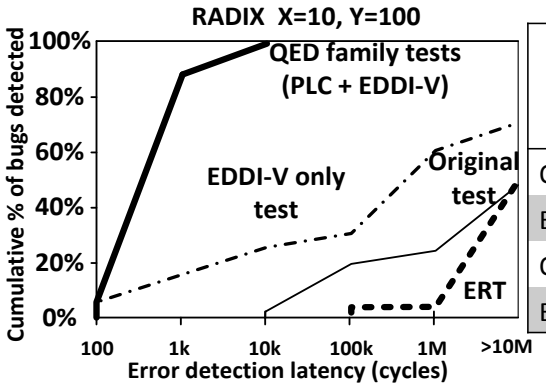
	Error detection latencies (clock cycles)		
	Minimum	Median	Maximum
QED family tests	19	620	2.8k
EDDI-V only test	19	138k	9.9M
Original test	7.5k	602k	9.9M
ERT test	68k	5M	10M

Figure 4.13 Plot showing Error Detection Latencies (EDL) and coverage of the FFT test from SPLASH-2 [Woo 95] with *X=10* and *Y=100* for the bug scenario parameters. The vertical axis represents the absolute cumulative percentage (not normalized percentage) of bug scenarios detected with respect to the list of bug scenarios resulting from Table 3.1 and Table 3.2. The horizontal axis represents the error detection latencies.



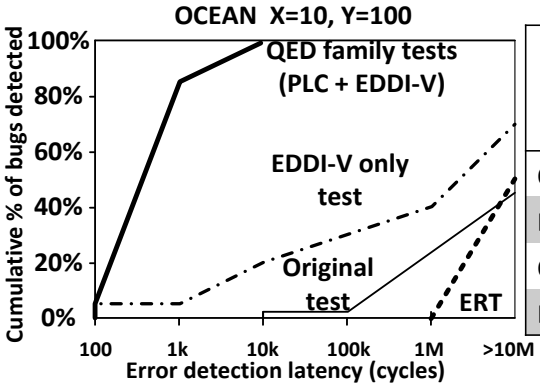
	Error detection latencies (clock cycles)		
	Minimum	Median	Maximum
QED family tests	24	590	2.2k
EDDI-V only test	50	34k	9.9M
Original test	6k	1.8M	10M
ERT test	698k	7.9M	10M

Figure 4.14 Plot showing Error Detection Latencies (EDL) and coverage of the LU test from SPLASH-2 [Woo 95] with *X=10* and *Y=100* for the bug scenario parameters. The vertical axis represents the absolute cumulative percentage (not normalized percentage) of bug scenarios detected with respect to the list of bug scenarios resulting from Table 3.1 and Table 3.2. The horizontal axis represents the error detection latencies.



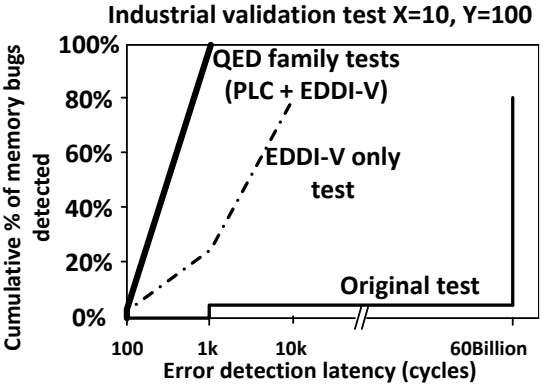
	Error detection latencies (clock cycles)		
	Minimum	Median	Maximum
QED family tests	29	680	5.3k
EDDI-V only test	171	156k	994k
Original test	14k	1.9M	9.9M
ERT test	63k	7.9M	10M

Figure 4.15 Plot showing Error Detection Latencies (EDL) and coverage of the RADIX test from SPLASH-2 [Woo 95] with *X=10* and *Y=100* for the bug scenario parameters. The vertical axis represents the absolute cumulative percentage (not normalized percentage) of bug scenarios detected with respect to the list of bug scenarios resulting from Table 3.1 and Table 3.2. The horizontal axis represents the error detection latencies.



	Error detection latencies (clock cycles)		
	Minimum	Median	Maximum
QED family tests	32	550	5.4k
EDDI-V only test	44	852k	9.9M
Original test	45k	2.6M	9.9M
ERT test	850k	2.1M	10M

Figure 4.16 Plot showing Error Detection Latencies (EDL) and coverage of the OCEAN test from SPLASH-2 [Woo 95] with *X=10* and *Y=100* for the bug scenario parameters. The vertical axis represents the absolute cumulative percentage (not normalized percentage) of bug scenarios detected with respect to the list of bug scenarios resulting from Table 3.1 and Table 3.2. The horizontal axis represents the error detection latencies.



	Error detection latencies (clock cycles)		
	Minimum	Median	Maximum
QED family tests	79	730	990
EDDI-V only test	79	2.2k	5k
Original test	975k	6B	6B

Figure 4.17 Plot showing Error Detection Latencies (EDL) and coverage of the Industrial validation test with *X=10* and *Y=100* for the bug scenario parameters. The vertical axis represents the absolute cumulative percentage (not normalized percentage) of bug scenarios detected with respect to the list of bug scenarios resulting from Table 3.1 and Table 3.2. The horizontal axis represents the error detection latencies.

In Figure 4.18 to Figure 4.22, the vertical axes represent the error detection latencies in clock cycles. The horizontal axes represent the *X* values for the bug

scenarios. In Figure 4.18 to Figure 4.22, 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). Figure 4.18 to Figure 4.22 show the minimum, median (represented by the square dot), and maximum error detection latencies. The results in In Figure 4.13 to Figure 4.22 correspond to 8-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 [Martin 05]. In Figure 4.13 to Figure 4.22 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 Figure 4.13 to Figure 4.22, 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 Figure 4.13 to Figure 4.22 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, Figure 4.17 and Figure 4.22 only considers memory bug scenarios (i.e., activation criteria 1-5 in Table 3.1 and effects A-G in Table 3.2). Figure 4.13 to Figure 4.22 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 4.3 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 4.3, 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 1 test in the family for Industrial validation test). The runtimes of QED family tests are significantly longer than 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 *ERT* tests

(Equivalent RunTime tests). The error detection latency and coverage of ERT tests are shown in Figure 4.13 to Figure 4.16 and Figure 4.18 to Figure 4.21. ERT test was not created for the industrial validation test because the runtime of the original test is already very long.

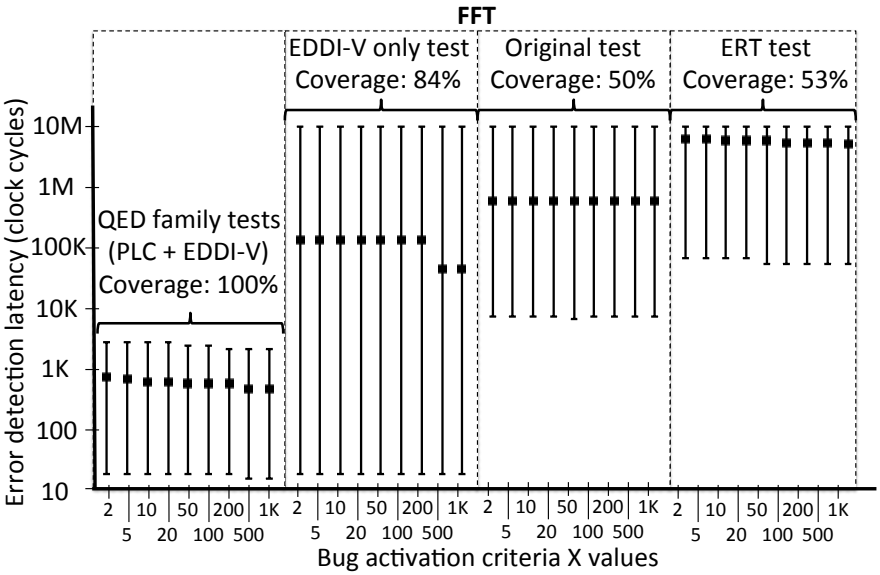


Figure 4.18 Graph showing the minimum, median (represented by the square dot), and maximum error detection latencies by varying both *X* and *Y* parameters of the bug activation criteria and bug effects for FFT from SPLASH-2 Woo 95]. The coverage numbers reported are absolute (not normalized) coverage.

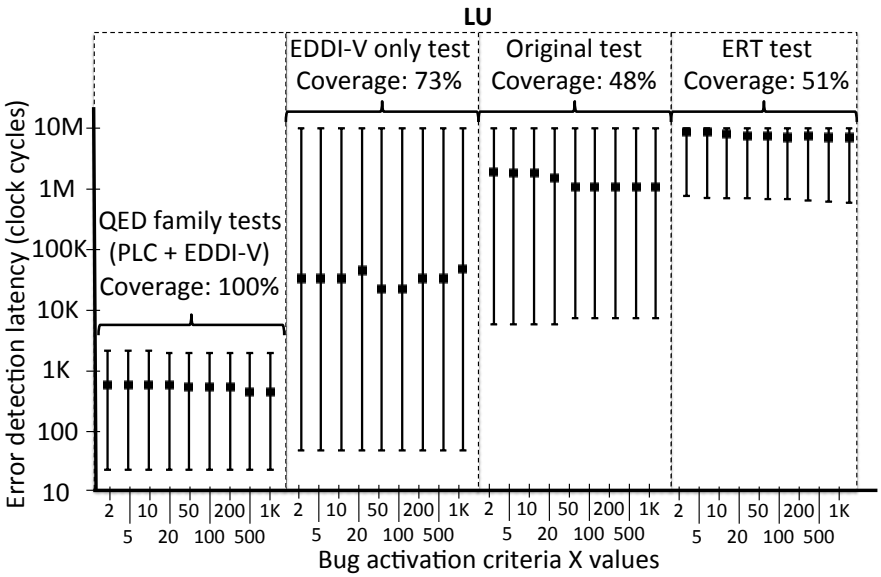


Figure 4.19 Graph showing the minimum, median (represented by the square dot), and maximum error detection latencies by varying both *X* and *Y* parameters of the bug activation criteria and bug effects for LU from SPLASH-2 Woo 95]. The coverage numbers reported are absolute (not normalized) coverage.

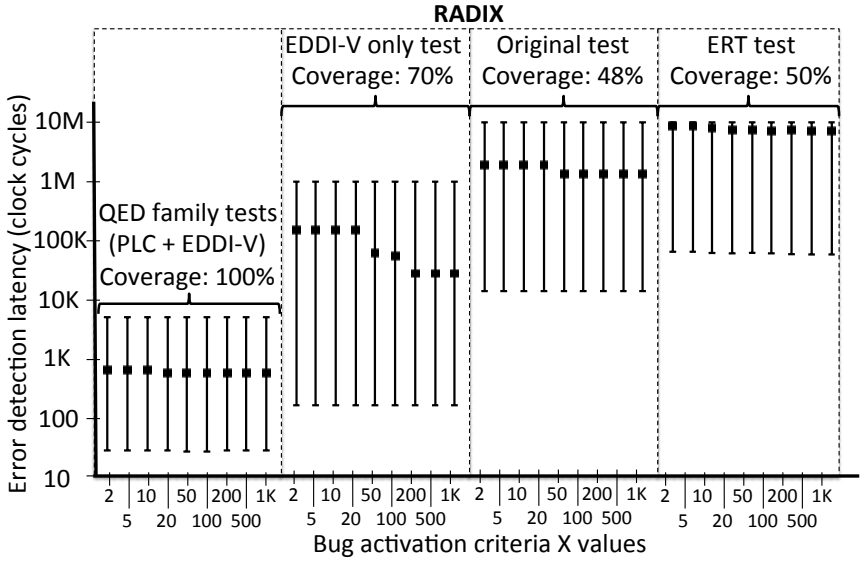


Figure 4.20 Graph showing the minimum, median (represented by the square dot), and maximum error detection latencies by varying both *X* and *Y* parameters of the bug activation criteria and bug effects for RADIX from SPLASH-2 Woo 95]. The coverage numbers reported are absolute (not normalized) coverage.

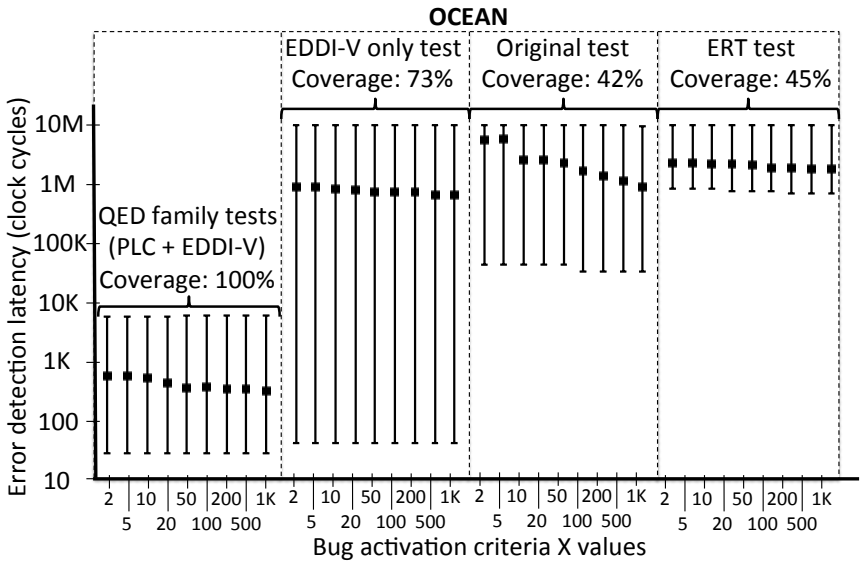


Figure 4.21 Graph showing the minimum, median (represented by the square dot), and maximum error detection latencies by varying both *X* and *Y* parameters of the bug activation criteria and bug effects for OCEAN from SPLASH-2 Woo 95]. The coverage numbers reported are absolute (not normalized) coverage.

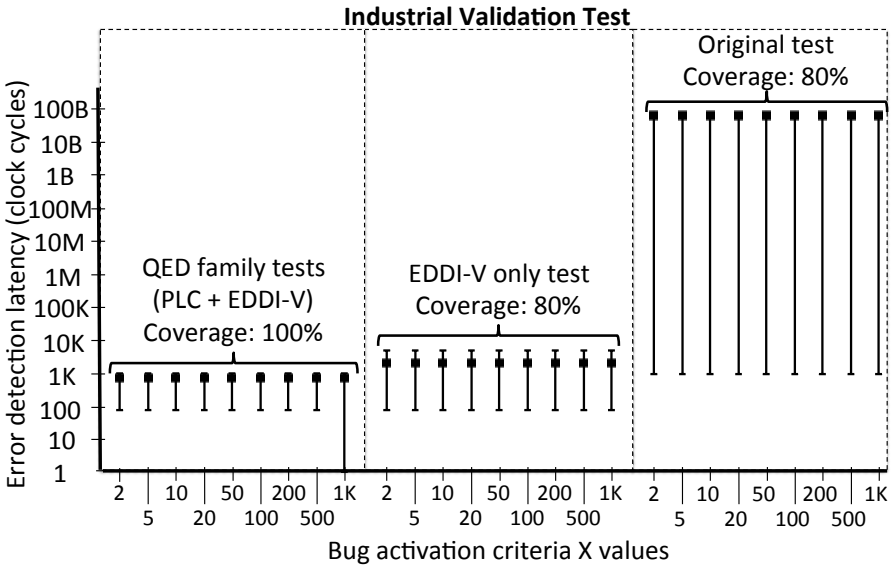


Figure 4.22 Graph showing the minimum, median (represented by the square dot), and maximum error detection latencies by varying both *X* and *Y* parameters of the bug activation criteria and bug effects for Industrial validation test. The coverage numbers reported are absolute (not normalized) coverage.

Table 4.3 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 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 resulting from Table 3.1 and Table 3.2. 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 2-fold 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.

4.10 Intel® Core ™ i7 Hardware Results[2]

Figure 4.23 shows a quad-core Intel® Core™ 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).

[2] The experiments in this section were performed in collaboration with Ted Hong and Yanjing Li.

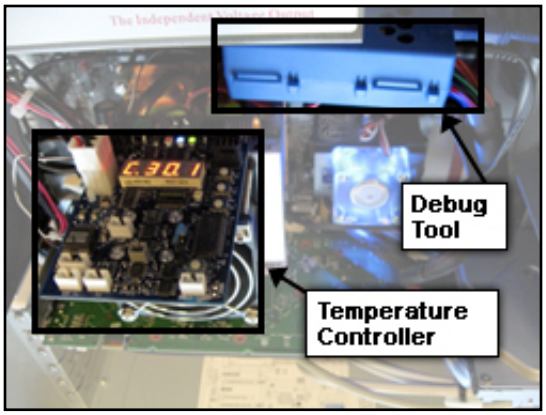


Figure 4.23 Quad-core Intel® Core™ i7 system with a DX58SO motherboard, a temperature controller, and a debug tool.

A. Error Detection Latency Results

We injected errors into the Intel® Core ™ i7 SoC hardware platform to measure the error detection latencies as well as coverage 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 (1.02V, 1.6GHz) and (1.02V, 3.2GHz), 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 1.02V. By fixing the voltage at 1.02V 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 1.02V and two steps (i.e., 266 MHz) below 3.2GHz, 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 Figure 4.24. In, Figure 4.24 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 [Dongarra 03]. 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 Figure 4.24, the results of the *QED test* are when we take into account the QED checks. Results of the *Original test* shown in Figure 4.24 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 ~ 6,000 clock cycles, as shown in Figure 4.24 (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 pre-generated 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 Figure 4.24 and Figure 4.25 does result in longer test execution times (approximately 2-fold). However, the overall debug time, and hence productivity, can improve drastically due to significantly shorter error detection latencies.

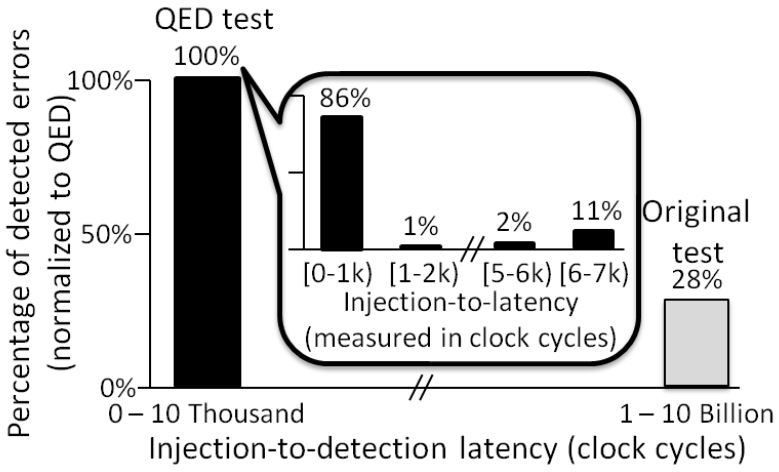


Figure 4.24 Histogram showing the distribution of measured injection-to-detection latencies for the Linpack test, which consists of 75 injection-to-detection latencies obtained from vulnerability window injections that resulted in errors detected by but not crashes. For confidentiality reasons, the percentage of detected errors is normalized to QED.

The following observations can be made from these results:

Observation 4: QED significantly reduces error detection latencies by 6 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.

Observation 5: QED detects errors that would otherwise remain undetected by the original test. QED results in significant (4-fold) 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 Figure 4.25. Each frequency and voltage operating point is classified as:

Did not boot – the system could not boot.

Test passed – both the original test and the QED test did not detect an error.

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.

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 Figure 4.25 (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 Figure 4.25 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.

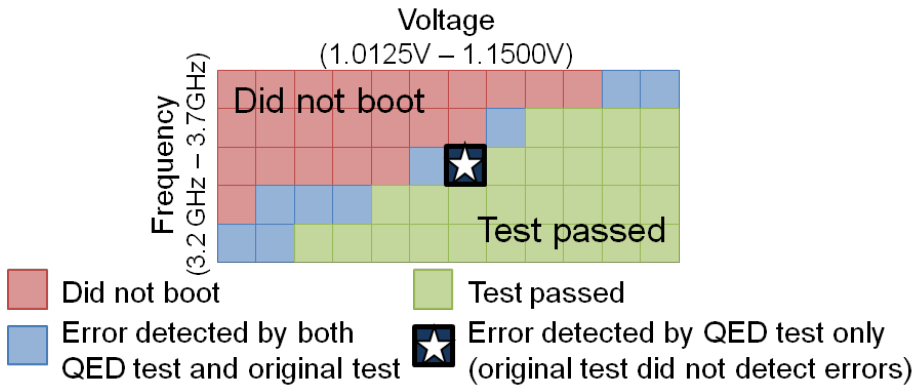


Figure 4.25 Linpack Shmoo plot. The voltage and frequency operating point labeled with a star represents unique error detection by the QED test: the original test did not detect any errors, whereas the QED test detected errors quickly.

4.11 Related Work

Existing related work 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

Chapter 2 already demonstrated that self-checking tests ([Aharon 95, Raina 09, Wagner 08]) 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 ([Benso 08, Bernardi 08, Katz 12, Parvathala 02, Shen 98]]) to improve the error detection latencies and coverage of such tests. QED can also improve the error detection latencies of tests that rely on multi-pass 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 QED's short error detection latencies and improved coverage. These include post-silicon debug techniques that rely on failure reproduction [De Paula 11, 12, Yang 09], simulation [Chang 07, Krstic 03], trace buffers [Abramovici 06, De Paula 11, 12, Park 09, 10] formal methods [De Paula 11, 12, Ho 09, Zhu 11], or emulator / accelerator based debugging [Keshava 10, Para 07]. QED is also compatible with techniques for enhancing the observability of on-chip signals during post-silicon validation [Abramovici 06, ARM CoreSight, DeOrio 09, Neishaburi 11, Tektronix]. 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 Sec. 4.7, 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. Sec. 4.4) while reducing test runtime overhead (e.g. Sec. 4.2C.).
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 [Josephson 06]. Some test runtime overhead can be tolerated if error detection latencies are significantly improved.

In addition, one can incorporate application-specific checks, such as algorithm-based fault-tolerance (ABFT) [Huang 84, Saxena 94], into QED transformations to further improve error detection coverage as well as error detection latency.

D. Memory Scrubbing for Fault-Tolerant Computing

Memory scrubbing [Abraham 83, Shirvani 00] is used in fault-tolerant computing to detect and correct errors inside memory arrays. 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:

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 [Bentley 01, Vasudevan 10]. While there exist techniques to automatically generating assertions for a given design [El Mandouh 12, Hangal 05, Li 10, Vasudevan 10], it may be difficult to implement all of such assertions in hardware. Reconfigurable logic can ease the implementation of assertions in hardware [Abramovici 06], but one has to be careful about selecting the relevant set of assertions to implement in hardware for effective post-silicon debug [Mitra 10]. 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.

4.12 Summary and Discussions

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 multi-core SoC hardware platform demonstrate 9 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™ i7 hardware platform demonstrate 6 orders of magnitude improvement in error detection latencies and simultaneously 4-fold 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 2-fold improvement in coverage. Such short error detection latencies can significantly improve post-silicon validation productivity. The software-only QED does not require any hardware changes and is readily applicable to existing designs.

CHAPTER 5. FAST QUICK ERROR DETECTION (FAST QED)

This chapter presents the Fast QED technique. Fast QED uses a fine-tuned mix of software techniques and hardware support to reduce the runtimes of QED family tests while, at the same time, preserves the short error detection latency and improved bug detection coverage benefits of software-only QED. Simulation results using an OpenSPARC T2-like SoC (a 500-million-transistor design) and a list of difficult logic bug scenarios demonstrate that Fast QED improves error detection latencies by up to 4 orders of magnitude and improves bug detection coverage by up to 2-fold when compared to traditional post-silicon validation tests that rely on end result checks. Fast QED incurs less than 0.4% chip-level area overhead.

The previous chapter presented the software-only Quick Error Detection (*QED*) technique to create post-silicon validation tests with very short error detection latencies and improved coverage. Software-only QED uses software techniques, referred to as *QED transformations*, to transform existing post-silicon validation tests into new QED

tests. A drawback of software-only QED is that the corresponding QED test runtimes can increase significantly (by up to 5 orders of magnitude depending on the QED transformations used). Since debug time is the main bottleneck in post-silicon validation and debug (where the tests are running on actual silicon, which are several orders of magnitudes faster than simulation or emulation), some test runtime increase (e.g., up to an order of magnitude) can be acceptable since error detection latencies and bug coverage are significantly improved. For example, 4-10X increases in test runtime are routinely observed in many post-silicon validation techniques that also incur very long error detection latencies [Adir 11, De Paula 11, Foutris 11, Wagner 08]. However, a large increase in test runtime, e.g., on the scale of 5 orders of magnitude, can be highly challenging because post-silicon validation is already severely constrained in terms of time and hardware resources [Adir 11, Bojan 07]. For example, an existing test that takes 1 minute to run may now end up taking a week. This problem gets worse for emulator / accelerator-based verification and debug since such platforms are slower than actual silicon.

This chapter presents the Fast Quick Error Detection (Fast QED) technique that overcomes the runtime challenge of software-only QED by using a fine-tuned mix of hardware and software techniques to create post-silicon validation tests with short error detection latencies and high coverage. This chapter presents the details of the Fast QED technique and demonstrates the follow benefits of Fast QED:

1. Fast QED enables up to 4 orders of magnitude improvement in test runtimes compared to software-only QED.

2. Post-place-and-route results for the multi-core OpenSPARC T2 SoC [OpenSPARC] demonstrate that Fast QED incurs only 0.4% increase in chip-level area, and negligible increase in chip-level power and performance.

3. Simulation results for a complex OpenSPARC T2-like SoC with bug scenarios from Chapter 3 demonstrate that Fast QED preserves the improved error detection latency and coverage benefits of software-only QED (up to 4 orders of magnitude improvement in error detection latencies and up to 2-fold improvement in bug coverage compared to non-QED post-silicon validation tests), while significantly improving test runtimes (as stated above). In fact, Fast QED achieves further error detection latency improvements compared to software-only QED.

The rest of this chapter is organized as follows. Sec. 5.1 presents an overview of the Fast QED technique. Sections 5.2 - 5.5 provides a detailed explanation of the Fast QED technique. An example of the Fast QED technique is provided in Sec. 5.6 Since Fast QED relies on hardware structures, Sec. 5.7 discusses technique to reduce the area overhead of Fast QED by reusing existing on-chip hardware structures found in many ICs. Results are presented in Sec. 5.8. Related work is discussed in Sec. 5.9. Followed by summary and discussions in Sec. 5.10.

5.1 Fast QED Overview

The software-only QED technique demonstrated in Chapter 4 transforms existing post-silicon validation tests (referred to as original tests) into QED tests using various QED transformations: software-only EDDI-V, CFCSS-V, CFTSS-V, and PLC. The software-only *Proactive Load and Check* (PLC) transformation incurs the highest runtime overhead [Lin 12, 14]. Hence, for Fast QED, we focus on Fast PLC, which improves test runtimes compared to software-only PLC.

An example of the software-only PLC transformation is shown in Figure 5.1. The software-only PLC transformation inserts Proactive Load and Check operations (*PLC operations*) into the test (Figure 5.1). The PLC operations execute on all threads running on all processor cores (Figure 5.1a). Each PLC operation (Figure 5.1b) proactively loads variables in the test and performs checks on the loaded values to quickly detect errors. The granularity at which the PLC operations are inserted is determined by the *Inst_min* and *Inst_max* parameters, defined as the minimum and maximum number of instructions from the original test that must execute before any instruction inserted by QED executes.

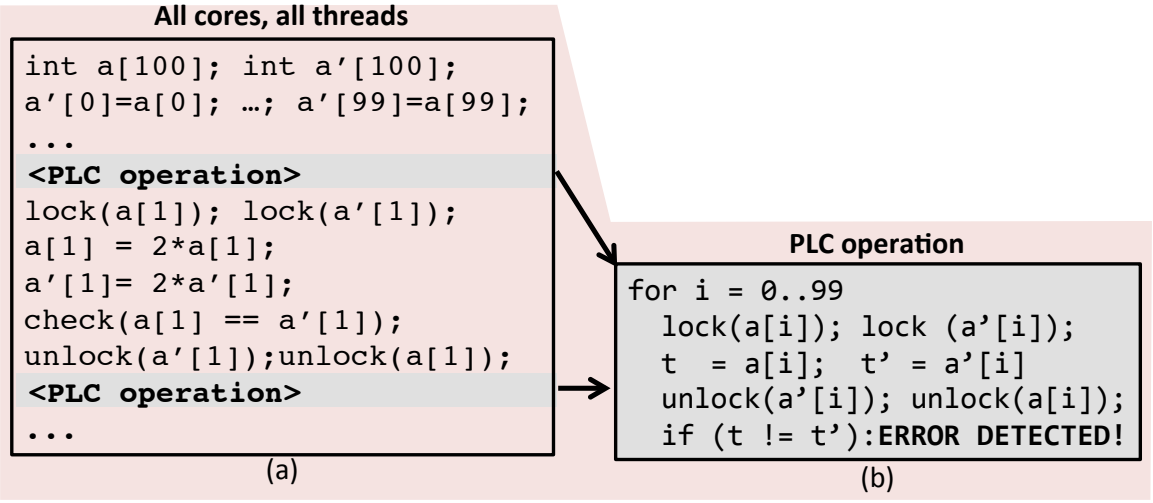


Figure 5.1 An example of the software-only PLC transformation. (a) Insertion of the software-only PLC operations in all threads on all processor cores. (b) The software-only PLC operation.

Instead of performing PLC operations in software running on processor cores, the Fast PLC technique uses small hardware checkers, referred to as *PLC-Hardware (PLC-H) checkers*, to perform PLC operations in order to quickly detect bugs inside uncore components and to reduce test runtimes. Multiple PLC-H checkers can perform PLC operations concurrently (Sec. 5.5). PLC-H checkers are inserted in all cache memories: L1, L2, and any higher-level caches (L3, L4 if they exist in the design). For OpenSPARC T2, we insert PLC-H checkers in the L1 and L2 caches (Sec. 5.5) since the design does not have any higher-level caches. We do not insert PLC-H checkers for other on-chip memories such as register files, TLB, and FIFO buffers, because bugs that affect these structures can be quickly detected using a combination of the software-only EDDI-V [Hong 10, Lin 14] and the PLC-H checkers for cache memories. We do not insert PLC-H checkers for external DRAMs because we target bugs inside SoCs.

However, PLC-H checkers can be extended to detect errors caused by bugs in external DRAMs.

The uncore components in an SoC can be broadly classified into:

(a) Uncore components that are part of the cache subsystem: cache memories as well as the cache controllers.

(b) Uncore components (not in *a*) that communicate with processor cores using cache memories (via loads / stores). For OpenSPARC T2, this category includes memory controllers, network interface unit (network controller), PCI Express interface, and on-chip interconnection network connecting processor cores to caches.

(c) Uncore components that do not use cache memories to communicate with processor cores, but instead, use special instructions to communicate (e.g., I/O instructions). For OpenSPARC T2, this includes the programmable I/O and the interrupt-processing unit.

Fast PLC uses PLC-H checkers to target bugs inside components in the first two categories by performing PLC operations on the caches. Bugs inside components in category *c*, as well as inside processor cores, are quickly detected using the EDDI-V transformation of software-only QED. Example of EDDI-V for bugs in category *c* is shown in Figure 5.1d, where we duplicate the I/O instruction and check the results. Figure 5.2 summarizes the uncore components covered by PLC-H checkers. The following sections present the details of Fast PLC.

Uncore category	Examples in the OpenSPARC T2 SoC	Covered by PLC-H checkers?
(a)	L1 caches, L2 caches, cache controllers	Yes, because bugs in these components will manifest as errors in the caches, which are checked by the PLC-H checkers.
(b)	Memory controllers, network interface unit, PCI Express interface, on-chip interconnection network	Yes, because bugs in these components will manifest as errors in the caches, which are checked by the PLC-H checkers
(c)	Programmable I/Os, interrupt processing unit	Covered by Software-only EDDI-V

Figure 5.2 Uncore components covered by PLC-H checkers.

5.2 Validation Test Preparation for Fast QED

Before a validation test can take advantage of the Fast PLC technique using PLC-H checkers, it is transformed using the software-only EDDI-V QED transformation (Chapter 4) to ensure that bugs inside processor cores as well as uncore components in category *c* above are quickly detected. Figure 5.3 shows an example of the EDDI-V transformation. Figure 5.3a shows the existing validation test, and Figure 5.3b shows the software-only EDDI-V transformed test with initialization of variables, duplication of instructions, and checking the results of original instructions vs. the results of the corresponding duplicated instructions.

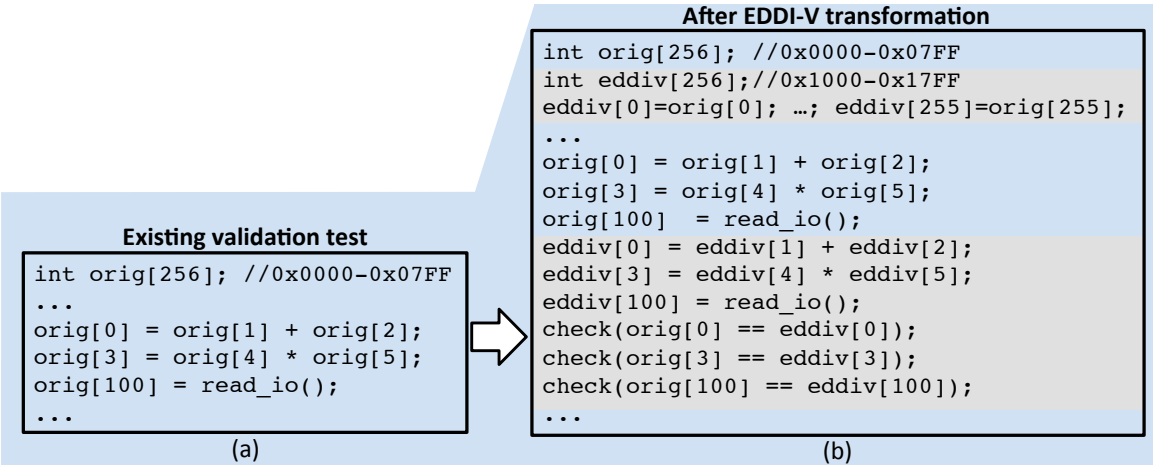


Figure 5.3 An example of the software-only EDDI-V based QED transformation. (a) Existing validation test. (b) After EDDI-V Transformation.

5.3 PLC-H Checker Description

A PLC-H checker proactively loads from a variable in the original test (referred to as an original variable) and its corresponding variable created by the EDDI-V transformation (referred to as an EDDI-V variable), and performs a check by comparing the values of the two variables. The addresses corresponding to the original variables and the EDDI-V variables are programmed into the address generator of the PLC-H checker (described below). If either the original or the corresponding duplicated variables are not cached on-chip (e.g., they are only stored in an off-chip memory such as DRAM) then they are not checked by the PLC-H checker. Any mismatch indicates an error. Figure 5.4 shows a single PLC-H checker integrated into a memory built-in-self-test (*MBIST*) engine. Such MBIST engines are found in many SoCs [OpenSPARC, Poehl 10, Zarrineh

00]. If no MBIST engine exists, PLC-H does not reuse any MBIST components. Each PLC-H checker consists of the following:

1. The **PLC-H controller** controls other components in the PLC-H checker. It is responsible for determining when the PLC-H checker can perform PLC operations (details of the PLC-H controller is presented in Sec. 5.4).

2. When instructed by the PLC-H controller, the **address generator** generates the address of an original variable or the corresponding EDDI-V variable. The address ranges corresponding to the original variables and the EDDI-V variables are programmed into the address generator via debuggers (e.g., JTAG). If a separate PLC-H checker is used for each cache memory array (as demonstrated in Sec. 5.5), the address generator only needs to generate addresses of variables that are cached in that particular cache memory array. For example, in OpenSPARC T2, the address range cached by the cache memory array 0 of bank 0 of the L2 cache is 0x0000-0x7FFF. If the address range of the original variables spans 0x6000-0x6FFF and 0x8000-0x8FFF, then the address generator of the PLC-H checker for that cache array only needs to generate addresses 0x6000-0x6FFF. The address range for each cache memory array is known from the design specification. To simplify address generation, during software-only EDDI-V transformation, our memory allocation library function partitions the memory address into two sets: one for the original variables and one for the EDDI-V variables. This is achieved by partitioning the memory addresses into "chunks" of 0x1000 addresses, alternating between addresses for original variables and addresses for EDDI-V variables

(e.g., addresses 0x0000-0x0FFF, 0x2000-0x2FFF, etc. for original variables, and 0x1000-0x1FFF, 0x3000-0x3FFF, etc. for EDDI-V variables). For an original variable with address *A,* its corresponding EDDI-V variable is stored in address *A*+0x1000. For OpenSPARC T2, the smallest cache memory array (for both L1 and L2) is 8Kbytes; by dividing the memory address into 0x1000 chunks (4KBytes), this partition scheme ensures that both the original address range and its corresponding EDDI-V address range can be cached in the same array. The chunk size is a configurable parameter (we chose 0x1000 for OpenSPARC T2).

3. When instructed by the PLC-H controller, the **data register** holds the value loaded from the cache memory array.

4. The **comparator** compares the value held in the data register with the value loaded from the cache memory. Any mismatch indicates error. The error signal is mapped to on-chip debug circuit (e.g., JTAG).

5. The **multiplexers** select between PLC-H, MBIST, and normal modes. During normal and MBIST modes, the PLC-H checker does not perform any PLC operation. During MBIST mode, the PLC-H checker allows the MBIST to test the array. The MBIST mode is not needed if PLC-H checkers do not reuse MBIST components (Sec. 5.7).

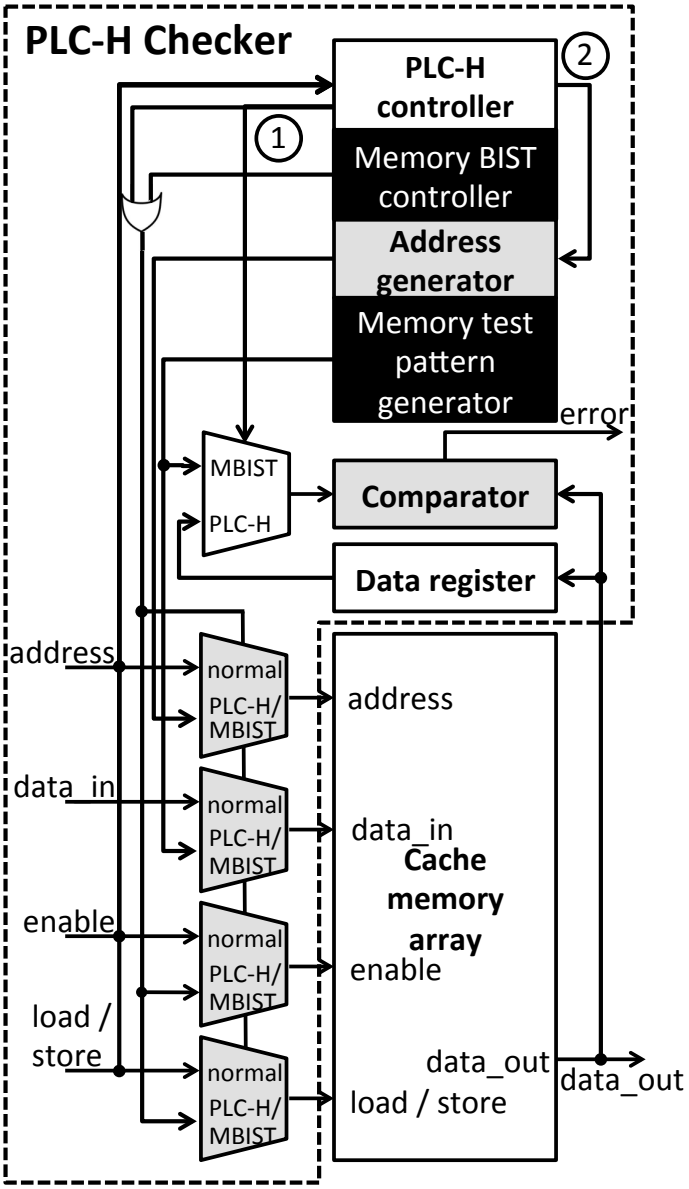


Figure 5.4 PLC-H checker block diagram. Components shaded in gray can be shared between an MBIST engine and a PLC-H checker; components in black are part of an MBIST engine (not used by PLC-H checker).

5.4 PLC-H Controller

The PLC-H controller ensures that a PLC operation is performed only when the following criteria are satisfied:

Criterion 1. The cache memory array being checked does not have a load/store operation in progress. This is done by monitoring the cache array enable signal from processor cores or uncore components.

Criterion 2. The PLC-H checker should not introduce excessive intrusiveness (in order to prevent situations when a bug becomes undetected by Fast PLC due to excessive PLC operations). This is satisfied by counting the number of load / store operations (*OP_cnt* in Figure 5.5). A PLC operation is only performed when *OP_cnt* equals the configurable parameter *OP_cnt_min* (the minimum number of normal load/store operations that must execute before a PLC operation occurs).

Criterion 3. A PLC operation must not happen in between a store to an original variable and a store to the corresponding EDDI-V variable. This ensures that the PLC-H checker does not cause false fails. In software-based PLC, locks are used for this purpose. However, locks may not be available to PLC-H checkers, and improperly-implemented locks may lead to deadlocks. To satisfy criterion 3, we utilize the fact that EDDI-V always inserts EDDI-V store instructions in the same order as that of the original store instructions (e.g., in Figure 5.1d, *orig[0]* is written first, followed by *orig[3]*; correspondingly, *eddiv[0]* is written first, followed by *eddiv[3]*). As a result, we count the number of store operations (*ST_cnt* in Figure 5.5) to original and EDDI-V variables,

and only perform a PLC operation when the two numbers match (i.e., *ST_cnt* == 0 in Figure 5.5). This approach is effective for strong memory ordering architecture (e.g., x86, x86_64, AMD64, and SPARC [McKenney 05]), which ensures that store operations are never reordered.

Architectures with weak memory ordering (e.g., ARM, IA64, and POWER [McKenney 05]) can reorder store operations. For such architectures, we insert a memory barrier instruction (in software) [McKenney 05] after every store instruction inserted by the software-only EDDI-V to ensure that those stores are not reordered. While the barriers can introduce some intrusiveness, the degree of intrusiveness can be systematically adjusted and controlled using the transformation parameter *Inst_min* for EDDI-V (Sec. 4.4), defined as the minimum number of instructions from the original test that must execute before any instructions (including memory barriers) inserted by the QED transformation can execute. A large *Inst_min* means that the memory barriers are inserted infrequently and longer sequences of memory instructions may execute in the original reordered state. In Sec. 5.8, results are presented for both the SPARC with strong memory ordering, and an architecture with weak memory ordering. Empirical results demonstrate that the insertion of memory barriers (controlled by *Inst_min*) for weak memory ordering architectures does not change the error detection latency and bug coverage benefits of Fast QED (Sec. 5.8).

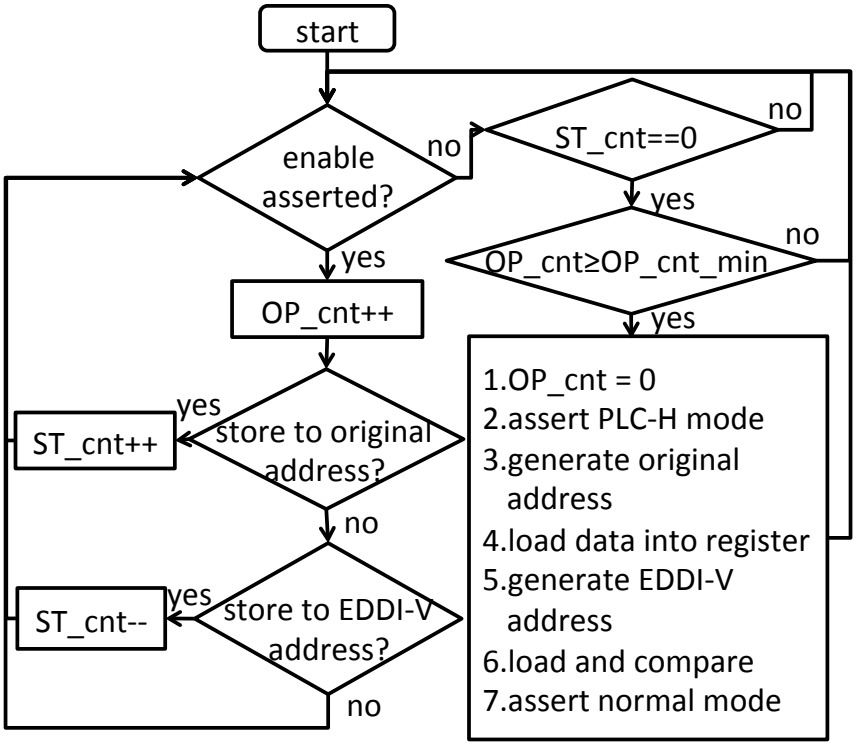


Figure 5.5 Flowchart of PLC-H controller.

5.5 PLC-H Checker Insertion Strategy

We use the OpenSPARC T2 SoC as a case study to discuss strategies for inserting PLC-H checkers. One option is to insert only a single PLC-H checker for the entire SoC. Since the PLC-H checker must perform PLC operations for all cache memories in the SoC, this can take an extremely long time. The OpenSPARC T2 has 4 MBytes of L2 cache with a maximum read size of 64 Bytes. Thus, it takes a minimum of 4 MBytes / 64 Bytes = 65,536 clock cycles to perform PLC operations for the entire L2 cache. Without careful insertion of PLC-H checkers, bugs may not be detected quickly. To overcome this limitation, multiple PLC-H checkers are inserted. For OpenSPARC T2, we insert 136

PLC-H checkers: one for each L1 data cache (8 total), and 16 for each bank of the eight L2 cache banks (128 total: each bank of L2 is constructed from 16 separate memory arrays). Each L1 cache contains 512 entries (cache lines), and one entry is read every cycle; thus, the entire L1 cache can be checked in 512 cycles. Each memory array in the L2 cache also contains 512 entries, but it takes 2 cycles to read an entry. Thus, it takes a minimum of 1,024 cycles to check all L2 memory arrays. Actual cycle count is higher due to *OP_cnt_min*, which controls how often PLC-H checkers check the memory array. The PLC-H checkers perform PLC operations concurrently. Figure 5.6 shows the PLC-H checkers inserted in the OpenSPARC T2.

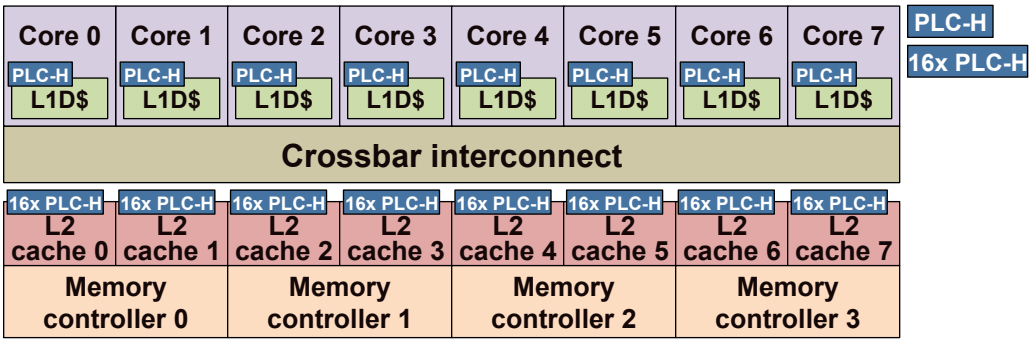


Figure 5.6 OpenSPARC T2 SoC with PLC-H checkers.

5.6 PLC-H Checker Example

When the PLC-H controller determines that the three criteria (Sec. 5.4) are satisfied, it asserts "PLC-H" on signal ① in Figure 5.4 to perform a single PLC operation; during this time, the cache does not respond to normal load / store operations in the system (e.g., from processor cores). These operations are not lost, but are held at the input buffers of the caches (input buffers hold pending load / store operations when

the cache is busy). For OpenSPARC T2, the input buffer for each cache can hold 8 pending load / store operations. If the buffers are full, further load / store operations are stalled.

Next, the PLC-H controller instructs the address generator (using signal ②) to generate the address of an original variable (e.g., 0x0000 for *orig[0]* in Figure 5.1d). This address is looked up in the tag entry of the cache to determine its location in the cache memory array (e.g., the 1st entry in the L1 cache memory array). If the address does not exist in the tag entry (i.e., a cache miss), then it is not loaded and PLC operation is not performed on this variable. If the address does exist, its value is loaded into the data register. Next, the PLC-H controller instructs the address generator to generate the address of the corresponding EDDI-V variable (e.g., 0x1000 for *eddiv[0]* in Figure 5.1d). This address is looked up in the tag entry to determine its location in the cache memory array (e.g., the 256th entry in the L1 cache memory array). If this address exists, it is loaded and its value is compared to the corresponding original variable's value stored in the data register. Any mismatch indicates an error. If the address does not exist, PLC operation is not performed on this variable. The PLC-H controller then asserts "normal" on signal ① in Figure 5.4 to allow the cache to respond to normal load / store operations. This completes a single PLC operation.

5.7 Memory BIST Reuse to Minimize PLC-Checker Cost

In this section, we demonstrate how reusing MBIST engines in SoCs can minimize the area and power costs of PLC-H checkers. The block diagram of a PLC-H checker integrated into a "non-programmable" MBIST engine in OpenSPARC T2 is shown in Figure 5.4. Programmable MBIST engines [Zarrineh 00] that exist in many SoCs can further reduce the area of PLC-H checkers: we can reuse the already-existing address generators in programmable MBIST engines.

We used the Synopsys Design Compiler with the Synopsys EDK 32nm library (standard cells and memories) for synthesis and Synopsys IC compiler for place-and-route to estimate the area of PLC-H checkers. 136 PLC-H checkers are used on the OpenSPARC T2 (Sec. 5.5), which contains 80 non-programmable MBIST engines; 56 of these engines are located in the L1 and L2, which we reuse for PLC-H checkers. The remaining 80 PLC-H checkers do not reuse MBIST engines. The 136 PLC-H checkers introduce 0.3% chip-level area overhead after synthesis, and 0.4% chip-level area overhead after place-and-route. PLC-H checkers can be used for emulation-based verification and debug, and are added only in the design mapped on the emulator (not necessarily in the final design). Therefore, they do not introduce any area, power, or performance costs in the final design.

5.8 Results

To demonstrate the effectiveness of Fast QED in achieving fast runtimes, short error detection latencies, and high coverage for a wide variety of bugs, we present

simulation results for the bug scenarios in Chapter 3. We used GEMS [Martin 05] to simulate an OpenSPARC T2-like multi-core SoC (Figure 5.6), a 500-million-transistor design, with 8 processor cores (64 hardware threads total). Each processor core has private L1 caches. A crossbar connects the processor cores to 8 banks of shared L2 cache and 4 memory controllers. The OpenSPARC T2 implements Total Store Ordering [OpenSPARC] (a form of strong memory ordering).

Bug scenarios from Chapter 3 were simulated using the methodology in Sec. 4.9. For the power management bug scenarios in Sec. 3.1 we simulated the bug effects when the system exits from the power-saving state. To simulate exit from the power-saving state, we randomly selected a sequence of 5 instructions from the *original test* to serve as a trigger for entering power-saving state (a system enters power-saving state when it executes a specific sequence of instructions, i.e., to save the system states). On the next clock cycle, the system is considered to have exited the power-saving state and the bug effect is simulated (we consulted with validation engineers to ensure the validity of this approach). The sequence of instructions for the trigger to enter power-saving state is the same for the original test, the QED test, and the Fast QED test; this allows us to quantify any intrusiveness introduced by Fast QED.

We used a combination of SPLASH-2 [Woo 95] and SPECINT 2000 [Henning 00] benchmarks as the original tests. For SPLASH-2, we used 8-threaded versions of the tests (one thread per processor core). For SPECINT 2000 (which are single threaded), each of the 8 cores runs a single instance of the test. 64-threaded versions were not used

because the simulator is not cycle-accurate when simulating processor core supporting more than 1 hardware thread [Martin 05].

The runtime and area cost (post place-and-route) is summarized in Table 5.1. Fast PLC introduces 5-6X increase in test runtime on the OpenSPARC T2 SoC compared to the original tests because the runtime of Fast PLC is bounded by the runtime of software-only EDDI-V used by Fast PLC. For most instructions in the original test, EDDI-V inserts one duplicated instruction, one compare instruction, and a conditional branch instruction (to indicate if an error is detected). On non-superscalar processors (e.g., in OpenSPARC T2), this is approximately 4X increase in runtime (actual runtime may be slightly higher due to cache misses). On superscalar processors, the runtime overhead of EDDI-V tests is only 1.1X to 2X [Oh 02a]; hence, the runtime of Fast PLC can be further reduced for superscalar processors.

Given the growing complexity and importance of uncore components in SoCs, if only uncore bugs are targeted, we can reduce the runtime of Fast QED by duplicating only store instructions from the original test. This reduces the runtime overhead of Fast QED to less than 15% ("Fast QED (uncore only)" in Table 5.1), and does not change the error detection latencies and coverage of Fast QED for uncore bugs (i.e., activation criterion 1 in Table 3.3, bug effect A-I in Table 3.4, activation criterion 1-5 in Table 3.1 and bug effect A-G in Table 3.2) (Figure 5.8 to Figure 5.12).

To put the runtime of Fast QED into perspective, many post-silicon validation techniques also incur long test runtimes (and may incur millions or billions cycles of

error detection latencies, unlike Fast QED). The multi-pass checking technique [Adir 11] requires running a test multiple times resulting in several fold increase in test runtime and can incur millions or billions of cycles of error detection latencies. The technique in [De Paula 11] requires running original tests multiple times (e.g., 10). The technique in [Foutris 11] increases test runtime by 6X, and [Wagner 08] increases test runtime by approximately 4X.

Table 5.1 Comparison of normalized runtimes and area costs for software-only QED vs. Fast QED and Fast QED for uncore only bugs.

Techniques	Area cost	Normalized runtimes								
		bzip2	crafty	fft	lu	mcf	parser	vpr	quake	art
Original	0%	1X	1X	1X	1X	1X	1X	1X	1X	1X
Software-only QED (PLC) (Chapter 4)	0%	52,224X	59,392X	28,672X	32,768X	58,368X	55,234X	56,134X	57,382X	51,023X
Fast QED	0.4%	5.22X	5.92X	5.52X	5.84X	6.15X	6.12X	5.78X	6.22X	6.36X
Fast QED (uncore only)	0.4%	1.13X	1.10X	1.11X	1.04X	1.14X	1.12X	1.12X	1.10X	1.14X

Figure 5.7 illustrates the tradeoffs between number of the PLC-H checkers inserted and the distribution of error detection latencies. The horizontal axis corresponds to the number of PLC-H checkers inserted and the area cost (on OpenSPARC T2). The vertical axis represents the distribution of error detection latencies showing the minimum, median (represented by a square dot) and the maximum error detection latencies for bug scenarios in Chapter 3. The number of PLC-H checkers start from 16 (1 for each of the 8 L1 caches and 1 for each of bank of the 8 L2 cache banks in OpenSPARC T2). We gradually increase the number of PLC-H checkers in each bank of L2 from 1 to 16 (1

PLC-H per array, 2 PLC-H per array, and so on), while the number of PLC-H checkers in L1 stays constant, until we reach 136 PLC-H checkers (Sec. 5.5). Area cost ranges from 0.05% (16 PLC-H checkers) to 0.4% (136 PLC-H checkers). With 16 PLC-H checkers (area cost of 0.05%), the median and maximum error detection latencies are 7k and 20k clock cycles, respectively for the *bzip2* and the *lu* tests. With 136 PLC-H checkers (area cost of 0.4%), the median and maximum error detection latencies are 412 and 3k clock cycles, respectively, for the *bzip2* test and 385 and 5k clock cycles, respectively, for the *lu* test. Similar trends are observed for other tests.

Observation 1: Fast QED enables up to 4 orders of magnitude improvement in runtimes compared to worst-case software-only QED tests, with only 0.4% increase in chip area. This, with Observations 2 and 3 presented later, demonstrate that Fast QED achieves major improvement in test runtimes while **simultaneously** preserving the quick error detection and coverage benefits of software-only QED.

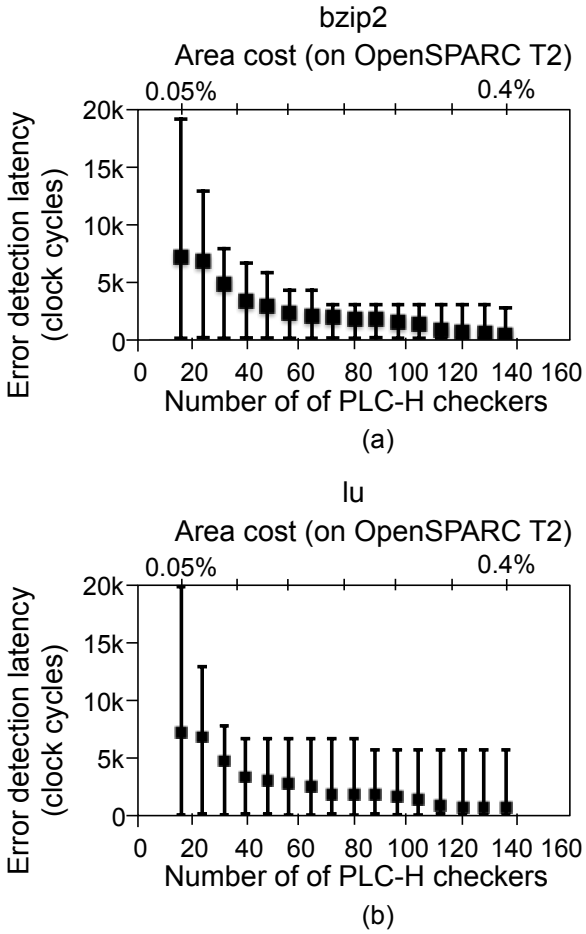
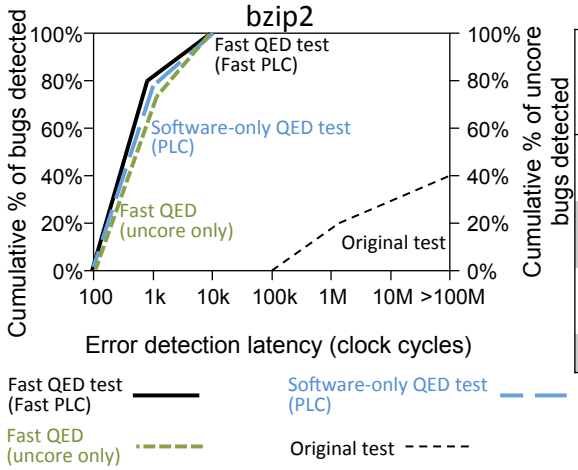


Figure 5.7 Error detection latencies vs. area cost and number of PLC-H checkers for OpenSPARC T2 [OpenSPARC] using bzip2 test from SPECINT [Henning 00] and lu test from SPLASH-2 [Woo 95].

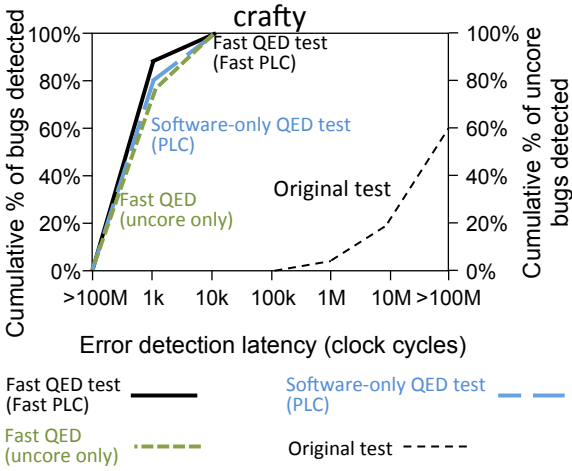
The error detection latencies and coverage are summarized in Figure 5.8 to Figure 5.12, where the horizontal axes represent error detection latencies in clock cycles (logarithmic scale); we also show the median and maximum error detection latencies (*Med.*, *Max. EDL*). The vertical axes represent the cumulative percentage of bug scenarios detected with respect to all bug scenarios in Chapter 3 for "Original test", "Fast QED test" and "Software-only QED test" and with respect to uncore bugs only for "Fast

QED (uncore only)". The "Original test" represents the original test with "end result checks" that compare the results of the test against pre-computed, known correct results. This is possible because the inputs to the tests are known *a priori* (default inputs). The "Fast QED test" is created from the original test using the Fast PLC (which includes software-only EDDI-V, as explained in 5.2). For Fast PLC, the simulated system was modified to include 136 PLC-H checkers (Sec. 5.5). Fast QED do not rely on any information about the bugs and are not specifically tailored to the bugs simulated. The "Software-only QED test" corresponds to QED tests created using software-only PLC (Sec. 4.2) (which includes software-only EDDI-V). "Fast QED (uncore only)" was explained above. *Inst_min*, *Inst_max*, and *OP_cnt_min* are set to 5.



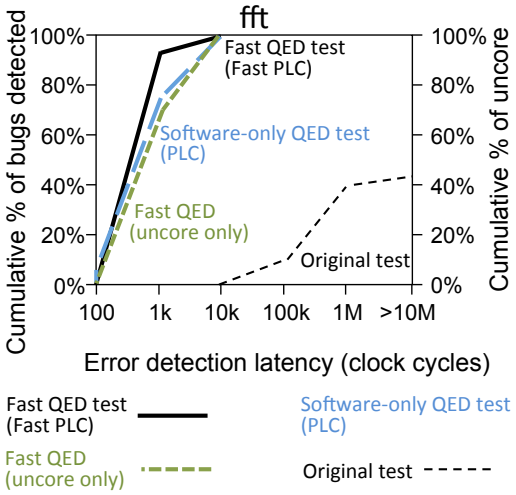
	Error detection latencies (clock cycles)		Runtimes (normalized to original test)
	Median	Maximum	
Fast QED test (Fast PLC)	380	3k	5.2x
Fast QED test (uncore only)	419	3k	1.13x
Software-only QED test (PLC)	392	3k	52,224x
Original test	1M	100M	1x

Figure 5.8 Error detection latency, coverage, and runtime results for the bzip2 test from SPECINT [Henning 00]. The cumulative percentage of bugs detected for "Fast QED test (Fast PLC)", "Software-only QED test (PLC)", and "Original test" are reported for all bug scenarios in Chapter 3. The cumulative percentage of bugs detected for "Fast QED (uncore only)" are with respect to only the uncore bug scenarios.



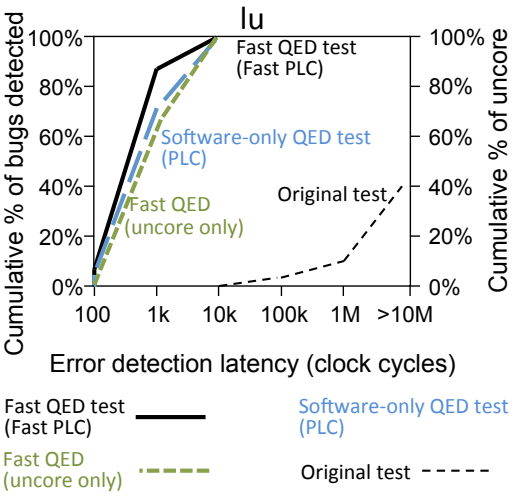
	Error detection latencies (clock cycles)		Runtimes (normalized to original test)
	Median	Maximum	
Fast QED test (Fast PLC)	382	3k	5.9x
Fast QED test (uncore only)	415	3k	1.10x
Software-only QED test (PLC)	399	3k	59,392x
Original test	11M	100M	1x

Figure 5.9 Error detection latency, coverage, and runtime results for the crafty test from SPECINT [Henning 00]. The cumulative percentage of bugs detected for "Fast QED test (Fast PLC)", "Software-only QED test (PLC)", and "Original test" are reported for all bug scenarios in Chapter 3. The cumulative percentage of bugs detected for "Fast QED (uncore only)" are with respect to only the uncore bug scenarios.



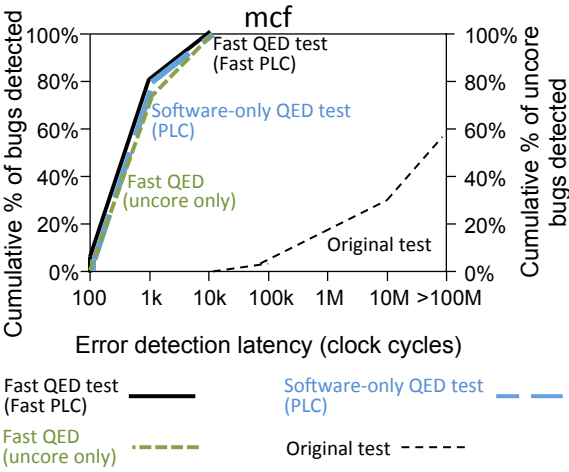
	Error detection latencies (clock cycles)		Runtimes (normalized to original test)
	Median	Maximum	
Fast QED test (Fast PLC)	352	4k	5.5x
Fast QED test (uncore only)	575	4k	1.11x
Software-only QED test (PLC)	450	4k	28,672x
Original test	300k	10M	1x

Figure 5.10 Error detection latency, coverage, and runtime results for the fft test from SPLASH-2 [Woo 95]. The cumulative percentage of bugs detected for "Fast QED test (Fast PLC)", "Software-only QED test (PLC)", and "Original test" are reported for all bug scenarios in Chapter 3. The cumulative percentage of bugs detected for "Fast QED (uncore only)" are with respect to only the uncore bug scenarios.



	Error detection latencies (clock cycles)		Runtimes (normalized to original test)
	Median	Maximum	
Fast QED test (Fast PLC)	352	4k	5.8x
Fast QED test (uncore only)	575	4k	1.11x
Software-only QED test (PLC)	450	4k	28,672x
Original test	300k	10M	1x

Figure 5.11 Error detection latency, coverage, and runtime results for the lu test from SPLASH-2 [Woo 95]. The cumulative percentage of bugs detected for "Fast QED test (Fast PLC)", "Software-only QED test (PLC)", and "Original test" are reported for all bug scenarios in Chapter 3. The cumulative percentage of bugs detected for "Fast QED (uncore only)" are with respect to only the uncore bug scenarios.



	Error detection latencies (clock cycles)		Runtimes (normalized to original test)
	Median	Maximum	
Fast QED test (Fast PLC)	402	6k	6.1x
Fast QED test (uncore only)	445	6k	1.14x
Software-only QED test (PLC)	405	6k	58,368x
Original test	10M	100M	1x

Figure 5.12 Error detection latency, coverage, and runtime results for the mcf test from SPECINT [Henning 00]. The cumulative percentage of bugs detected for "Fast QED test (Fast PLC)", "Software-only QED test (PLC)", and "Original test" are reported for all bug scenarios in Chapter 3. The cumulative percentage of bugs detected for "Fast QED (uncore only)" are with respect to only the uncore bug scenarios.

Observation 2: Fast QED retains the error detection latency benefits of software-only QED. Fast QED also achieves lower error detection latencies compared to software-only QED (Figure 5.8 to Figure 5.12) for all bug scenarios. The error detection latencies of the original tests can be up to hundreds of millions of clock cycles. Fast QED detects all bugs with error detection latencies less than 7 thousand clock cycles, and detects most bugs with error detection latencies of only a few hundred cycles.

Observation 3: Fast QED retains the coverage benefits of software-only QED. Fast QED detected all bug scenarios detected by the software-only QED (and original tests). Fast QED (and software-only QED as well) detected up to 2X more bugs vs. original tests. This empirically demonstrates Fast QED does not introduce excessive intrusiveness. We confirmed that the coverage benefits of Fast QED vs. original test are due to Fast QED's check operations and not because of its execution time (similar to the observations in Sec. 4.9).

Observation 4: Fast QED for uncore only retains the error detection latency and coverage benefits of software-only QED for uncore bugs while incurring less than 15% runtime overhead.

To demonstrate effectiveness of Fast QED for weak memory ordering architectures, we modified our simulator to simulate an SoC with weak memory ordering. For the weak memory ordering architecture only, we introduce an additional bug activation criterion to the list of bug activation criteria in Table 3.1 and Table 3.3:

When memory reordering occurs.

Figure 5.13 Additional bug activation criterion to characterize the effectiveness of Fast QED for weak memory ordering architectures.

This activation criterion enables us to evaluate the intrusiveness of Fast QED's memory barriers on weak memory ordering architectures (which allow reordering of memory accesses). This activation criterion is not used for strong memory ordering architectures because memory reordering does not occur in strong memory ordering architectures. We simulated the bug scenarios in Chapter 3 (with the additional bug activation criterion in Figure 5.13) on the weak memory ordering architecture. Error detection latencies, coverage, and runtimes for *bzip2* and *lu* are reported in Figure 5.14 (similar trends were observed for other tests).

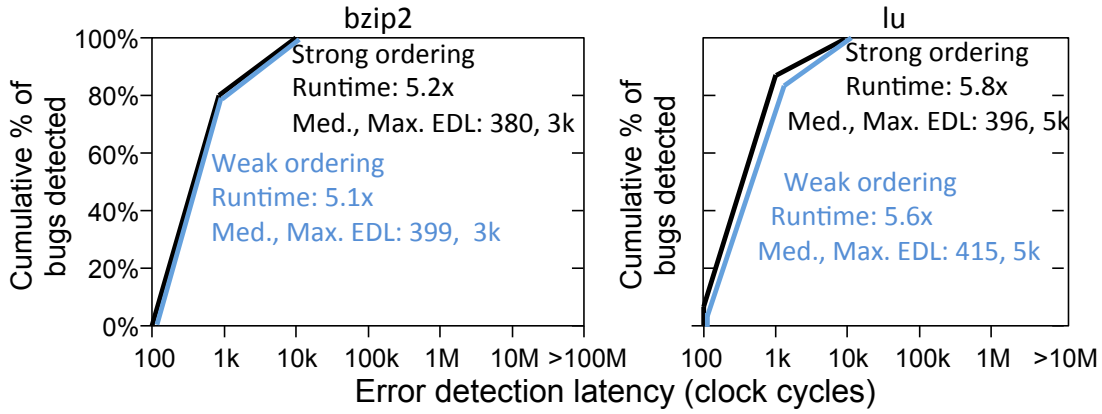


Figure 5.14 Error detection latencies, coverage, and runtimes for bug scenarios in Chapter 3 (with activation criterion 2 used only for weak memory architecture) for both strong and weak memory ordering architectures.

Observation 5: Fast QED is effective for both strong and weak memory ordering architectures. The error detection latencies, coverage and runtimes of Fast QED for strong memory ordering and weak memory ordering are the same. The insertion of memory barriers for Fast QED in the weak memory ordering architecture did result in 5-15% fewer bug activations compared to the original tests (memory barrier instruction prevents the reordering of memory operations before the barrier with memory operations after the barrier), but did not change the error detection latency and coverage benefits of Fast QED.

5.9 Related Work

Similar to software-only QED presented in Chapter 4, Fast QED is a technique for creating effective post-silicon validation tests with short error detection latencies and improved coverage. A drawback of software-only QED is that the QED test runtimes can increase significantly (by up to 5 orders of magnitude depending on the QED transformations used). Fast QED overcomes this challenge with only 0.4% chip-level area cost and negligible increases in system-level power and performance. Fast QED retains the short error detection latency and improved coverage benefits of software-only QED.

Fast QED is different from traditional validation tests [Adir 11, Aharon 95, Foutris 11, Raina 98, Wagner 08] in two respects: 1. Traditional tests can incur extremely long error detection latencies (up to billions of clock cycles) vs. very short error detection latencies using software-only QED and Fast QED; 2. Fast QED incorporates a fine-tuned

mix of hardware and software techniques to reduce test runtime, and improve error detection latency and bug coverage.

PLC-H checkers are different from memory scrubbing techniques [Abraham 83, Shirvani 00] for fault-tolerant computing or MBIST techniques used for memory testing. These techniques generally focus on errors inside memory arrays (sometimes assisted by error-correcting codes), and do not target bugs outside memory arrays (vs. PLC-H targets bugs inside cache controllers, memory controllers, and interconnection networks). Furthermore, scrubbing and MBIST occur infrequently, and can result in very long error detection latencies. There are several techniques for improving observability during post-silicon validation: e.g., trace buffer insertion [Abramovici 06, Basu 11, Ko 08, Liu 09] and memory access logging [DeOrio 08, 09]. These techniques can benefit from the short error detection latencies and high coverage of software-only QED and Fast QED.

Similar to software-only QED, Fast QED is different from design-specific assertions [Abramovici 06, Boule 07, Ernst 07, Hangal 05, Vasudevan 10], which are difficult to keep up-to-date, and to validate their correctness [Bentley 01, Vasudevan 10]. While automatic assertion generation techniques exist [Ernst 07, Hangal 05, Vasudevan 10], it is difficult to select only the relevant set of assertions to reduce area costs while simultaneously ensuring effective debug. Assertions may depend on signals located in different regions of an IC and must be decomposed into components that only use nearby signals. In contrast, Fast QED performs extensive checks, and does not suffer from these

drawbacks. Reconfigurable debug logic for hardware assertions [Abramovici 06] also can be reused for PLC-H checkers.

Upon quick bug detection using Fast QED or software-only QED, post-silicon bug localization techniques can be invoked [Abramovici 06, De Paula 11, Park 09, 10, Zhu 11].

5.10 Summary and Discussions

The Fast QED technique uses a fine-tuned mix of hardware and software techniques to create post-silicon validation tests with short error detection latencies and high coverage benefits of software-only QED tests, without incurring significant runtime overheads of software-only QED. Fast QED enables up to 5 orders of magnitude improvement in error detection latencies and up to 2-fold improvement in coverage compared to the original (non-QED) tests. Fast QED achieves up to 4 orders of magnitude improvement in test runtimes compared to software-only QED, and incurs 0.4% increase in chip-level area, and negligible increases in system-level power and performance. Fast QED enables flexible tradeoffs between area cost, error detection latencies, and test runtime. The use of Fast QED during emulation-based verification and debug enables quick error detection, high coverage, and low test runtime without adding area costs in the manufactured IC.

CHAPTER 6. SYMBOLIC QUICK ERROR DETECTION (SYMBOLIC QED)

This chapter presents the Symbolic QED technique, a systematic and structured approach that automatically and quickly localizes logic bugs, without any human intervention or extra hardware modification. Existing post-silicon validation and debug techniques are mostly ad hoc, and often involve manual steps. Such ad hoc approaches cannot scale with increasing IC complexity. Symbolic QED combines the following steps in a coordinated fashion: 1. Quick Error Detection (QED) tests that quickly detect bugs with short error detection latencies and high coverage. 2. Formal analysis techniques to localize bugs and generate minimal-length bug traces upon detection of the corresponding bugs. We demonstrate the practicality and effectiveness of Symbolic QED using the OpenSPARC T2, a 500-million-transistor open-source multicore System-on-Chip (SoC) design, and using "difficult" logic bug scenarios that occurred in various state-of-the-art commercial multicore SoCs. Our results show that Symbolic QED: (i) is fully automatic (unlike manual techniques in use today that can be extremely time-consuming and expensive); (ii) requires only a few hours in contrast to manual approaches that might take days or weeks (or even months) for difficult bugs or formal techniques that often take days or fail completely for large designs; (iii) generates counter-examples (for activating and detecting logic bugs) that are up to 6 orders of magnitude shorter than those produced by traditional techniques; and, (iv) does not require any additional hardware.

This chapter presents the Symbolic QED for the EDDI-V and the PLC QED transformations. Symbolic QED for the CFCSS-V and the CFTSS-V QED transformations are presented in Appendix 1.

After the bugs are quickly detected by QED, the corresponding bugs must be localized, i.e., by identifying a sequence of inputs (e.g., instructions) that activate and detect the bug (also referred to as a *bug trace*), and the hardware design block where the bug is (possibly) located. This chapter presents the *Symbolic Quick Error Detection* or *Symbolic QED* for quickly and automatically localizing logic bugs. Key characteristics of Symbolic QED are: 1) It is applicable to any System-on-Chip (SoC) design as long as it contains at least one programmable processor (a generally valid assumption for existing SoCs); 2) It is broadly applicable for logic bugs inside processor cores, accelerators, and uncore components; 3) It doesn't require failure reproduction; 4) It doesn't require any human intervention or manual efforts during bug localization; 5) It doesn't require any additional hardware to localize bugs; and 6) It doesn't require design-specific assertions.

This chapter demonstrates the effectiveness and practicality of Symbolic QED by showing that: 1) Symbolic QED correctly and automatically localizes difficult logic bugs in a few hours (less than 7 for OpenSPARC T2 SoC, a 500-million-transistor open-source SoC, see Sec. 6.9). Such bugs would generally take days or weeks of (manual) work to localize using traditional approaches; 2) Symbolic QED does not require additional hardware (such as trace buffers) for localizing logic bugs; 3) For each detected logic bug, Symbolic QED provides a set of candidate components representing the possible locations of the bug in the design; 4) For each detected logic bug, Symbolic QED automatically generates a minimal bug trace using formal analysis; and 5) Symbolic QED generates bug traces that are up to 6 orders of magnitude shorter than those produced by traditional post-silicon validation techniques.

Symbolic QED relies on the following steps that work together in a coordinated fashion: 1) Quick Error Detection (QED) tests that detect bugs with short error detection latencies and high coverage (Chapter 4); and 2) Formal techniques that enable bug localization and generation of minimal bug traces upon bug detection (Sec. 6.2 to Sec. 6.8).

6.1 A Motivating Example

This section present a bug scenario (from Chapter 3) abstracted from an actual difficult bug found during post-silicon validation of a commercial multicore SoC:

Two stores in 2 cycles to adjacent cache lines in a cache delay the next cache coherence message received by that cache by 5 clock cycles.

The bug is *only* activated when two store operations occur within 2 clock cycles of each other to adjacent cache lines. The next cache coherence message (e.g., invalidation) is delayed because of a delay in the receive buffer of the cache (these details were not known before the bug was found and localized).

During post-silicon validation, a test running on the SoC created a deadlock. As shown in Figure 6.1, the deadlock occurs because one of the processor cores (core 4) performed a store to memory location [A] followed by a store to memory location [B] within 2 clock cycles ([A] and [B] were cached on adjacent cache lines). As a result, the bug scenario was activated in cache 4. After the bug is activated, processor core 1 performs a store to memory location [C]. Since memory location [C] was cached in multiple caches (cache 1 and cache 4), the store operation to memory location [C] had to invalidate other cached copies of memory location [C] (including the cached copy in cache 4). However, due to the bug, the invalidation message received by cache 4 was delayed by 5 clock cycles. Before the invalidation occurred, processor core 4 loaded from memory location [C]. Since the cached copy of memory location [C] in cache 4 was still marked as valid, it loaded a stale copy (which contained the wrong value at that point). Then, millions of clock cycles later, processor core 4 used the wrong value of memory location [C] in code that performed locking, resulting in deadlock.

When such a deadlocks is detected (e.g., by using a timeout), the bug must be localized by identifying the bug trace and the component where the bug is located. Since it is not known *a priori* when the bug was activated or when the system entered deadlock, it can be very difficult to obtain the bug trace. Additionally, the bug trace can be extremely long due to the long error detection latency, containing many extraneous instructions unnecessary for activating or detecting the bug. As mentioned above, such bugs can be extremely challenging to localize using traditional approaches such as failure reproduction, trace buffers, simulation, or traditional formal methods.

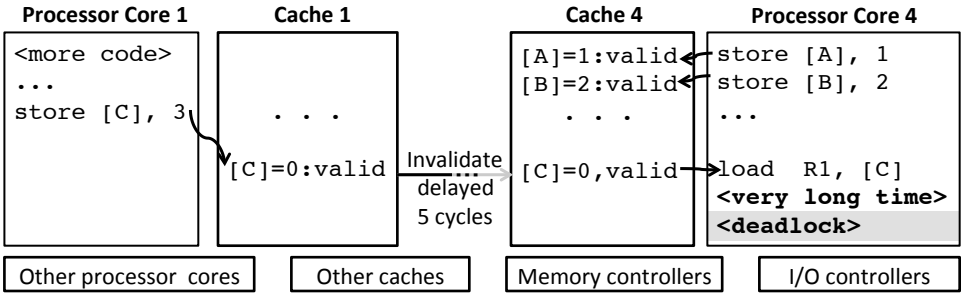


Figure 6.1 An example bug scenario.

As shown in Sec. 6.9, Symbolic QED correctly localizes this bug to cache 4 and produces a bug trace that is only 3 instructions long. Symbolic QED takes only 2.5 hours to automatically localize this logic bug without requiring failure reproduction, any additional hardware, or any manual efforts. This is possible because Symbolic QED uses bounded model checking (BMC), which finds the minimal bug trace, if one exists [Clarke 01] (details in Sec. 6.4). Additionally, Symbolic QED employs special "design reduction" techniques to effectively handle large multi-core SoC designs such as the OpenSPARC T2 SoC (details in Sec. 6.8). In contrast, traditional post-silicon bug localization

approaches would likely require manual effort, additional hardware (e.g., trace buffers), or both, and could take days or weeks. Furthermore, the bug traces found by traditional post-silicon techniques can be significantly longer than those found by Symbolic QED (as demonstrated in Sec. 6.9).

While the main focus of this chapter is post-silicon bug localization, the Symbolic QED technique explained here can also be used for bug detection and localization during pre-silicon verification, as well as emulation-based verification.

The rest of the chapter is organized as follows. Section 6.2 provides an overview of Symbolic QED. Section 6.3 gives a quick overview of the EDDI-V and PLC QED transformations, which are used by Symbolic QED presented in this chapter. Sections 6.4 to 6.8 present the details of the Symbolic QED technique. Results are presented in Sec. 6.9. Related work is discussed in Sec. 6.10, follow by summary in Sec. 6.11. This chapter presents the Symbolic QED technique using the EDDI-V and the PLC QED transformations. Appendix 1 presents details of the Symbolic QED technique for the CFCSS-V and the CFTSS-V QED transformations (to localize bugs that affect the control flows of processor cores).

6.2 Symbolic QED Overview

Symbolic QED localizes bugs and produces short bug traces consisting of only a few instructions (often less than 10) automatically. Within the space of QED-compatible bug traces (explained below), the traces produced by Symbolic QED are *minimal*,

meaning no shorter bug traces exist. These short bug traces make bugs easier to understand and fix.

Symbolic QED relies on bounded model checking (*BMC*), a technique used in formal verification. Given a model of a system (e.g., the RTL) and a property to be checked (e.g., a check inserted by QED), the system is formally analyzed to see if the property can be violated in a bounded number of steps (clock cycles). If so, a *counter-example* (a concrete trace violating the property) is produced. BMC guarantees that if a property can be violated, a minimal-length counter-example is returned [Clarke 01].

We first review three challenges associated with using BMC for post-silicon bug localization:

1. BMC needs a property to check. Since the bugs are not known *a priori*, it is difficult to craft such properties (and avoid false positives);
2. Large design sizes limit the effectiveness of BMC. If a design is too large, a typical BMC tool will not even be able to load the design (see 6.9). Even if a large design can be loaded, running BMC on it is likely to be very slow; and,
3. The performance of BMC techniques is affected by the number of cycles required to trigger and observe a bug. As the number of cycles increases, BMC performance slows down, especially for large designs. Thus, unless a short counter-example exists, BMC will take too long or will be unable to find it.

Challenge 2 is addressed in Sec. 6.8. Here, we focus on challenges 1 and 3. The key idea is to create a BMC problem that searches through ***all possible*** QED tests. As shown in Chapters 4 and 5, QED tests are excellent for detecting a wide variety of bugs; therefore, we use the QED checks (details in Sec. 6.4) as the property for the BMC tool, thus addressing challenge 1. As shown in Chapters 4 and 5, QED tests are also designed to be able to find errors quickly. By searching all possible QED tests using the minimality guarantees of BMC, it is usually possible to find a very short trace triggering the bug, addressing challenge (3). We explain the details of Symbolic QED in Sections 6.4 – 6.9. First, we provide an overview of the EDDI-V and the PLC QED transformations, which are used by Symbolic QED.

6.3 EDDI-V and PLC Preliminaries

Since the Symbolic QED technique in this chapter uses the EDDI-V (Error Detection by Duplicated Instruction for Validation) and the PLC (Proactive Load and Check) QED techniques, this section provides a quick overview of EDDI-V and PLC QED transformations.

A. EDDI-V

EDDI-V targets processor core bugs by frequently checking the results of original instructions against the results of duplicated instructions created by EDDI-V. First, the registers and memory space are divided into two halves, one for the original instructions and one for the duplicated instructions. Next, corresponding registers and memory

locations for the original and the duplicated instructions are initialized to the same values. Then, for every load, store, arithmetic, logical, shift, or move instruction in the original test, EDDI-V creates a corresponding duplicate instruction that performs the same operation, but on the registers and memory reserved for duplicate instructions. The duplicated instructions execute in the same order as the original instructions. The EDDI-V transformation also inserts periodic check instructions (called ***Normal checks***) that compare the results of the original instructions against those of the duplicated instructions.

For every duplicated load instruction, an additional ***Load check*** instruction is inserted immediately after (before the loaded values are used by any other instructions) to check that the value loaded by the original instruction matches the value loaded by the corresponding duplicated instruction. Similarly, for store instructions, a ***Store check*** instruction is inserted immediately before the original store instruction to check that the value about to be stored matches the value about to be stored by the duplicated instruction. Each check instruction is of the form:

CMP Ra, Ra'

where *Ra* and *Ra'* are an original and (corresponding) duplicate register respectively. A mismatch in any check instruction indicates an error. In order to minimize any intrusiveness that might prevent bug detection by QED, insertion of the duplicated instructions and the check instructions is controlled by the parameters *Inst_min* and

Inst_max, the minimum and maximum number of instructions from the original test that must execute before any duplicated or check instructions execute.

B. PLC

PLC targets bugs inside uncore components by frequently and proactively performing loads from memory (through uncore components) and checking the values loaded. PLC first transforms an original test into an EDDI-V-transformed QED test. Next, PLC inserts Proactive Load and Check operations (*PLC operations*) throughout the transformed test, which runs on all cores and all threads. Each PLC operation checks the values in memory for a selected set of variables. For each selected variable, a PLC operation loads the value from the memory reserved for original instructions (address A) and then loads the value from the corresponding memory reserved for duplicated instructions (address A'). Any mismatch indicates an error. An example of a PLC operation for a single variable is shown below. Here, CMP Ra, Ra' is referred to as a *PLC check*. In a PLC operation, a lock is used if the variable is shared between multiple cores / threads or if there are sources of non-determinism in the system (e.g., due to interrupts, I/O, or OS functionalities such as context switches).

```
LOCK(A);
LOCK(A');
Ra  = LD(A)
Ra' = LD(A')
UNLOCK(A');
UNLOCK(A);
CMP Ra, Ra' // compare & detect error
```

A PLC operation checks all the variables selected for PLC. Various PLC strategies are discussed in [Lin 12, 14, 15].

6.4 Solving for QED-Compatible Bug Trace Using Bounded Model Checking

Both EDDI-V and PLC QED tests provide very succinct properties to check using check instructions of the form[3]:

CMP Ra, Ra'

For PLC checks and Load checks, *Ra* and *Ra'* hold values loaded from uncore components; for Normal checks and Store checks, *Ra* and *Ra'* hold the results of computations executed on the cores. An error is detected when the two registers are not equal. Thus, we use BMC to find counter-examples to properties of the form:

Ra==Ra'

where *Ra* is an original register and *Ra'* is the corresponding duplicated register. However, without additional constraints, the BMC engine will find trivial counter-examples that do not correspond to real bugs. For example, consider the following sequence of instructions,

MOV Ra ← 1
MOV Ra' ← 2
CMP Ra, Ra'

[3] This chapter presents Symbolic QED using the EDDI-V and PLC QED techniques. Symbolic QED using the CFCSS-V and CFTSS-V QED techniques are presented in Appendix 1.

In this case, Ra≠Ra' but it is not due to a bug. In order to avoid such situations, we require that counter-examples must be *QED-compatible*. We define a *QED-compatible* trace as a sequence of inputs with the following properties:

1. Inputs must be valid instructions (specifications of valid instructions can be directly obtained from the Instruction Set Architecture (ISA) of the processor cores);
2. The registers and memory space are divided into two halves, one for "original" instructions and one for "duplicated" instructions. For every instruction (excluding control-flow changing instructions) that operates on the registers and memory space allocated for the original instructions, there exists a corresponding duplicated instruction that performs the same operation but operates on the registers and memory space allocated for the duplicated instructions.
3. The sequence of original instructions and the sequence of duplicated instructions must execute in the same order.
4. The comparison (i.e., the property checked by the BMC tool) between an original register *R* and its corresponding register *R'* occurs only if the original and duplicate instructions are in sync, i.e. for each original instruction that has been executed, its corresponding duplicate instruction has also been executed.

6.5 QED Module for Creating QED-Compatible Bug Trace

Ensuring that only QED-compatible bug traces are considered by BMC requires constraining the inputs to the design. We accomplish this by adding a new QED module to the fetch stage of *each* processor core during BMC. Note that, although the QED module is added to the processor cores, Symbolic QED is effective not only for bugs inside processor cores, bugs inside uncore components, as well as bugs related to power management features (as demonstrated in Sec. 6.9). **The QED module is only used within the BMC tool and is not added to the manufactured IC.** The QED module only needs to be designed once for a given ISA, and made available as a "library component" for use during validation. The design of a QED module is simple, and can be tested in only a few minutes (see Sec. 6.9).

The *QED module* automatically transforms a sequence of original instructions into a QED-compatible sequence. Any control-flow altering instruction determines the end of the "sequence of original instructions."[4] The QED module only requires that this sequence is made up of valid instructions and that they read from or write to only the registers and memory allocated for the original instructions (conditions that can be

[4] For future work, it is possible to use a pseudo instruction "DUPLICATE" to determine the end of the "sequence of original instructions. This instruction is not executed by the processor core (i.e., it behaves as a NOP), and is used only to signal to the QED module that it has reached the end of a sequence of original instructions. This "DUPLICATE" instruction allows Symbolic QED to search through sequences of instructions that may not be possible with the existing approach described in this section. For example, currently, it is not possible for the QED module to create a sequence of instructions with an odd number of instructions between two control-flow altering instructions. For example, it is not possible to create the sequence {BRANCH, ADD, BRANCH}, since the QED module will modify this sequence of instructions into {BRANCH, ADD, ADD, BRANCH} (i.e., it will created a duplicated version of the ADD instruction. However, with the "DUPLICATE" instruction, it will be possible to create a sequence of instructions with an odd number of instructions between two control-flow altering instructions. For example, {BRANCH, ADD, BRANCH, DUPLICATE} will be transformed into {BRANCH, ADD, BRANCH, BRANCH, ADD, BRANCH}.

specified directly to the BMC tool). The sequence of original instructions is first executed unmodified (up to but not including the control-flow instruction), and the sequence of original instructions is committed. Then it is executed a second time; instead of using the original registers and memory, the instructions are modified to use the registers and memory allocated for the duplicated instructions. Since duplication is triggered only by a control-flow instruction, the QED module does not use a fixed value for *Inst_min* and *Inst_max*. Instead, (by design) the BMC tool will consider counter-examples (in this case, sequences of original instructions), starting with smaller sequences and then moving to longer sequences [Clarke 01]. This makes it possible for the BMC tool to **implicitly** (and **simultaneously**) search through a wide variety of instruction sequences of increasing lengths in order to find a bug trace. After the second execution, a signal is asserted to indicate that the original and corresponding duplicated registers should contain the same values under bug-free situations, i.e., the BMC tool should check the property *Ra == Ra'*.

Note that because the BMC tool can choose a wide variety of instructions as input to the QED module (including loads and stores), it can effectively create checks that could be generated by a QED transformation, including Normal, Load, Store, and PLC checks. It should also be noted that a PLC check generated by the QED module does not require locks. Locks are not needed in this case because: (i) we ensure that the QED modules for each core are synchronized: they all start executing duplicated instructions

on the same clock cycle;[5] and (ii) the behavior of the design during BMC is deterministic (i.e., there are no source non-determinism in the system e.g., due to interrupts, I/O, or OS functionalities such as context switches). Thus, the original and the duplicate sequences of instructions must compute the same results unless there is a bug.

To see why locks are not needed, we can consider a PLC on a variable X as composed of 4 events:

1) Store to the variable X by processor core A,

2) Load from the variable X by a processor core B,

3) Store to the corresponding duplicated variable X' by processor core A, and,

4) Load from the corresponding duplicated variable X' by processor core B.

Note that, multiple processor cores may load from X and X'. In order to guarantee there are no false fails, we need to ensure either one of two situations occur:

Situation 1: If event 1 occurs *before* event 2, then event 3 must occur *before* event 4;

or

Situation 2: If event 1 occurs *after* event 2, then event 3 must occur *after* event 4.

This means that event 3 and event 4 occur in the same order as that of event 1 and event 2. Figure 6.2 illustrates these two situations.

[5] When executing original instructions, as soon as some QED module encounters a control-flow instruction, all QED modules switch to *DUP_MODE* (details later), indicating that no new original instructions should be started. Then, once the original instructions on all cores have committed, the duplicated instructions begin executing on all cores simultaneously.

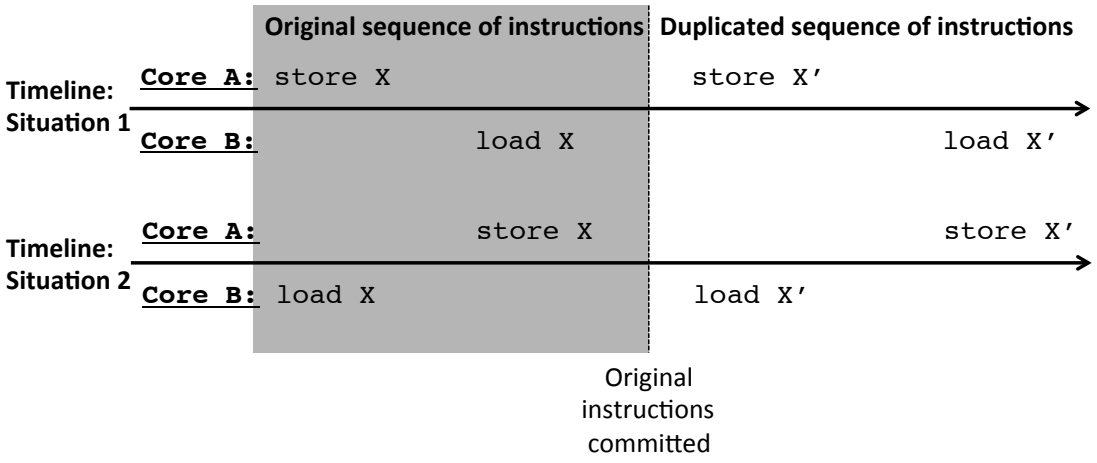


Figure 6.2 A timeline illustrating situations where PLC on variable X does not result in false fail.

Symbolic QED guarantees that only situation 1 or situation 2 will occur. To see why, first, we note that the QED module guarantees that the duplicated sequence of instructions always executes in the same order as that of the original sequence of instructions[6]. Therefore, the only way that event 3 and event 4 (i.e., store to duplicated variable X' and load form duplicated variable X') will occur in different order than that of event 1 and event 2 is if there are delays in executing the duplicated sequence of instructions (by either processor core A or processor core B). For example, in situation 1, if processor core A delays executing the store to the duplicated variable X' until after the load from duplicated variable X' on processor core B. However, there cannot be any delays because:

[6] This is because (as explained later) for the duplicated execution, the QED module modifies the operands of the original sequence of instructions such that they operate on registers and memory reserved for the duplicated sequence of instructions (the order of the instructions are preserved).

1) All of the processor cores start executing the duplicated sequence of instructions at the same clock cycle. This is because we wait until the original instructions are committed before we execute the duplicated instructions on the same clock cycle. Since the original instructions have committed, there are no instructions in the pipeline that would delay the execution of the duplicated instructions, and therefore, the duplicated instructions are guaranteed to start executing on the same clock cycle.

2) The execution of the instructions is deterministic. Therefore all instructions will complete in a deterministic amount of time (i.e., there are no delays, for example, due to context switches).

As a result, Symbolic QED guarantees that PLC will not result in false fails even without locks. This is effective even if there are multiple processor cores loading from variables X and X'. This is because we ensure that all processor cores start executing the duplicated sequence of instructions on the same clock cycle. As a result, the order for which event 3 and event 4 occur will always be the same as the order for which event 1 and event 2 occurred.

Note that this method of executing the original and duplicated sequences of instructions still allows the BMC tool to choose a wide variety of instruction in order to activate bugs. As a result, this does not significantly impact the ability of Symbolic QED to detect bugs in general. This is empirically demonstrated in Section 6.9 where Symbolic QED is effective for a wide variety of difficult logic bug scenarios. However, there may be situations where requiring all processor cores to execute the duplicated sequences of

instructions on the same clock cycle will miss a bug. Therefore, for future work, one may want to allow the processor cores to execute duplicated sequences of instructions on arbitrary clock cycles. However, one must be careful about loading from shared variables in order to prevent false fails. For example, the QED module may need to insert locks for PLC.

Figure 6.3 shows how the QED module integrates with the fetch unit. The pseudo code of the QED module is shown in Figure 6.4. The inputs to the QED module are: 1) *enable*, which disables the QED module if 0 (this signal can set by the validation engineers to disable the QED module); 2) *instruction_in*, which is the instruction from the fetch unit to be executed by the processor core; 3) *target_address*, which contains the address of the next instruction to execute by the processor core; and 4) *committed*, which is a signal from the processor core to indicate if the instruction fetched has been committed (i.e., the result written to register or memory).

The outputs from the QED module are: 1) *PC*, which is the address of the next instruction to fetch; 2) *PC_override*, which determines if the processor core should use the *PC* from the QED module or the PC from the fetch unit; 3) *instruction_out*, which is the modified instruction computed by the QED module; 4) *instruction_override*, which determines whether the processor core should use the modified instruction from the QED module or the instruction from the fetch unit; and 5) *qed_ready* signals when both original and duplicated registers should have the same values (under bug-free conditions).

Initially, *qed_ready* is set to false; it is only set to true when both original and duplicated instructions have executed and committed (i.e., committed is true).

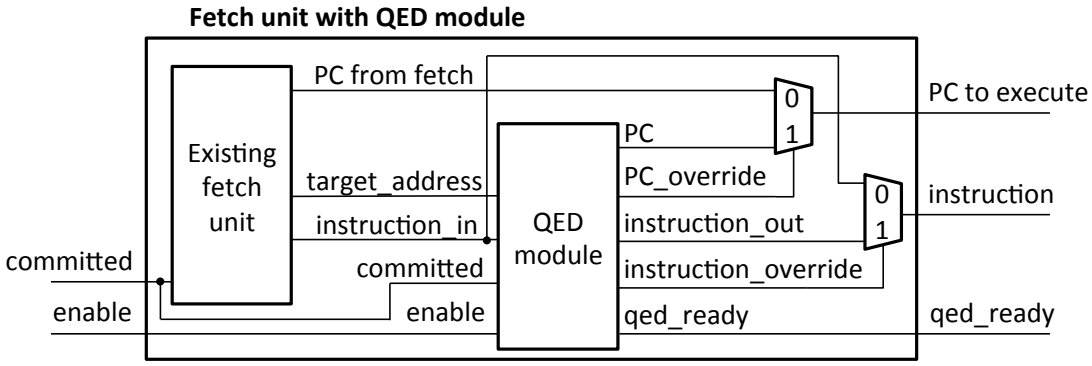


Figure 6.3 The QED module interfaced to the existing fetch unit.

The QED module has internal variables: 1) *current_mode*, which tracks if the QED module is executing original instructions (*ORIG_MODE*), or executing duplicated instructions (*DUP_MODE*), or if the BMC tool should perform a check (*CHECK_MODE*), this variable is shared by all of the QED modules in the design so that all QED modules are in the same mode; 2) *rewind_address*, which holds the address of the first instruction in the sequence of original instructions, (initialized to the PC obtained from the initial state in Sec. III.C); 3) *PC_override_i* and 4) *instruction_override_i*, which are internal versions of *PC_override* and *instruction*_override (the only difference is that when the *enable* is set to 0, then both *PC_override* and *instruction_override* are also set to 0, disabling the QED module).

The QED module starts in *ORIG_MODE*. When a control-flow instruction is fetched, the QED module checks the *committed* signal to wait for all of the original

instructions to commit before it loads the address stored in *rewind_address* into *PC*, and sets *PC_override_i* to 1 (if *enable* is 1, *PC_override* is also set to 1), and switches to *DUP_MODE*. The processor core then re-executes instructions starting from the address stored in *rewind_address*. In *DUP_MODE*, the duplicated instruction is output on *instruction_out*, and *instruction_override_i* is set to 1 so the core executes the duplicated instruction instead of the original instruction from the fetch unit. After all the duplicated instructions finish execution and commits (which is determined by observing the *committed* signal), the corresponding registers should be equal. At this point the QED module switches to *CHECK_MODE*. Once the results are written to registers (the *committed* signal from the processor core is 1), *qed_ready* is set to true. This time, the processor will execute the control-flow instruction, and the QED module will store the address of the next instruction to execute in *rewind_address* and then return to *ORIG_MODE*.

```
INPUT:  enable, instruction_in, target_address, committed
OUTPUT: PC, PC_override, instruction_out, instruction_override,
qed_ready
// initialization
current_mode ← ORIG_MODE;   qed_ready ← false;
PC_override_i ← 0;              instruction_override_i ← 0;
rewind_address ← PC obtained from initial state (Sec. III.C);
// end initialization
PC_override ←  enable ? 0 : PC_override_i;
instruction_override ← enable ? 0 : instruction_override_i;
if current_mode == CHECK_MODE, then
  if (enable == 1) and (committed == 1), then
    qed_ready ← true;    current_mode  ← ORIG_MODE;
   rewind_address ← target_address;
  end if
end if
if current_mode == ORIG_MODE, then
  qed_ready ← false;
  if is_control_flow_instruction(instruction_in), then
   wait_instructions_committed;
    current_mode ← DUP_MODE;
    PC ← rewind_address;    PC_override_i ← 1;
  end if
end if
if current_mode == DUP_MODE, then
  qed_ready ← false;
  if is_control_flow_instruction(instruction_in), then
    wait_instructions_committed;
    current_mode ← CHECK_MODE;
  else
    if not_control_flow_instruction(instruction_in), then
      instruction_out = create_duplicated_version(instruction_in);
      instruction_override_i ← 1;
    end if
  end if
end if
```

Figure 6.4 Pseudo code for the QED module.

An example of the transformation performed by the QED module is shown in Figure 6.5. This example also shows how PLC is performed: the LOAD(A) instruction is transformed by the QED module into LOAD(A') during the second execution. Comparing the registers at the end (which is done by the BMC tool) is then equivalent to

doing a PLC check on variables A and A'. Also note that both cores 1 and 2 start executing duplicated instructions when core 1 reaches the branch instruction. The initialization of memory is discussed in Sec. 6.6.

```
Core 1                          Core 2
A  = STORE(R1)                  R2 = R3 – R4
R2 = R3 + R4                    R1 = LOAD(A)
R5 = LOAD(A)                    R5 = LOAD(B)
BRANCH label
```

(a)

```
Core 1                          Core 2
A  = STORE(R1)                  R2 = R3 – R4
R2 = R3 + R4                    R1 = LOAD(A)   // PLC load
R5 = LOAD(A)  // PLC load       R5 = LOAD(B)
A' = STORE(R17)                 R18 = R19 – R20
R18= R19 + R20                  R17 = LOAD(A') // PLC Load
R21= LOAD(A') // PLC load       R21 = LOAD(B')
BRANCH label
```

(b)

Figure 6.5 Example of QED transformation by the QED module. (a) A sequence of original instructions on core 1 and core 2, and (b) the actual transformed instructions executed by the cores.

6.6 Initial State for Bounded Model Checking

The approach outlined above ensures that only QED-compatible traces are considered by BMC. However, the initial state for the BMC run must be a *QED-consistent* state, in which the value of each register in the processor core and memory location allocated for original instructions must match the corresponding register or memory location for duplicated instructions. This is to ensure that no false counter-examples are generated. One approach would be to start the processor from its reset state. However, the reset state may not be QED-consistent (or it may be difficult to confirm

whether it is). Some designs also go through a reset sequence that may span several clock cycles, making the BMC problem more difficult. For example, for OpenSPARC T2, only one processor core is active after a reset, and the system executes a sequence of initialization instructions (approximately 600 clock cycles long) to activate other processor cores in the system.

It is advantageous to start from a QED-consistent state after the system has executed the reset sequence (if any) to improve the runtime of BMC (also demonstrated by results in Sec. IV). A simple way to obtain a QED-consistent state is to run "some" QED test (independent of specific tests for bug detection and debug) in simulation, and stop immediately after QED checks have compared all of the register and memory values (this ensures that each "original" register or memory location has the same value as its corresponding "duplicate" register or memory location). This can be accomplished with a simple (short) test that just writes to the original and corresponding duplicated registers and memory locations, and checks them to ensure that they are in a QED-consistent state. The register values (including the PC) and memory values are read out of the simulator and then used to set the register values, PC, and memory values of the design when preparing to run BMC. If the design contains multiple processor cores, the processor cores can be simulated together. Alternatively, each core can be simulated independently and the results merged together to set up the BMC run. In this case, some care must be taken to ensure that the values in shared memory locations are the same at the end of each simulation (e.g. by running the same test on each core). One can obtain these values using

ultra-fast simulators (at a higher level of abstraction than RTL) that can simulate large designs with thousands of processor cores [Sanchez 13]. Thus, this initialization step does not affect the scalability of Symbolic QED.

6.7 Finding Counter-Example Using Bounded Model Checking

After inserting the QED module and setting the initial state, we use BMC to find a counter-example to the following property:

$$qed\_ready \rightarrow \bigwedge_{a \in \{0..\frac{n}{2}-1\}} Ra == Ra' \ ,$$

where n is the number of registers defined by the ISA. Here, (for $a \in \{0..n/2-1\}$), Ra corresponds to a register allocated for original instructions and Ra' corresponds to a register allocated for duplicated instructions. The property is the same for both EDDI-V and PLC, since as shown in Sec. 6.3, both EDDI-V and PLC checks an original register against the corresponding duplicate register. This is because for EDDI-V the register contains the result of an instruction or the result of a load instruction, and for PLC the register contains the result of a load instruction. We allow the instructions chosen by BMC to include load and store instructions, enabling our approach to activate and detect bugs in uncore components as well as those in processor cores.

6.8 Handling Large Designs

A state-of-the-art commercial BMC tool may not be able to load a complete SoC (e.g., this is the case for OpenSPARC T2). Here, we discuss two techniques for handling

such large designs that do not require any additional hardware. Design reduction techniques are important not only for handling large design, but also for better bug localization.

A. Bugs Inside Processor Cores vs. Outside Processor Cores

If a (standard, not symbolic) QED test fails either a Normal check or a Store check, we can immediately deduce that the bug is inside the processor core where the check failed[7]. This is because, by design, Normal and Store checks catch any incorrect value produced by a processor core before it leaves the processor core and propagates to the uncore components or to other processor cores. Thus, we just need to perform BMC on the single processor core where the check failed in order to find a bug trace. If the test fails at a Load check or a PLC check, we cannot immediately infer where the bug is. For these cases, we consider the Partial Instantiation technique, to simplify the design to be analyzed by BMC.

B. Partial Instantiation

Partial instantiation works through two design reduction techniques.

Technique 1 takes all components with multiple instances and repeatedly reduces their count by half until there is only 1 left. For example, in a multi-core SoC, the processor cores are removed from the design until there is only 1 processor core left.

[7] The entire test must be transformed by QED for this to work. If some QED checks are left out then this cannot be guaranteed. For example, if some of the Normal checks and Store checks are omitted, an error caused by a bug inside the processor core may propagate to an uncore component.

Technique 2 removes a module as long as its removal does not divide the design into two disconnected components. For example, if a design has a processor core connected to a cache through a crossbar, the crossbar is not removed (without also removing the cache). This is because if the crossbar is removed, the processor core is disconnected from the cache.

All possible combinations and repetitions of the two techniques are considered when producing candidates for analysis. Since we find bug traces in the form of instructions that execute on processor cores, each analyzed design must contain at least one processor core.

Figure 6.6 shows the steps for this approach. Once the full set of simplified (partially instantiated) designs is created, they are can be analyzed using the BMC tool independently (in parallel). An example is presented below.

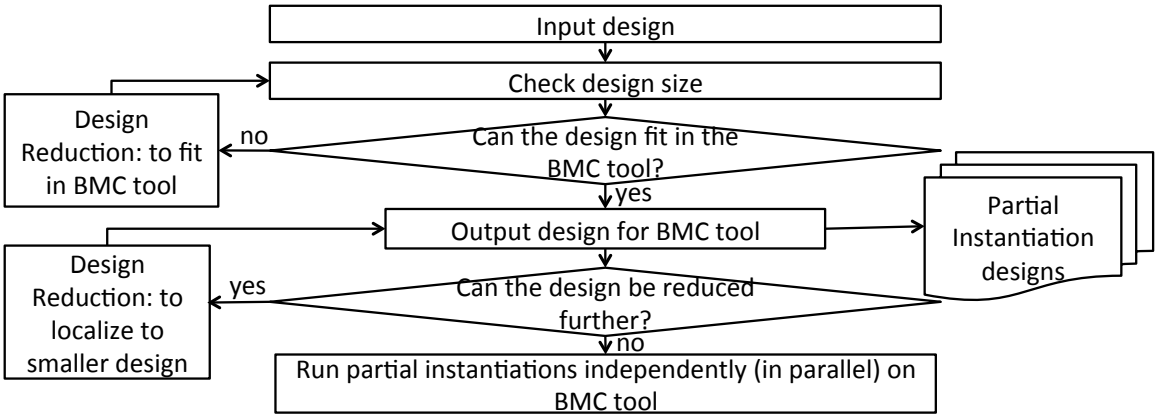


Figure 6.6 The partial instantiation approach for design reduction.

As an example, consider the OpenSPARC T2 with 8 processor cores, 1 crossbar, 8 banks of cache, 4 memory controllers, and I/O controller (Figure 6.7). Consider the

OpenSPARC T2 design with 8 processor cores, 1 crossbar, 8 banks of shared L2 cache, 4 memory controllers, and an I/O controller (Figure 6.7). This entire design is too big to be analyzed by the BMC tool, and it is not saved as a partial instance. One possibility is to remove the I/O controller, resulting in 8 processor cores, 1 crossbar, 8 banks of cache, and 4 memory controllers; this is still too big for the BMC tool, and it is not saved as a partial instance. Alternatively, components with multiple instances (e.g., the cores, caches, and memory controllers) can be halved, reducing the design to 4 processor cores, 1 crossbar, 4 banks of cache, 2 memory controllers, and the I/O controller. This still does not fit in the BMC tool, and so again, it is not saved as a partial instance. At this point, we can take either of our two reduced designs as candidates for further reduction. Let us consider the second one. The crossbar is not removed, as it would disconnect the processor cores from the other components. Suppose instead that we apply Technique 1 again. This reduces the design to 2 processor cores, 1 crossbar, 2 banks of cache, 1 memory controller, and the I/O controller. This design still does not fit. Next, either the I/O controller or the memory controller can be removed by applying technique 2. By removing the I/O controller, we are left with 2 processor cores, 1 crossbar, 2 banks of cache, and 1 memory controller. This does fit in the BMC tool and so the configuration is saved. Alternatively, by removing the memory controller, we are left with 2 processor cores, 1 crossbar, 2 banks of cache, and the I/O controller, which also fits and is saved. Now, even though at this point we have two candidate configurations for BMC, we continue to apply design reduction techniques to generate more partial instances. The

reason for this is for better localization: if the BMC can find a bug trace in a smaller configuration, then this indicates that the components removed by the design reduction techniques are not necessary for activating and detecting the bug. Continuing with the reduction, by applying technique 1, the number of cores and caches can be reduced, resulting in 1 processor core, 1 crossbar, 1 bank of cache, 1 memory controller, and the I/O controller. Further reductions result in smaller and smaller subsets of the design, each of which fits in the BMC tool and is saved. When no more reductions are possible (i.e., when the design is reduced down to just a single core), all of the saved designs are analyzed independently (in parallel) by the BMC tool.

6.9 Results

We demonstrate the effectiveness of Symbolic QED using the OpenSPARC T2 SoC [OpenSPARC] (Figure 6.7), which is the open-source version of the UltraSPARC T2, a 500-million-transistor SoC with 8 processor cores (64 hardware threads), a private L1 cache, 8 banks of shared L2 cache, 4 memory controllers, a crossbar-based interconnect, and various I/O controllers. We simulated logic bug scenarios from Chapter 3 on the OpenSPARC T2 SoC. These represent a wide variety of "difficult" bug scenarios extracted from various commercial multicore SoCs. They are considered difficult because they took a long time (days to weeks) to localize. The bug scenarios include bugs in the processor cores, bugs in the uncore components, and bugs related to power management. The bug scenarios were simulated by modifying the RTL of the

OpenSPARC T2 SoC design such that for each bug scenario, if the bug activation criterion is satisfied, the bug effect is simulated.

The RTL of the OpenSPARC T2 SoC was modified to incorporate these bug scenarios. For the 80 bug scenarios resulting from Table 3.1 and Table 3.2, we set the bug scenario parameter *X* to 2 clock cycles and bug scenario parameter *Y* to 2 clock cycles. The details of *X* and *Y* are in Chapter 3; in particular note that smaller values for *X* and *Y* means that bugs are more difficult to activate, and thus detect. For example, consider bug activation criterion 1 from Table 3.1: "Two stores in X clock cycles to different cache lines." Also consider a sequence of instructions of the form:

```
STORE [0x00] ← R1
ADD R1, R2, R3
STORE [0x10] ← R2
```

In this , each instruction takes 1 clock cycle to execute, and [0x00] and [0x10] correspond to different cache lines. This sequence of instructions satisfies bug activation criterion 1 from Table 3.1 when $X = 3$, but is unable to satisfy bug activation criterion 1 from Table 3.1 when $X = 2$. Hence, smaller values of *X* and *Y* result in bug scenarios that are more difficult to activate and detect.

We also did not vary the *X* and *Y* parameters. This is because if a bug trace can activate and detect a bug scenario with small *X* and *Y* parameters, the same bug trace can also activate and detect the same bug scenario with greater *X* and *Y* parameters. For example, the following bug trace, which satisfies bug activation criterion 1 from Table

3.1 with (X=2): “two stores in 2 clock cycles to different cache lines,” also satisfies the same bug activation criterion with (X=3): “Two stores in 3 clock cycles to different cache lines.”

```
STORE [0x00] ← R1
STORE [0x10] ← R2
```

For the 12 power management bug scenarios in Sec. 3.1, the activation criterion is set to a sequence of 5 instructions selected from the original test, executed on a designated processor core. This is to emulate a power management controller. This is to emulate a power management controller, which puts the system into power-saving state when it executes a specific sequence of instructions. When inserting bugs, if a bug is inserted into a component, the bug is in all instances of the component.

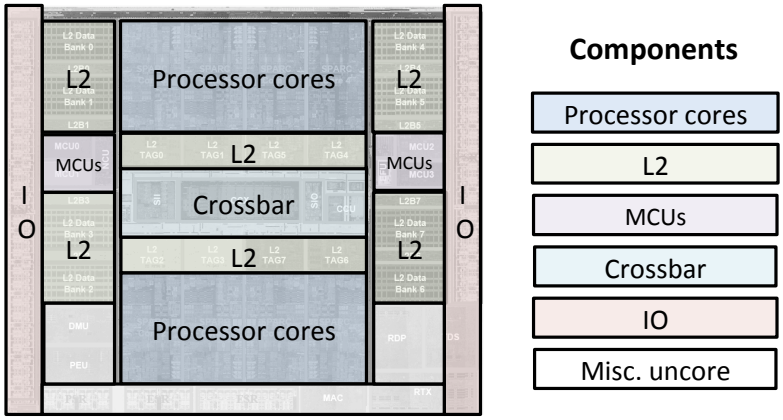


Figure 6.7 OpenSPARC T2 SoC diagram.

For BMC, we used the Questa Formal tool (version 10.2c_3) from Mentor Graphics [Questa] on an AMD Opteron 6438 with 128GB of RAM. We used the EDDI-

V and the PLC (Sec. 4.1 and Sec. 4.2) QED transformations to transform an 8-thread version of the FFT test (from SPLASH-2 [Woo 95]) and an in-house parallelized 8-thread version of the matrix multiplication test (MMULT) into QED tests to detect bugs. The *Inst_min* and *Inst_max* QED transformation parameters were set to 100, a setting which typically allows bugs to be detected within a few hundred clock cycles (as shown in Chapter 4).[8] Trying additional tests (beyond FFT and MMULT) was deemed unnecessary because both tests (after QED transformation) were able to detect all bugs (and the BMC step in Symbolic QED is independent of the QED tests that detect the bug).

We added the QED module described in Sec. 6.5 to the RTL of the fetch unit in the OpenSPARC T2 processor core. The resulting fetch unit with the QED module was tested in Questa Formal to ensure it correctly transforms a sequence of original instructions into a QED-compatible bug trace. The testing process for 50 sequences of original instructions of varying length (1 to 10 instructions long) took approximately 1 minute of runtime. Moreover, we simulated all of the bug traces produced by Symbolic QED (which depends on the QED module) to ensure that the bug traces indeed activate and detect the corresponding bugs. No additional hardware (e.g., trace buffers) was added to the design.

The results are summarized in Table 6.1. The Original (No QED) column shows results from running the original validation tests (FFT or MMULT) using end result

[8] These *Inst_min* and *Inst_max* parameters do not correspond to the *Inst_min* and *Inst_max* values for the bug traces found by Symbolic QED shown later; they are only used to create the QED tests for detecting bugs.

checks to check the results of the test against pre-computed, known correct results. The QED column shows results from running the same tests after QED transformation. Note that unlike Symbolic QED, both the Original (No QED) and the QED tests (without the analysis techniques discussed in Sec. 6.8A.) are only able to report the existence of a bug; they cannot localize a bug (i.e., determine if the bug is in the processor core, in any of the uncore components, or is caused by interactions between the components), nor can they determine very precisely when a bug is activated. The table is categorized into processor core bugs, uncore bugs (here we include bugs that are inside uncore components as well as in the interface between processor cores and uncore components) from Chapter 3, and power management bugs from Sec. 3.1. Each entry contains two sets of numbers, the top set contains the results obtained from the FFT test, and the bottom set contains results obtained from MMULT.

In Table 6.1, "Bug trace length (instructions)" shows the [minimum, average, maximum] number of instructions in the bug trace. "Bug trace length (cycles)" represents the [minimum, average, maximum] number of clock cycles required to execute the bug trace. The two numbers are different because the number of cycles per instruction (CPI) is not 1 for all instructions (for example, a load or store instruction may take multiple clock cycles to execute). For Symbolic QED, the reported length for bug traces corresponds to the number of instructions in the trace found by the BMC tool (not including duplicated instructions created by the QED modules). For bugs that are only found by executing instructions on multiple processor cores, the number of instructions

for each core may be different. For example, one core could have a bug trace that is 3 instructions long, while another core has a bug trace that is 1 instruction long. We report the length of the longest bug trace in such situations (3 in this example) because all cores must complete to activate and detect the bug (and the cores execute the instructions in parallel).

Table 6.1 Results comparing original tests (No QED), QED tests, and Symbolic QED on FFT (top values) and MMULT (bottom values). For bug traces, we report the [minimum, average, maximum] length in instructions and clock cycles. We also report [minimum, average, maximum] BMC runtimes.

		Original (No QED)	QED	**Symbolic QED**
Processor core only bugs	Bug trace length (instructions)	[643,551k,4.9M] [12k,534k,2.3M]	[324,57k,233k]† [421,67k,321k]†	**[3, 3, 3]** **[3, 3, 3]**
	Bug trace length (clock cycles)	[842,572k,5.1M] [15k,544k,2.5M]	[367,66k,265k] † [522,69k,272k] †	**[13, 15, 16]** **[13, 15, 16]**
	Coverage	50.0% 54.2%	100% 100%	**100%** **100%**
	BMC runtime (minutes)	N/A	N/A	**[22, 46, 90]** **[22, 47, 89]**
	Bugs localized	0% 0%	0% 0%	**100%*** **100%***
Uncore bugs	Bug trace length (instructions)	[620,1.6M,9.8M] [1k,536k,2.5M]	[231,59k,232k] † [392,80k,421k] †	**[3, 4, 4]** **[3, 4, 4]**
	Bug trace length (clock cycles)	[722,1.9M,11M] [2k,550k,2.7M]	[292,72k,289k]† [442,95k,435k]†	**[14, 22, 29]** **[14, 22, 29]**
	Coverage	55.3% 57.1%	100% 100%	**100%** **100%**
	BMC runtime (minutes)	N/A	N/A	**[78,164,188]** **[76,163,190]**
	Bugs localized	0% 0%	0% 0%	**100%*** **100%***
Power management bugs	Bug trace length (instructions)	[1.5k,236k,495k] [963,213k,422k]	[10k,68k,302k]† [1k,47k,134k]†	**[5, 5, 5]** **[5, 5, 5]**
	Bug trace length (clock cycles)	[1.9k,251k,512k] [1.5k,220k,430k]	[13k,75k,319k]† [2k,49k,149k]†	**[17, 19, 22]** **[17, 19, 22]**
	Coverage	66.7% 66.7%	100% 100%	**100%** **100%**
	BMC runtime (minutes)	N/A	N/A	**[205,266,333]** **[206,264,335]**
	Bugs localized	0% 0%	0% 0%	**100%*** **100%***

* Symbolic QED localizes 100% of the bugs ***without*** using trace buffers.

† If trace buffers are used for QED, then the trace lengths in terms of instructions are: for FFT, [63,451,863] for processor core bugs, [29, 487,832] for uncore bugs, and [42, 297, 742] for power management bugs; and for MMULT, [44, 309, 874] for processor bugs, [32, 502, 884] for uncore bugs, and [67, 392, 742] for power management bugs. The trace lengths in terms of clock cycles are: for FFT, [82, 512, 922] for processor core bugs, [38, 532,930] for uncore bugs, and [66, 412, 912] for power management bugs; and for MMULT, [69,420,921] for processor core bugs, [58, 582, 944] for uncore bugs, and [79, 482, 801] for power management bugs. Other entries remain the same.

Observation 1: Symbolic QED **automatically** produces bug traces that are up to 6 orders of magnitude shorter than traditional post-silicon validation tests that rely on end result checks, and up to 5 orders of magnitude shorter than QED tests. The bug traces produced by Symbolic QED are very short (we confirmed their correctness using simulation). Furthermore, **Symbolic QED does not need trace buffers (or any additional hardware) to produce correct bug traces.** These are very difficult bugs that took many days or weeks of (manual) work to localize use traditional approaches (also evident by the long bug traces produced by traditional techniques). Short bug traces make debugging much easier. A more detailed visualization of the trace lengths for each bug scenario is presented in Figure 6.10.

In Table 6.1, "Coverage" is the percentage of the 92 bugs detected. Both Symbolic QED and QED detected all 92 bug scenarios, while the original tests detected only a little more than half of the bugs. This is because the original tests (No QED) may not contain the instructions needed to activate a bug, and even if they do, there may not be sufficient checks to detect it. In contrast, QED performs extensive checks to detect errors, and Symbolic QED searches through a wide variety of QED tests to find a sequence of instructions that will activate and detect the bug.

"BMC runtime" represents the [minimum, average, maximum] number of minutes it took for the BMC tool to find the bug traces. And "Bugs localized" represents the percentage of bugs localized. Note that both Original (No QED) and QED tests can only detect bugs, not localize bugs.

We did not include any results from running the BMC without our Symbolic QED technique for two reasons: (i) the full design does not load into the BMC tool; and (ii) even if it did, we would need properties to check to run BMC, and there is no clear way to create such properties (other than manual insertion which would be subjective and extremely time-consuming). Indeed, the Symbolic QED technique for expressing a generic property to check is one of our key contributions.

Observation 2: Symbolic QED correctly and automatically produces short counter-examples for all bugs in less than 7 hours, **without relying on any additional hardware**. Symbolic QED is effective for large designs such as the OpenSPARC T2, which are challenging when using traditional post-silicon techniques.

For Symbolic QED, all of the processor core bugs were detected by either a Normal check or a Store check. Thus (as described in Sec. 6.8A.) we are able to determine that the bug must be inside the processor cores. This was determined solely based on the QED checks, not because we knew which bugs were simulated. The BMC runtime reported for these bugs corresponds to a BMC run in which only the processor core was loaded.

For uncore and power management bugs, the partial instantiation technique (Sec. 6.8B.) was used. The BMC tool analyzed the partial instances in parallel. For the

OpenSPARC T2, there were 9 parallel BMC runs for each bug; each run corresponded to one of the following partial instances, which are ranked by size in descending order.[9]

1) 2 processor cores, 2 L2 cache banks, the I/O controller.

2) 2 processor cores, 2 L2 cache banks, 1 memory controller.

3) 2 processor cores, 2 L2 cache banks.

4) 1 processor core, 1 L2 cache bank, 1 memory controller, and the I/O controller.

5) 1 processor core, 1 L2 cache bank, and the I/O controller.

6) 1 processor core, 1 L2 cache bank, and 1 memory controller.

7) 1 processor core and the I/O controller.

8) 1 processor core, 1 L2 cache bank.

9) 1 processor core.

Recall that if a bug is in a component, it is in all instances of the component. For these bugs, the BMC runtime reported corresponds to the runtime of the smallest partial instance that produced a counter-example. Note that, this partial instance also provides us with a small candidate list of components that may contain the bug. For example, for a given bug, if both partial instances 6 and 8 produced a counter-example, then only the result from partial instance 8 was reported. This suggests that the additional components in partial instance 6 were not required for activating or detecting the bug. For example, while both partial instances 6 and 8 contain processor cores and caches, partial instance 8

[9] Partial instantiation 1 is the largest that will fit into the BMC tool; all designs also contain the crossbar that connects the components together.

does not have a memory controller. This suggests that the memory controller was not required to activate and detect this bug.

Observation 3: Symbolic QED correctly localizes bugs and provides a candidate list of components corresponding to possible locations of bugs in a design.

Figure 6.8 reports a breakdown of the bugs localized by Symbolic QED. Symbolic QED correctly localized all 92 bug scenarios. For 26.1% of the bugs, Symbolic QED localized the bugs to exactly 1 processor core; for 56.5% of the bugs, Symbolic QED localized the bug to 1 processor core, 1 L2 cache bank and the crossbar that connects the two; and for 17.4% of the bugs, Symbolic QED localized the bug to 2 processor cores, 2 L2 cache banks, and the crossbar that connects the components.

The BMC runtimes reported in Table 6.1 for Symbolic QED use the QED-consistent initial state constraint discussed in Sec. 6.6. The detailed runtimes for each bug are also presented in Figure 6.9. Figure 6.9 reports three runtimes for each bug: the runtime when starting from the state immediately after a reset sequence (which is QED-consistent in this case), the runtime when starting from a QED-consistent initial state obtained by running the FFT QED test and seeding BMC with the resulting register and memory values (Sec. 6.6), and the runtime when similarly seeding BMC after running MMULT. Results demonstrate that using a QED-consistent initial state obtained by running a QED test achieves up to 5X improvement in runtime compared to starting from the state after reset. Note also that no significant differences were observed between the results from using the FFT test and those using the MMULT test.

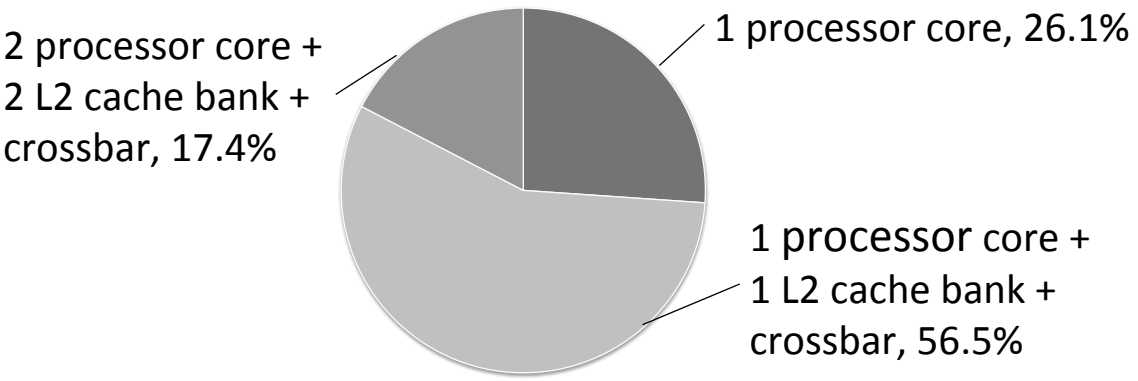


Figure 6.8 Graph showing the percentage breakdown (by list of candidate modules) of bugs localized by Symbolic QED. All 92 bugs were correctly localized.

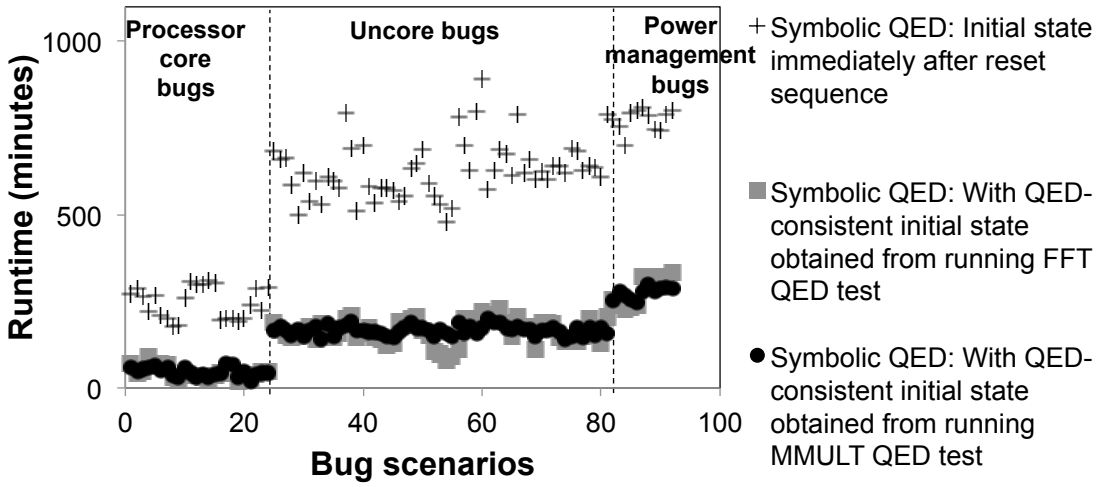


Figure 6.9 The BMC runtimes for Symbolic QED for each bug.

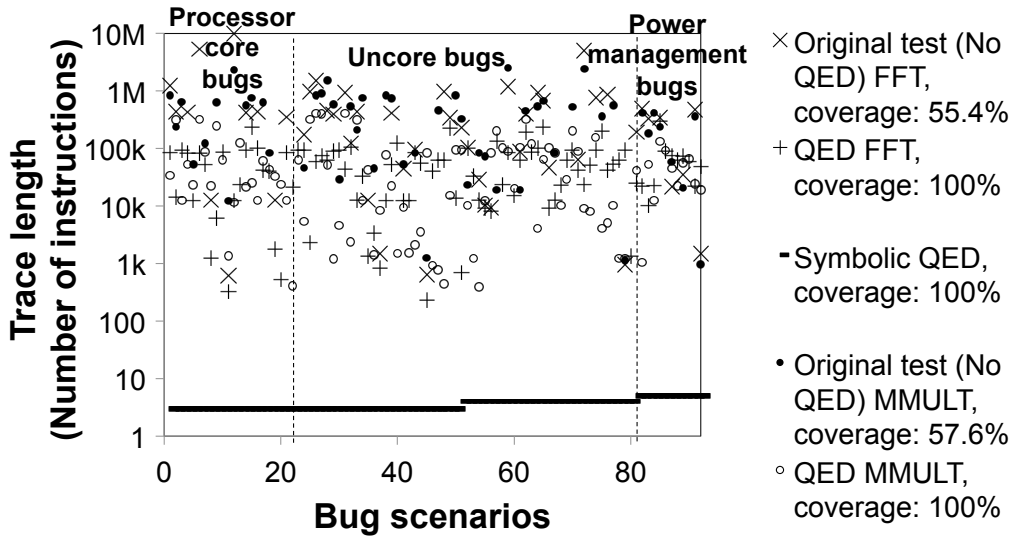


Figure 6.10 Trace length (in terms of number of instructions) for each bug.

While this section demonstrated the effectiveness of Symbolic QED on the OpenSPARC T2 SoC, Symbolic QED does not rely on any information about the specific implementation of the OpenSPARC T2. As a result, Symbolic QED is applicable to a wide variety of SoC designs.

6.10 Related Work

The Symbolic QED technique in this chapter mostly relies on the QED technique (Chapter 4) for creating post-silicon validation tests, but there are important differences. Unlike Symbolic QED, QED alone does not directly localize bugs at a fine level of hardware granularity. As shown in Sec. 6.9, the bug traces obtained by QED can be very long (up to 5 orders of magnitude longer when no trace buffers are used) compared to Symbolic QED. For bugs inside processor cores, Symbolic QED can potentially be further enhanced using techniques such as self-consistency checking [Jones 05].

However, [Jones 05] addresses only processor core bugs. Our experiences with bugs in commercial SoCs indicate that uncore components are also an important source of difficult bugs in SoCs [Lin 12, 14, 15].

The growing importance of post-silicon validation and debug has motivated recent publications on bug localization and generation of bug traces. IFRA and the related BLoG [Park 09, 10] techniques for post-silicon bug localization target processors only (and the published results target electrical bugs). The effectiveness of IFRA and BLoG for bugs inside uncore components is unclear. They also require manual efforts unlike Symbolic QED.

Many post-silicon bug localization approaches rely on trace buffers and assertions. Chapter 1 already discussed the inadequacy of these techniques. (Some of the heuristics for trace buffer insertion, e.g., restoration ratio and its derivatives only work for logic bugs since they use simulations to compute the logic values of signals that are not traced). In contrast, Symbolic QED does not require any trace buffers or design-specific assertions, and provides a very succinct and generic property to quickly detect and localize logic bugs.

BackSpace and its derivatives [De Paula 08, 11, 12] provide a concrete bug trace once an error is detected or the system crashes by using formal methods to stitch together multiple short traces (or system states) into a longer trace. Some BackSpace derivatives require failure reproduction, which, as discussed in Chapter 1 and in [De Paula 11, 12], is challenging due to Heisenbug effects [Gray 85]. nuTAB-BackSpace addresses some of

the failure reproduction challenges but requires design-specific "rewrite rules" to determine if two similar states are equivalent. These rewrite rules have to be manually crafted by designers and require designer intuition, which may be difficult for large designs. Furthermore, the bug traces found may be very long, and unlike Symbolic QED, these techniques cannot reduce the length of the bug traces. Moreover, techniques that solely rely on formal methods for bug localization (e.g., [De Paula 08, 11, 12, Zhu 11]) are not scalable to large designs such as the OpenSPARC T2. Some formal techniques require specific bugs models (e.g., [Zhu 11] which targets a specific model for electrical bugs) may not work for logic bugs, since it is very difficult to create bug models for all logic bugs [ITRS 09].

Approaches that rely on detailed RTL simulations to obtain the internal states of a design are not scalable for large designs because full system RTL-level simulation of large designs is extremely slow, less than 10 clock cycles per second [Schelle 10]. [DeOrio 11] presented a technique for post-silicon bug diagnosis, but it requires multiple detailed RTL simulations of the internal states of a design to guide the insertion of hardware structures for debugging. BuTraMin [Chang 09] is a pre-silicon technique for shortening the length of a bug trace. For use in post-silicon validation and debug of large designs, it will require massive simulations to capture logic values of all flip-flops in the system, which will be difficult. However, there might be opportunities to use such techniques after Symbolic QED localizes bugs and produces corresponding short bug traces (as demonstrated in this chapter).

6.11 Summary and Discussions

This chapter presented the Symbolic QED technique, a structured and automated approach that overcomes post-silicon validation and debug challenges. Symbolic QED automatically detects and localizes logic bugs in post-silicon validation and provides a list of candidate components that may contain the bugs. Symbolic QED produces bug traces that are up to 6 orders of magnitude shorter than traditional post-silicon validation tests that rely on end-result-checks, and up to 5 orders of magnitude shorter than QED tests. Symbolic QED is completely automated, does not require human intervention (manual efforts), and does not need any additional hardware.

Symbolic QED is both effective and practical, as demonstrated on the OpenSPARC T2, where it correctly localized difficult logic bug scenarios that occurred during post-silicon validation of various commercial multicore SoCs. These difficult bug scenarios originally took many days or weeks of (mostly manual) debug work to localize. Other formal techniques for debug may take days or fail completely for large designs such as the OpenSPARC T2. As demonstrated in this chapter, Symbolic QED is effective for bugs inside processor cores and uncore components, as well as bugs related to power-management features. Symbolic QED is applicable to any SoC design as long as it contains at least one programmable processor core (a generally valid assumption for existing SoCs [Foster 15]).

CHAPTER 7. CONCLUDING REMARKS

This dissertation presented the Quick Error Detection (QED) technique for post-silicon validation and debug. Results from multiple hardware platforms, as well as simulation results using realistic bug scenarios on a complex multi-core SoC, demonstrate that QED improves error detection latencies by up to 9 orders of magnitude and improves bug detection coverage by up to 4-fold. QED can be implemented entirely in software, in which case it is readily applicable to existing post-silicon validation flows. Alternatively, QED can utilize small hardware support to significantly improve the runtimes of QED tests (by up to 4 orders of magnitude) while preserving the improved error detection latency and improved bug detection coverage benefits of software-only QED. The hardware support has very small chip-level area overhead (less than 0.4% as demonstrated on the OpenSPARC T2 SoC), and negligible power and performance overhead. In this dissertation, we've demonstrated the effectiveness of QED for a wide variety of bugs and SoC components, including bugs inside processor cores, bug inside uncore components, bugs related to power management features, electrical bugs, as well as logic bugs. Furthermore, it has been demonstrated that QED is also effective for bugs inside hardware accelerators [Campbell 15].

Enabled by such short error detection latencies and high coverage, this dissertation also presented the Symbolic QED technique, which is a systematic and structure approach that automatically localizes logic bugs during post-silicon validation

and debug. Symbolic QED is a systematic and structure approach because it does not rely on any information about the specific implementation of the design in order to localize bugs. Symbolic QED is applicable to any SoC design as long as it contains at least one programmable processor core (a generally valid assumption for existing SoCs [Foster 15]). Results demonstrate that Symbolic QED automatically localizes logic bugs in only a few hours, compared to multiple days or even weeks of (manual) work required when using traditional post-silicon bug localization approaches. Symbolic QED is completely automated, does not require human intervention.

Based on the work in this dissertation, several opportunities exist to further enhance post-silicon validation and debug using QED. Examples include: 1) expand QED for system-level identification of failing ICs and for root-causing No-Trouble-Found (NTF) ICs [Conroy 05], 2) localize electrical bugs (Symbolic QED presented in this dissertation focused on logic bugs); 3) localize bugs in analog and mixed signal components of SoCs; 4) localize bugs in low-level system software (i.e., firmware) of SoCs and electronic systems, and 5) perform system-level bug localization.

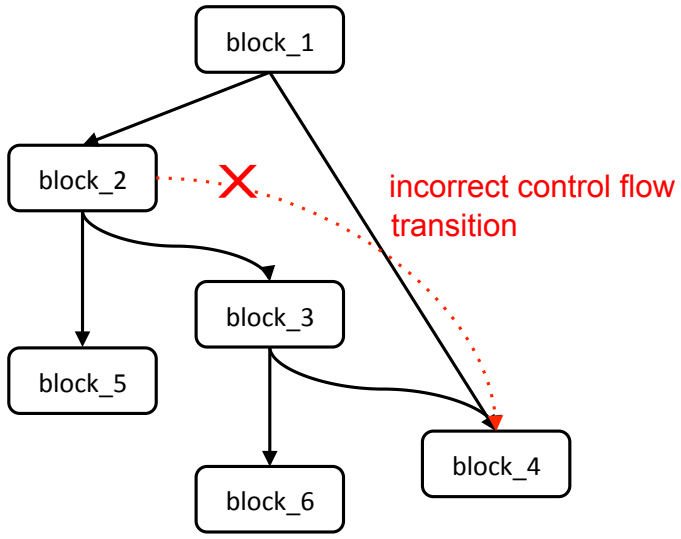
APPENDIX 1. SYMBOLIC QED FOR CFCSS-V AND CFTSS-V

The Symbolic QED technique presented in Chapter 6 focused on the EDDI-V and the PLC QED transformations. This appendix extends the Symbolic QED technique for the CFCSS-V and the CFTSS-V QED transformations, which target bugs that affect the control flows of processor cores. Results on the OpenSPARC T2 SoC, a 500-million-transistor design, demonstrate that Symbolic QED quickly finds very short bug traces for bugs that affect the control flows of processor cores. The Symbolic QED for the CFCSS-V and the CFTSS-V QED transformations can be combined with the Symbolic QED for the EDDI-V and the PLC QED transformations presented in Chapter 6.

A1.1 Symbolic QED for the CFCSS-V QED Transformation

The CFCSS-V technique checks the control flows of a test. First, we provide an overview of the CFCSS-V QED transformation. Consider a sequence of instructions shown in Figure A1.1a and the control flow graph constructed from this sequence of instructions shown in Figure A1.1b. A CFCSS-V check block is inserted at the beginning of each block of instructions. The pseudo code for the CFCSS-V check block for *block_4* (i.e., <*CFCSS-V check 4*>) is shown in Figure A1.2.

```
block_1:
    <CFCSS-V check 1>
    <instructions>
    branch_if_equal block_4
block_2:
    <CFCSS-V check 2>
    <instructions>
    branch_if_equal block_5
block_3:
    <CFCSS-V check 3>
    <instructions>
    branch_if_equal block_6
block_4:
    <CFCSS-V check 4>
    <instructions>
```



Instructions
(a)

Control flow graph
(b)

Figure A1.1 (a) An example sequence of instructions, and (b) the corresponding control flow graph. The dashed arrow represents an incorrect control flow transition from block_2 to block_4.

```
if ((signature == block_1_sigature) or (signature == block_3_signature))
then
  signature = block_4_signature // update signature
else
  ERROR DETECTED
end if
```

Figure A1.2 Pseudo code for <CFCSS-V check 4>.

In Figure A1.2, *block_1_signature*, *block_3_signature*, and *block_4_signature* are the unique software signatures assigned to *block_1*, *block_3*, and *block_4*, respectively. The variable *signature* holds the software signature of the last block of instructions executed. As shown in Figure A1.1, when the control flow transitions from one block of

instructions to another block, the CFCSS-V check block is executed first (before any other instructions in the block). In the case of *block_4*, the <*CFCSS-V check 4*> shown in Figure A1.2 is executed first. <*CFCSS-V check 4*> checks to see if the variable *signature,* which contains the software signature of the last block of instructions executed, is equal to either *block_1_signature* or *block_3_signature* (i.e., the unique software signatures for *block_1* and *block_3* respectively). This is because, as shown in Figure A1.1b, *block_1* and *block*_3 are the only two blocks of instructions that may execute immediately before *block_4*. If *signature* equals either *block_1_signature* or *block_3_signature*, then no control flow error is detected and *signature* is updated to equal the signature of *block_4* (i.e., *block_4_signature*). However, if *signature* does not equal either *block_1_signature* or *block_3_signature*, then a control flow error is detected. For example, as shown in Figure A1.1b when the control flow incorrectly transitions from *block_2* to *block_4*, *signature* would contain the software signature of *block_2* (i.e., *block_2_signature*). Since the software signatures are unique (Sec. 4.3), if *signature* ≠ *block_1_signature* and *signature* ≠ *block_3_signature*, it follows that an error is detected.

Now, we describe how to adapt CFCSS-V to Symbolic QED. We start by defining the property to be checked by the BMC tool. As shown above, the CFCSS-V QED transformation assigns unique software signatures to instructions / blocks of instructions. Symbolic QED for the CFCSS-V QED transformation uses the memory address of each instruction as its unique software signature. For example, consider the

following sequence of instructions. The memory address of each instruction is shown on the left (before the colon).

```
0x00000000: STORE [0x00] ← R1
0x00000004: ADD R1, R2, R3
0x00000008: BRANCH label
0x0000000C: SUBTRACT R4, R5, R6
```

In this example, the unique software signature of the STORE instruction is 0x00000000, the unique software signature of the ADD instruction is 0x00000004, the unique software signature of the BRANCH instruction is 0x00000008, and the unique software signature of the SUBTRACT instruction is 0x0000000C.

For CFCSS-V, the property to be checked by the BMC tool is:

```
(currently_executing_PC == ADDRESS_taken_branch) or
(currently_executing_PC == ADDRESS_not_taken_branch)
```

Where *currently_executing_PC* is the memory address (i.e., the PC or program counter) of the instruction that the processor core is currently executing (for in-order processor cores, this corresponds to the PC of the instruction in the execution stage, for out-of-order or speculative-execution processor cores, this is the PC of the most recently committed instruction). *ADDRESS_taken_branch* and *ADDRESS_not_taken_branch* correspond to the address of the first instruction in either the taken or the not taken branch of the last executed instruction.

If the last executed instruction is a conditional branch instruction, *ADDRESS_taken_branch* is the address of the first instruction in the taken branch, and *ADDRESS_not_taken_branch* is the address of the first instruction in the not taken branch (usually it is an increment of the PC). If the last executed instruction is a non-conditional branch instruction (e.g., branch always) *ADDRESS_taken_branch* is the target address of the branch instruction, and since non-conditional branch instructions do not have a not taken branch, *ADDRESS_not_taken_branch* is set to the same value as that of *ADDRESS_taken_branch.*

Alternatively, if the last executed instruction is not a control flow altering instruction (e.g., ALU instructions, load / store instructions) *ADDRESS_taken_branch* is set to the same value as that of *ADDRESS_not_taken_branch* (usually it is an increment of the PC), since there is no taken branch.

Unlike the EDDI-V and the PLC QED transformations, the CFCSS-V QED transformation does not duplicate instructions; as a result, Symbolic QED for the CFCSS-V transformation only requires the following input constraint:

> *Inputs must be valid instructions (specifications of valid instructions can be directly obtained from the Instruction Set Architecture (ISA) of the processor cores).*

This input constraint can be specified directly to the BMC tool (i.e., by forbidding the BMC tool to use invalid instructions as the input). As a result (unlike for EDDI-V and PLC in Sec. 6.5), a QED module is not needed to constrain the inputs. This input

constraint is a subset of the input constraints of Symbolic QED for the EDDI-V and the PLC QED transformations (i.e., it is just constraint 1 in Sec. 6.4). Thus, it is compatible with the input constraints of Symbolic QED for the EDDI-V and the PLC QED transformations.

Note that for the input constraint, we do not explicitly require that there must be a branch instruction in the input. This is because since the BMC tool searches through all possible sequences of instructions for the input (including sequences of instructions that contain branch instructions), if one or more branch instructions are required to activate and detect a bug, the BMC tool will automatically find a bug trace that contains the necessary branch instruction(s).

Furthermore, Symbolic QED for the CFCSS-V transformation does not insert any instructions or transform any instructions. Therefore, the *Inst_min* and *Inst_max* transformation parameters, which control when instructions created by QED transformations are inserted into a test, are not relevant.

Since the input constraint of Symbolic QED for the CFCSS-V QED transformation is compatible with the input constraints of Symbolic QED for the EDDI-V and the PLC QED transformations, and since Symbolic QED for the CFCSS-V QED transformation does not require a QED module, Symbolic QED for the CFCSS-V QED transformation may be combined with Symbolic QED for the EDDI-V and the PLC QED transformations. This is useful if one is unable to determine whether a bug affects the control flow of the processor core or other parts of the processor core / SoC.

As discussed in Sec. 6.6, one can run a simple QED test to obtain the initial state (register and memory values) of the design for the BMC tool. Since the CFCSS-V QED transformation does not require duplicated registers and memory locations, we can stop the test as soon as the test has written some values to all of the registers and memory locations. However, if we are combining Symbolic QED for the CFCSS-V transformation with Symbolic QED for the EDDI-V and the PLC transformations, the technique in Sec. 6.6 must be used to obtain an initial state that is QED-consistent (which is required by the EDDI-V and the PLC transformations). One can obtain these values using ultra-fast simulators (at a higher abstraction than RTL) that can easily simulate large designs [Sanchez 13].

A1.2 Symbolic QED for the CFTSS-V QED Transformation

CFTSS-V tracks the control flow between blocks of instructions. This is useful to determine if there is a deadlock or livelock. Note, that for CFTSS-V, one may not necessarily need to use Symbolic QED. This is demonstrated in Sec. 4.8, where QED family tests (with a range of different values for the *Inst_min* and *Inst_max* QED transformation parameters and different windows for the QED transformations) are sufficient to find a very short bug trace (9 instructions). However, if one does not want to create QED family tests, Symbolic QED for the CFTSS-V QED transformation can be used to produce short bug traces. Furthermore, since Symbolic QED searches through all

possible bug traces, it may be possible that Symbolic QED will find a bug trace that is missed by or different from the one found by the QED family tests.

In this section, we define *deadlock* as a situation when signals in the processor core do not change, and the processor core does not make progress (i.e., it does not commit any instructions). We define *livelock* as a situation when some signals in the processor core change, but the processor core does not make progress (i.e., it does not commit any instructions). Note that, we do not consider tests that contain self-loops (e.g., an infinite loop) as examples of livelocks.

If a processor core does not livelock or deadlock, then it should be able to commit new instructions. Therefore, we are interested in bug traces that correspond to situations when the processor core is unable to commit new instructions. To accomplish this, the property that we ask the BMC tool to check is the number of instructions that the processor core can commit, as shown below.

F (number of committed instructions == C)

This property states that eventually (i.e., **F**), the processor core has to commit *C* number of instructions. Thus, a counter-example to this property corresponds to a situation where the processor core is unable to commit *C* number of instructions (due to livelocks / deadlocks). The parameter *C* should be set as large as possible. However, since the design is analyzed in a BMC tool, which can only analyze a limited number of clock cycles (due to the size of the design), *C* is limited by the number of clock cycles that the

BMC tool can analyze. For example, if a processor core can only commit 1 instruction every 2 clock cycle and the BMC tool can only analyze 10 clock cycles of the design, it is unreasonable to ask the BMC tool to find a situation where the processor core is unable to commit 1,000 instructions (i.e., C = 1,000). Therefore, C corresponds to the number of instructions that the processor core can guarantee to commit in the given number of clock cycles analyzed by the BMC. C is a parameter that depends on the BMC tool used and the design being analyzed. For the OpenSPARC T2 processor core design, we determined empirically that that the maximum number for C is 17 instructions for our BMC tool (Sec. A1.3).

To keep track of the number of instructions that have committed, we add a small counter that counts the number of instructions committed (Figure A1.3). **This counter is only used within the BMC tool and is not intended to be added to the manufactured IC**. The counter counts up by 1 every time an instruction is committed, which is determined when an *instruction commit* signal from the commit stage of a processor core is active. The output of the counter corresponds to the number of committed instructions (which is used by the property check by the BMC tool). The counter is reset to 0 at the start of the BMC analysis.

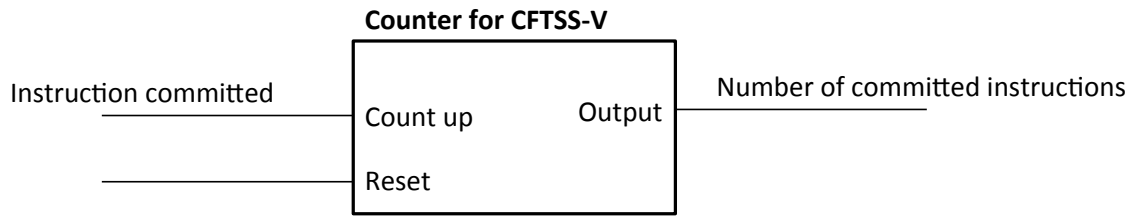


Figure A1.3. The existing fetch unit with the QED module for CFTSS-V.

Similar to Symbolic QED for the CFCSS-V QED transformation, we require the following constraints on the inputs.

> *Inputs must be valid instructions (specifications of valid instructions can be directly obtained from the Instruction Set Architecture (ISA) of the processor cores).*

As discussed in the previous section, this input constraint can be specified directly to the BMC tool (i.e., by forbidding the BMC tool to use invalid instructions as the input). We also do not explicitly require that there must be a branch instruction in the input. This is because since the BMC tool searches through a wide variety of sequences of instructions for the input (including sequences of instructions that contain branch instructions), if one or more branch instructions are required to activate and detect a deadlock / livelock, the BMC tool will automatically find a bug trace that contains the necessary branch instruction(s).

Furthermore, as is the case for CFCSS-V, Symbolic QED for the CFTSS-V transformation does not insert any instruction or transform any instructions. Therefore,

the *Inst_min* and *Inst_max* QED transformation parameters (which controls when instructions created by QED transformations are inserted into a test) are not needed.

Symbolic QED for the CFTSS-V QED transformation may be combined with the Symbolic QED for the EDDI-V and the PLC QED transformations, as well as Symbolic QED for the CFCSS-V QED transformation.

We use the same technique as that for the CFCSS-V QED transformation discussed in Sec. A1.1 to obtain the initial state for the BMC tool.

A1.3 Results

To evaluate Symbolic QED for the CFCSS-V and the CFTSS-V QED transformations, we implemented bug scenarios from Chapter 3 that affect the control flows of processor cores. Specifically, bug activation criteria 1-8 from Table 3.1 and bug effect H from Table 3.2 were implemented. In addition, we also implemented the bug from the case study presented in Sec. 4.8, i.e., a deadlock occurs when a specific sequence of 9 instructions (from the original test) are executed. The deadlock is implemented by stopping the pipeline of processor core so that the processor core does not fetch new instructions and does not commit instructions that are already in the pipeline of the processor core. The bugs were implemented by modifying the RTL of the OpenSPARC T2 (Fig. A1.4).

In the case of Symbolic QED for the CFTSS-V QED transformation, we set C = 17 to get the property: “eventually, 17 instructions have to commit”. This is because 17

instructions are the maximum number of committed instructions that can be analyzed by our BMC tool (if we set $C = 18$, the BMC tool is unable to determine if the property can be violated).

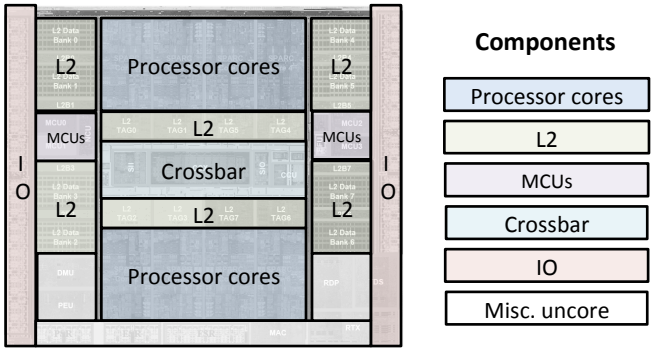


Figure A1.4. The OpenSPARC T2 SoC diagram.

Table A1.1 reports the results of Symbolic QED for the CFCSS-V QED transformation and Table A1.2 reports the results of Symbolic QED for the CFTSS-V QED transformation. Since the CFCSS-V QED transformation targets bugs that result in incorrect control-flow transitions (i.e., branch to an incorrect address) and not livelocks / deadlocks, Table A1.1 only reports results corresponding to the 8 bug scenarios that results in incorrect control-flow transitions (i.e., bug activation criteria 1-8 from Table 3.1 and bug effect H from Table 3.2) and not bugs that result in livelock / deadlock. In contrast, since the CFTSS-V QED transformation targets bugs that results in livelocks / deadlocks, Table A1.2 only reports results corresponding to the bug that results in a deadlock (i.e., the bug from the case study presented in Sec. 4.8).

The bug scenarios were implemented by modifying the RTL of the OpenSPARC T2 SoC. Since both CFCSS-V and CFTSS-V target bugs that affect the control flows of processor cores, only the processor core was analyzed in the BMC tool.

The Original (No QED) column shows results from running the original validation tests (FFT or MMULT) using end result checks to check the results of the test against pre-computed, known correct results. Similar to the case study in Sec. 4.6, for the bug scenario that resulted in a deadlock (Table A1.2) we used a time-out of 10 seconds to detect if a deadlock occurred. Since the OpenSPARC T2 is designed to operate at 1.2 GHz, this corresponds to an error detection latency of 12 billion clock cycles.

In Tables A1.1 and A1.2, the QED family test columns show results from running the same tests after CFCSS-V or CFTSS-V QED transformations. Each entry contains two sets of numbers, the top set contains the results obtained from the FFT test, and the bottom set contains results obtained from MMULT.

As discussed in Chapter 6, unlike Symbolic QED, both the Original (No QED) and the QED family tests cannot determine very precisely when a bug is activated for the bug scenarios in Tables 3.1 and 3.2. However, for the bug that causes a deadlock, we can determine which block of instructions caused the deadlock by examining what is the unique software signature stored in the variable *signature* inserted by the CFTSS-V-based QED transformation. This unique software signature corresponds to the signature of the last block of instructions executed by the processor core before the deadlock occurred. Furthermore, we can find a short sequence of instructions necessary to

activate the deadlock by sweeping the values of *Inst_min* and *Inst_max* and varying the window for QED transformation (as demonstrated in the case study in Sec. 4.8).

In Table A1.1, "Bug trace length (instructions)" shows the [minimum, average, maximum] number of instructions in the bug traces. "Bug trace length (cycles)" represents the [minimum, average, maximum] number of clock cycles required to execute the bug traces. The two numbers are different because the number of cycles per instruction (CPI) is not 1 for all instructions (for example, a load or store instruction may take multiple clock cycles to execute). For Symbolic QED, the reported "Bug trace length (instructions)" corresponds to the number of instructions in the bug trace found by the BMC tool and "Bug trace length (clock cycles)" correspond to the number of clock cycles it take to execute the bug trace. In Table A1.1, BMC runtime for Symbolic QED corresponds to the number of minutes it takes for the BMC tool to find a bug trace.

In Table A1.2, since only a single bug is analyzed, "Bug trace length (instructions)" shows the number of instructions in the bug trace, and "Bug trace length (cycles)" represents the number of clock cycles required to execute that bug trace. Again, the two numbers are different because the CPI is not 1 for all instructions. In Table A1.2, BMC runtime for Symbolic QED corresponds to the number of minutes it takes for the BMC tool to find a bug trace. For the QED family test (CFTSS-V) results, we created a family of tests by varying the *Inst_min* and *Inst_max* QED transformation parameters as well as the window of transformation (Sec. 4.4). The result in Table A1.2 corresponds to the QED test in the family that produced the shortest bug trace.

Table A1.1 Results comparing original tests (No QED), QED family tests (CFCSS-V), and Symbolic QED for CFCSS-V QED transformation for FFT (top values) and MMULT (bottom values). For bug traces, we report the [minimum, average, maximum] length in instructions and clock cycles. We also report [minimum, average, maximum] BMC runtimes in minutes.

		Original (No QED)	QED family test (CFCSS-V)	**Symbolic QED (CFCSS-V)**
Control flow bugs from Tables 3.1 and	Bug trace length (instructions)	[643,321k,1.2M] [12k,319k,1.1M]	[14,39k,91k]† [14,38k,66k]†	**[3, 3, 4]** **[3, 3, 4]**
	Bug trace length (clock cycles)	[781,620k,2.2M] [14k,619k,2.1M]	[22,37k,77k] † [22,38k,69k] †	**[13, 15, 16]** **[13, 15, 16]**
	Coverage	62.5% 62.5%	100% 100%	**100%** **100%**
	BMC runtime (minutes)	N/A	N/A	**[19, 39, 71]** **[10, 39, 73]**

† If trace buffers are used for QED, then the trace lengths in terms of instructions are: for FFT with CFCSS-V [14,92,231], MMULT with CFCSS-V [14, 120, 192]. The trace lengths in terms of clock cycles are: for FFT with CFCSS-V [22, 112, 265], MMULT with CFCSS-V [22, 141, 222]. Other entries remain the same.

Table A1.2 Results comparing original tests (No QED), QED family tests (CFTSS-V), and Symbolic QED for CFTSS-V QED transformation for FFT (top values) and MMULT (bottom values). For bug traces, we report the length in instructions and clock cycles. We also report the BMC runtime in minutes.

		Original (No QED)	QED family test (CFTSS-V)	**Symbolic QED (CFTSS-V)**
Control flow bug that results in a deadlock	Bug trace length (instructions)	~12B ~12B	9 9	**9** **9**
	Bug trace length (clock cycles)	12B 12B	19 19	**19** **19**
	Coverage	100% 100%	100% 100%	**100%** **100%**
	BMC runtime (minutes)	N/A	N/A	**72** **69**

Summary and Discussions

This appendix presented Symbolic QED for the CFCSS-V and the CFTSS-V QED transformations, which localize logic bugs that affect the control flows of processor cores. Symbolic QED for the CFCSS-V and the CFTSS-V QED transformations quickly localize bugs (in less than 1 hour for most bugs) and produce very short corresponding bug traces. Results demonstrate that the bug traces produced by Symbolic QED for the CFCSS-V QED transformations are up to 5 orders of magnitude shorter than traditional post-silicon validation tests that rely on end result checks, and up to 4 orders of magnitude shorter than QED tests (if trace buffers are not used). Moreover, bug traces produced by Symbolic QED for the CFTSS-V QED transformation are up to 9 orders of magnitude shorter than traditional post-silicon validation tests that rely on timeouts to detect deadlocks. Symbolic QED is completely automated, does not require human intervention, and does not need any additional hardware.

This appendix demonstrated the effectiveness of Symbolic QED for localizing logic bugs during post-silicon validation and debug. It is also possible to use Symbolic QED to localize logic bugs during pre-silicon verification. This is because logic bugs correspond to incorrect hardware implementation (e.g., incorrectly written RTL). Therefore, Symbolic QED, which analyzes a model of a design (in this dissertation the model is the RTL) to localize logic bugs, can be effectively used to localize logic bugs during pre-silicon validation, where the RTL model is also available.

PUBLICATIONS FROM THIS DISSERTATION

- T. Hong, Y. Li, S. Park, D. Mui, D. Lin, Z. Khaleq, N. Hakim, H. Naeimi, D. Gardner and S. Mitra, "QED: Quick Error Detection Tests for Effective Post-Silicon Validation," *Proceedings of the IEEE International Test Conference*, pp. 1-10, Austin, TX, November 2010.

- D. Lin, T. Hong, F. Fallah, N. Hakim and S. Mitra, "Quick Detection of Difficult Bugs for Effective Post-Silicon Validation," *Proceeding of the IEEE/ACM Design Automation Conference*, pp. 561-566, San Francisco, CA, June 2012.

- D. Lin, T. Hong, Y. Li, F. Fallah, D.S. Gardner, N. Hakim and S. Mitra, "Overcoming Post-Silicon Validation Challenges through Quick Error Detection (QED)," *Proceedings of the IEEE/ACM Design Automation and Test in Europe Conference*, pp. 320-325, Grenoble, France, March 2013 (Invited).

- D. Lin and S. Mitra, "QED Post-Silicon Validation and Debug: Frequently Asked Questions," *Proceedings of the IEEE Asia and South Pacific Design Automation Conference*, pp. 478-482, Singapore, January 2014 (Invited).

- D. Lin, T. Hong, Y. Li, E. S., S. Kumar, F. Fallah, N. Hakim, D. S. Gardner and S. Mitra, "Effective Post-Silicon Validation of System-on-Chips using Quick Error Detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1573-1590, Vol. 33, Issue 10, October 2014.

- D. Lin, E. S, S. Kumar, E. Rentschler and S. Mitra, "Quick Error Detection Tests with Fast Runtimes for Effective Post-Silicon Validation and Debug," *Proceedings of the IEEE/ACM Design Automation and Test in Europe Conference*, pp. 1168-1173, Grenoble, France, March 2015.

- K. A. Campbell, D. Lin, S. Mitra, D. Chen, "Hybrid Quick Error Detection (H-QED): Accelerator Validation and Debug Using High-Level Synthesis Principles," *Proceeding of the IEEE/ACM Design Automation Conference*, pp. 53-58, San Francisco, CA, June 2015.

- D. Lin, E. Singh, C. Barrett, S. Mitra, "A Structured Approach to Post-Silicon Validation and Debug Using Symbolic Quick Error Detection," *Proceedings of the IEEE International Test Conference*, Anaheim, CA, October 2015.

REFERENCES

[Abraham 83] J. A. Abraham, E. S. Davidson, and J. H. Patel, "Memory System Design for Tolerating Single Event Upsets," *IEEE Trans. Nuclear Science*, vol. 30, no. 6, pp. 4339-4344, Dec. 1983.

[Abramovici 06] M. Abramovici, "A Reconfigurable Design-for-Debug Infrastructure for SoCs," *Proc. IEEE/ACM Des. Autom. Conf.*, 2006, pp. 7-12.

[Adir 10] Adir, A., *et al.,* "Reaching Coverage Closure in Post-Silicon Validation," *Proc. Haifa Verif. Conf.,* pp. 60-74. Springer-Verlag, 2010.

[Adir 11a] Adir, A., *et al.,* "Threadmill: A Post-Silicon Exerciser for Multi-Threaded Processors," *IEEE/ACM Design Automation Conf.*, 2011.

[Adir 11b] A. Adir, S. Copty, S. Landa, A. Nahir, G. Shurek, A. Ziv, C. Meissner, and J. Schumann, "A Unified Methodology for Pre-Silicon Verification and Post-silicon Validation," *Proc. IEEE/ACM Des. Autom. Test Eur. Conf.*, 2011, pp. 1-6.

[Aharon 95] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek, "Test Program Generation for Functional Verification of PowerPC Processors in IBM*," Proc. IEEE/ACM Des. Autom. Conf.*, 1995, pp. 279-285.

[Amyeen 09] M. E. Amyeen, S. Venkataraman and M. W. Mak, "Microprocessor System Failures Debug and Fault Isolation Methodology," *Proc. IEEE Intl. Test Conf.*, 2009, pp. 1-10.

[ARM CoreSight] ARM CoreSight, http://www.arm.com/products/system-ip/coresight

[Benso 08] A. Benso, A. Bosio, S. D. Carlo, G. Di Natale, P. Prinetto, "March Test Generation Revealed," *IEEE Trans. Comput.*, vol. 57, no. 12, pp. 1704-1713, Dec. 2008.

[Bently 01] B. Bentley and R. Gray, "Validating the Intel Pentium 4 Processor." *Intel Technology Journal*, vol. 5 no. 1, pp. 1-8, Feb. 2001.

[Bernardi 08] 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 and Syst.*, vol. 27, no. 3, pp. 570-574, Feb. 2008.

[Bohr 09] Bohr, M., "The New Era of Scaling in an SoC World," *Proc. IEEE Solid-State Circuits Conf.,* pp. 23-28, 2009.

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

[Campbell 15] K. A. Campbell, *et al.*, "Hybrid Quick Error Detection (H-QED): Accelerator Validation and Debug using High-Level Synthesis Principles," *Proc. IEEE/ACM Design Automation Conf.*, pp. 1-6, 2015

[Chandy 83] 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.

[Chang 07] K.-H. Chang, I. L. Markov, V. Bertacco, "Automated Post-Silicon Debugging and Repair," *Proc, IEEE/ACM Intl. Conf. on Comput.-Aided Des.*, 2007, pp. 91-98.

[Chang 09] Chang, K., I. L. Markov, V. Bertacco, "Bug Trace Minimization," *Functional Design Errors in Digital Circuits*, Vol. 32, pp. 77-103, 2009.

[Clarke 01] Clarke, E., A. Biere, R. Raimi, Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design*, Vol. 19, No. 1, pp. 7-34, 2001.

[Conroy 05] Z. Conroy, G. Richmond, X. Gu, and B. Eklow, "A Practical Perspective on Reducing ASIC NTFs," *Proc. IEEE Intl. Test Conf.*, 2005, pp. 1-7.

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

[De Paula 08] De Paula, F. M., *et al.*, "BackSpace: Formal Analysis for Post-Silicon

Debug," *Proc. Formal Methods in CAD*, pp. 1-10, 2008.

[De Paula 11] De Paula, F. M., *et al.*, "TAB-BackSpace: Unlimited-Length Trace Buffers with Zero Additional On-Chip Overhead," *Proc. IEEE/ACM Design Automation Conf.*, pp. 411-416, 2011.

[De Paula 12] F.M. De Paula, A.J. Hu, and A. Nahir, "nuTAB-BackSpace: Rewriting to Normalize Non-Determinism in Post-Silicon Debug Traces," *Proc. Intl. Conf. on Comput. Aided Verification*, 2012*,* pp. 513-531.

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

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

[DeOrio 11] DeOrio, A., D. S. Khudia, and V. Bertacco, "Post-Silicon Bug Diagnosis with Inconsistent Executions," *Proc IEEE Intl. Conf. Computer-Aided Design*, pp. 755-761, 2011.

[Deutsch 14] Deutsch, S., and K. Chakrabarty, "Massive Signal Tracing Using On-Chip DRAM for In-System Silicon Debug," *Proc. IEEE Intl. Test Conf.*, pp. 1-10, 2014.

[Dongerra 03] J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: Past, Present and Future," *Concurrency and Computation: Practice and Experience,* vol. 15, no. 9, pp. 803-820, Jul. 2003.

[El Mandouh 12] El Mandouh, E., and A.G. Wassal, "Automatic Generation of Hardware Design Properties from Simulation Traces," *Proc IEEE Intl. Symp. Circuits and Systems*, pp. 2317-2320, 2012.

[Ernst 07] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, C. Xiao, "The Daikon System for Dynamic Detection of Likely Invariants," *Science of Comput. Programming*, vol. 69, no. 1-3, pp. 35-45, Dec. 2007.

[Foster 07] Foster, T. J., D. L. Lastor, and P. Singh, "First Silicon Functional Validation and Debug of Multicore Microprocessors," *IEEE Trans. Very Large Scale Integration Systems,* Vol. 15, No. 5, 2007.

[Foster 15] Foster, H. D., "Trends in Functional Verification: A 2014 Industry Study," *Proc. IEEE/ACM Design Automation Conf.*, pp. 48-52, 2015.

[Friedler 14] Friedler, O., *et al.*, "Effective Post-Silicon Failure Localization Using Dynamic Program Slicing," *Proc. IEEE/ACM Design Automation Test in Europe*, pp. 1-6, 2014.

[Gao 08] M. Gao, H.-M. Chang, P. Lisherness, and K.-T. Cheng, "Time-Multiplexed Online Checking: A Feasibility Study," *Proc. IEEE Asian Test Symp.*, 2008, pp. 371-376.

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

[Gray 85] J. Gray, "Why Do Computers Stop and What Can Be Done About It," Tandem Computer, Cupertino, CA, Tech. Rep. 85.7, PN 87614, Jun. 1985.

[Hangal 05] S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, "IODINE: A Tool to Automatically Infer Dynamic Invariants," *Proc. IEEE/ACM Des. Autom. Conf.*, 2005, pp. 775-778.

[Hennessy 12] J. L. Hennessy, and D. A. Patterson, "Memory Hierarchy Design," *Computer Architecture: A Quantitative Approach,* 5th ed., Morgan Kauffman, 2012, pp. 72-131.

[Ho 95] R. C. Ho, C. H. Yan, M. A. Horowitz, and D. L. Dill, "Architecture Validation for Processors," *Proc. ACM/IEEE Intl. Symp. Comput. Arch.*, 1995, pp. 404-413.

[Ho 09] R. C. Ho, M. Theobald, B. Batson, J.P. Grossman, S. C. Wang, J. Gagliardo, M. M. Deneroff, R. O. Dror, and D. E. Shaw, "Post-Silicon Debug Using Formal Verification Waypoints," *Proc. Des. Validation Conf.,* 2009.

[Hong 10] T. Hong, Y. Li, S.-B. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "QED: Quick Error Detection Tests for Effective Post-Silicon Validation," *Proc. IEEE Intl. Test Conf.,* 2010, pp. 1-10.

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

[Jones 05] Jones, R. B., C.-J. H. Seger, D. L. Dill, "Self-Consistency Checking," *Proc. Formal Methods in CAD*, pp. 159-171, 2005.

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

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

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

[Ko 08] H. F. Ko, and N. Nicolici, "Automated Trace Signals Identification and State Restoration for Improving Observability in Post-Silicon Validation," *Proc. IEEE/ACM Des. Autom. Test Eur. Conf.*, 2008*,* pp. 1298-1303.

[Krstic 02] A. Krstic, W.-C. Lai, K.-T. Cheng, L. Chen, D. Dey, "Embedded Software-Based Self-Test for Programmable Core-Based Designs," *IEEE Des. & Test of Comput.*, vol. 19, no. 4, pp.18-27, Aug. 2002.

[Krstic 03] A. Krstic, L.-C. Wang, K.-T. Cheng, T. M. Mak, "Diagnosis-Based Post-Silicon Timing Validation Using Statistical Tools and Methodologies," *Proc. IEEE Intl. Test Conf.*, 2003, pp. 339-348.

[Li 10] Li, W., F. Alessandro, and S. A. Seshia, "Scalable Specification Mining for Verification and Diagnosis," *Proc. IEEE/ACM Design Automation Conf.*, 2010.

[Lin 12] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra, "Quick Detection of Difficult Bugs for Effective Post-Silicon Validation," *Proc. IEEE/ACM Des. Autom. Conf.*, 2012*,* pp. 561-566.

[Lin 14] Lin, D., *et al.*, "Effective Post-Silicon Validation of System-on-Chips Using

Quick Error Detection," *IEEE Trans. Computer Aided Design of Integrated Circuits Systems*, Vol. 33, No. 10, pp. 1573-1590, 2014.

[Lin 15a] Lin, D., *et al.*, "Quick Error Detection Tests with Fast Runtimes for Effective Post-Silicon Validation and Debug," *Proc. IEEE/ACM Design Automation and Test in Europe Conf.*, pp. 1168-1173, 2015.

[Lin 15b] Lin, D., *et al.*, "A Structured Approach to Post-Silicon Validation and Debug Using Symbolic Quick Error Detection," *Proc. IEEE International Test Conference*, 2015.

[Liu 09] X. Liu, and Q. Xiu, "Trace Signal Selection for Visibility Enhancement in Post-Silicon Validation," *Proc. IEEE/ACM Design Automation. Test in Europe Conf.*, 2009*,* pp. 1338-1343.

[Lu 82] D. J. Lu, "Watchdog Processors and Structural Integrity Checking," *IEEE. Trans. Comput.*, vol. 31, no. 7, pp. 681-685, Jul. 1982.

[Lovellette 02] M. N. Lovellette, K. S. Wood, D. L. Wood, J. H. Beall, P. P. Shirvani, N. Oh, and E. J. McCluskey, "Strategies for Fault-tolerant Space-based Computing: Lessons Learned from the ARGOS Testbed," *Proc. Aerospace Conf.*, 2002, pp. 5-2109 - 5-2119.

[Martin 05] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xiu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-Drive Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Comput. Arch. News*, vol. 33, no. 4, pp. 92-99, Nov. 2005.

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

[Mitra 10] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-Silicon Validation Opportunities, Challenges and Recent Advances," *Proc. IEEE/ACM Des. Autom. Conf.*, 2010, pp. 12-17.

[Nahir 14] Nahir, A., *et al.*, "Post-Silicon Validation of the IBM POWER8 Processor," *Proc. IEEE/ACM Design Automation Conf.*, pp. 1-6, 2014.

[Neishaburi 11] M. H. Neishaburi, and Z. Zilic, "Hierarchical Embedded Logic Analyzer for Accurate Root-Cause Analysis," *Proc. IEEE Intl. Symp. Defect and Fault Tolerance in VLSI and Nanotechnology Syst.*, 2011, pp. 120-128.

[Oh 02a] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE Trans. Reliability*, vol. 51, no.1, pp. 63-75, Mar. 2002.

[Oh 02b] 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.

[Oh 02c] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control Flow Checking by Software Signatures," *IEEE Trans. on Reliability,* vol. 51, no. 1, pp. 111-122, Mar. 2002.

[Olukotun 98] K. Olukotun, M. Heinrich, and D. Ofelt, "Digital System Simulation: Methodologies and Examples," *Proc. IEEE/ACM Des. Autom. Conf.*, 1998, pp. 658-663.

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

[Park 09] 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.

[Park 10] Park, S.-B., *et al.,* "BLoG: Post-Silicon Bug Localization in Processors Using Bug Localization Graph", *Proc. IEEE/ACM Design Automation Conf.,* pp. 368-373, 2010.

[Parvathala 02] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS - A microprocessor functional BIST method," *Proc. IEEE Intl. Test Conf.*, 2002, pp. 590-598.

[Patra 07] P. Patra, "On the Cusp of a Validation Wall," *IEEE Des. & Test of Comput.*, vol. 24, no. 2, pp. 193-196, Mar. 2007.

[Poehl 10] F. Poehl, F. Demmerle, J. Alt, and H. Obermeir, "Production Test Challenges for Highly Integrated Mobile Phone SoCs – A Case Study," *Proc. IEEE European Test Symp.*, pp. 17- 22, 2010.

[Questa] Questa Formal http://www.mentor.com/products/fv/questa-formal/.

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

[Reick 12] K. Reick, "Post-Silicon Debug – DAC Workshop on Post-Silicon Debug: Technologies, Methodologies, and Best-Practices," *IEEE/ACM Des. Autom. Conf,* 2012.

[Sanchez 13] Sanchez, D., and C. Kozyrakis, "ZSIM: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," *Proc. ACM Intl. Symp. Computer Architecture,* pp. 475-486, 2013.

[Schelle 10] Schelle, G., *et al.*, "Intel Nehalem Processor Core Made FPGA Synthesizable," *Proc. ACM/SIGDA Intl. Symp. Field Programmable Gate Arrays*, pp. 3-12, 2010.

[Shen 83] J. P. Shen, and M. A. Schuette, "On-line Self-Monitoring Using Signatured Instruction Streams," *Proc. IEEE Intl. Test Conf.*, 1983, pp. 275–282.

[Shen 98] S. Shen, and J. A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self-Test and Design Validation," *Proc. IEEE Intl. Test Conf.*, 1998, pp. 990-999.

[Shirvani 00] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-Implemented EDAC Protection Against SEUs," *IEEE Trans. on Reliability*, vol. 49, no. 3, pp 273-284, Sep. 2000.

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

[Tektronix] Tektronix, Clarus™ SoC Post-Silicon Validation Solution, http://www.tek.com/embedded-instrumentation/

[Van Campenhout 00] D. Van Campenhout, T. Mudge, and J. P. Hayes, "Collection and Analysis of Microprocessor Design Errors," *IEEE Des. & Test of Comput.,* vol. 17, no. 4, pp. 51-60, Oct.-Dec. 2000.

[Vasudevan 10] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, D. Johnson, "GoldMine: Automatic Assertion Generation Using Data Mining and Static Analysis," *Proc. IEEE/ACM Des. Autom. Test in Eur.*, 2010, pp. 626-629.

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

[Vermeulen 02] 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.

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

[Wisam 13] Wisam, K., *et al.*, "Improving Post-Silicon Validation Efficiency by Using Pre-Generated Data," *Proc. Intl. Haifa Verif. Conf.*, pp. 166-181, 2013.

[Woo 95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. ACM/IEEE Intl. Symp. Comput. Arch.*, 1995, pp. 24-36.

[Yang 09] Y. Yang, N. Nicolici, and A. Veneris, "Automated Data Analysis Solutions to Silicon Debug," *Proc. IEEE/ACM Des. Autom. Test in Eur.*, 2009, pp. 982-987.

[Yerramilli 06] Yerramilli, S., "Addressing Post-Silicon Validation Challenges: Leverage Validation & Test Synergy," Keynote, *IEEE Intl. Test Conf.*, 2006.

[Zhu 11] C.S. Zhu, G. Weissenbacher, S. Malik, "Post-Silicon Fault Localisation Using Maximum Satisfiability and Backbones," *Proc. IEEE/ACM Formal Methods Comp.-Aided Des.,* 2011, pp. 63-66.