QED: Quick Error Detection Tests for Effective Post-Silicon Validation

Ted Hong[1], Yanjing Li[1], Sung-Boem Park[3], Diana Mui[1], David Lin[1], Ziyad Abdel Kaleq[1], Nagib Hakim[3], Helia Naeimi[3], Donald S. Gardner[3], Subhasish Mitra[1, 2]

[1]Dept. of EE and [2]Dept. of CS
Stanford University
Stanford, CA 94305 USA

[3]Intel Corporation
Santa Clara, CA 95054 USA

Abstract

Long *error detection latency*, the time elapsed between the occurrence of an error caused by a bug and its manifestation as a system-level failure, is a major challenge in post-silicon validation of robust systems. In this paper, we present a new technique called *Quick Error Detection* (*QED*), which transforms existing post-silicon validation tests into new validation tests that significantly reduce error detection latency. QED transformations allow flexible tradeoffs between error detection latency, coverage, and complexity, and can be implemented in software with little or no hardware changes. Results obtained from hardware experiments on quad-core Intel® Core™ i7 hardware platforms and from simulations on a multi-core MIPS processor design demonstrate that:

1. QED significantly improves error detection latencies by six orders of magnitude, i.e., from billions of cycles to a few thousand cycles or less.

2. QED transformations do not degrade the coverage of validation tests as estimated empirically by measuring the maximum operating frequencies over a wide range of operating voltage points.

3. QED tests improve coverage by detecting errors that escape the original non-QED tests.

1 Introduction

The goal of post-silicon validation is to test manufactured chips in actual systems to ensure that no bugs escape to the field. A wide variety of validation tests – random instruction tests, architecture-specific focused tests, and end-user applications such as operating systems, games, and scientific applications [Intel 03] – are run, during which system responses are monitored for anomalous behaviors such as crashes, hangs, exceptions, or incorrect results. Any observed anomaly is debugged to determine its cause. The effort to debug from observed failures dominates the overall post-silicon validation effort for processors [Josephson 06], especially for elusive electrical bugs. *Electrical bugs* manifest themselves only under specific operating conditions (voltage, frequency, and/or temperature) [Patra 07] and may be caused by design marginalities, synchronization problems, noise, etc. It is critical to detect these bugs quickly after they manifest to enable effective debug.

Error detection latency, the time elapsed between the occurrence of an error caused by a bug and its detection at an observable point in the test program, can be as long as several billions of cycles. Long error detection latencies limit the effectiveness of existing post-silicon debug techniques that rely on simulation, formal analysis, and tracing. Simulation is orders of magnitude slower than actual silicon [Olukotun 98]; formal analysis over more than hundreds of cycles can be difficult [Ho 09]; and tracing is limited by the availability of on-chip storage [Abramovici 06]. In addition, long error detection latencies may also result in increased error masking, i.e., an error may not propagate to an observable point.

Inter-core interactions in multi-core System-on-Chips (SoCs) can result in extremely long error detection latencies. For example, suppose that an erroneous value caused by an electrical bug in Core 1 is stored into shared memory (Fig. 1). Several millions of cycles may elapse before a bug-free core (Core 2 in Fig. 1) eventually loads and processes the erroneous value, resulting in a system failure. The time from Core 1's store to Core 2's load, the *inter-core store-to-load latency*, is a lower bound on the error detection latency. Additional cycles, including additional inter- and intra-core stores and loads, may be required to propagate the error to an observable point in the test program. Figure 2 presents the distributions of inter-core store-to-load latencies for two representative programs from the Splash2 benchmark suite [Woo 95], executed on a simulated 4-core 4-way out-of-order MIPS processor [Renau 05]. For FMM, more than 82% of all inter-core store-to-load latencies are greater than one million cycles and more than 97% are greater than 100 thousand cycles. As discussed earlier, such long error detection latencies are very challenging for post-silicon debug.

This paper presents *Quick Error Detection tests* or *QED tests* to overcome the error detection latency challenge during post-silicon validation of processors. QED tests are obtained by transforming existing post-silicon validation tests into new tests with significantly lower (i.e., better) error detection latencies. QED tests are enabled by a variety of *QED transformations*, requiring software-only or hardware-software changes. Furthermore, QED transformations allow flexible tradeoffs between error detection latency, coverage (i.e., the percentage of bugs detected by a test program), and complexity (i.e., additional hardware and software modifications required for QED). Target error detection latencies are configurable and can range from very few cycles to a few thousand cycles, depending on the desired tradeoffs.

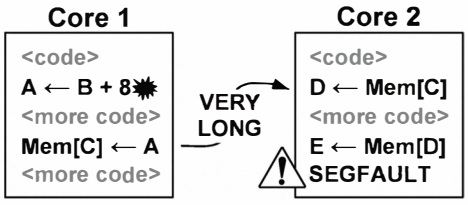


Figure 1. Illustration of long error detection latency due to inter-core interactions.

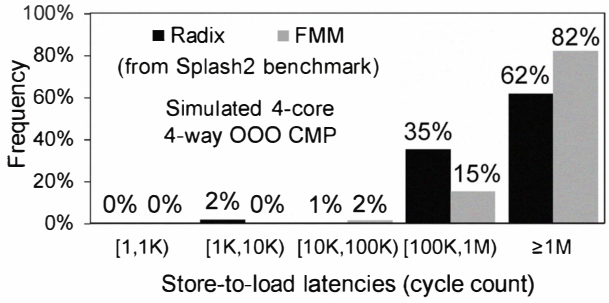


Figure 2. Distribution of inter-core store-to-load latencies.

978-1-4244-7207-9/10/$26.00 ©2010 IEEE

This paper targets electrical bugs for three reasons:

1. Electrical bugs are often very time-consuming to debug [Josephson 01].

2. Electrical bugs can be modeled as bit-flips at flip-flops. This is an effective model because most electrical bugs eventually manifest themselves as incorrect values arriving at flip-flops [McLaughlin 09]. The existence of such electrical bug models allows for simulation experiments.

3. There is little consensus about logic bug models [ITRS 09].

Results obtained from both hardware experiments and simulations demonstrate that:

1. QED significantly improves error detection latencies by six orders of magnitude, from billions of cycles to a few thousand cycles or less. With such short error detection latencies, bugs in processor cores can be detected very fast and can be effectively analyzed using debug techniques such as IFRA [Park 09], Backspace [De Paula 08], and trace buffers [Abramovici 06].

2. QED transformations do not degrade coverage of validation tests as observed empirically by measuring *Fmax* values, the maximum operating frequency values over a wide range of operating voltage points [Josephson 02].

3. QED tests improve coverage by detecting errors that escape the original non-QED tests. Coverage is often limited by silent errors and masked errors. A *silent error* occurs when a bug causes an error that is propagated to an observable point, but insufficient checking misses the error. A *masked error* occurs when a bug causes an error that is not propagated to an observable point [Barton 90]. Since comprehensive checks are instrumented by QED transformations, occurrences of silent and masked errors can be significantly reduced.

The major contributions of this paper are:

1. Introduction of the QED idea for post-silicon validation of processors.

2. Experimental results obtained from quad-core Intel® Core™ i7 platforms demonstrating six orders of magnitude reduction (i.e., improvement) in error detection latencies using QED. These results are also confirmed by detailed simulations of a 4-core 4-way out-of-order MIPS processor.

3. Empirical experimental results obtained from our hardware platforms demonstrating that a QED test can detect close to 4X more errors compared to a test utilizing only end-result-checks that compare actual program outputs to expected outputs.

4. Empirical analysis of QED's impact on coverage, as measured by Fmax, showing that a) QED tests do not degrade coverage, and b) QED tests improve coverage. These results are demonstrated using shmoo plots, spanning a wide range of voltage and frequency operating points, obtained from our hardware platforms.

Section 2 introduces QED transformations. Section 3 presents hardware- and simulation-based experiments, as well as experimental results demonstrating the effectiveness of QED. Related work is presented in Sec. 4, followed by conclusions and future work in Sec. 5.

2 QED Transformations

The idea of QED is inspired by concurrent error detection used in fault-tolerant computing, e.g., [Lu 82, Mahmood 88, Oh 02a, 02b, Rotenberg 99, Saxena 00]. However, post-silicon validation introduces unique requirements and opportunities distinct from fault-tolerant computing:

1. Unlike fault-tolerant computing, post-silicon validation tests do not need fault containment and recovery.

2. Unlike fault-tolerant computing where "high-level" checks are generally preferred to reduce performance penalties, some performance penalties may be acceptable in post-silicon validation. Instead, minimizing error detection latency is of paramount importance, since debug time rather than test execution time is a major bottleneck.

3. In post-silicon validation, test program inputs may be known *a priori* [Bentley 01]. This presents a unique opportunity for aggressive checking: QED transformations may be optimized for the corresponding test inputs.

4. Transformations for post-silicon validation tests must not adversely degrade coverage.

Compared to fault-tolerant computing, the first three aspects suggest that QED transformations can be "simpler." However, the last constraint requires that QED transformations must provide enough flexibility to avoid degradation of coverage.

The next two sections present two families of QED transformations. Both families are based on the concept of instruction duplication and comparison of results produced by the original and the duplicated code. With proper granularity of instruction duplication and checking, errors caused by bugs can be quickly detected, provided that the original and the duplicated blocks of instructions are not **identically** affected by the errors. For intermittent electrical bugs, it is unlikely that the same error would appear in two separate executions of the same code [Patra 07]. The concept of QED can be further extended to reduce the likelihood of identical error effects by executing the original and duplicated code blocks "differently" through the incorporation of design diversity into QED (e.g., through data, time, or algorithmic diversity) [Mitra 02, Oh 02b]. A comprehensive evaluation of such diversity-enhanced QED is beyond the scope of this paper.

2.1 Error Detection by Duplicated Instructions for Validation (EDDI-V)

Error Detection by Duplicated Instructions for Validation (EDDI-V) is a QED transformation that extends the EDDI technique used in fault-tolerant computing [Oh 02a]. EDDI-V bounds target error detection latency and provides configurability to trade off target error detection latency for less intrusiveness. Here, *intrusiveness* is loosely defined as the amount of "deviation" in the execution behavior of a QED test from that of the original test (due to the incorporation of QED). EDDI-V does not require hardware modifications and can be automated.

EDDI-V strategically duplicates instructions and compares their results. As illustrated in Fig. 3, each "block" of instructions is duplicated and a check is inserted to compare the results of the two blocks. If the check detects any error that occurs in these blocks, then the error detection latency is bounded by the sum of two terms:

1. The time elapsed between the start of the original block and the end of the duplicated block.

2. The time it takes to perform the check.

This can provide a great reduction in error detection latency compared to the original program, which may detect errors only after a visible failure (e.g., program crash) or using its original checks (if available, e.g., with end-result-checks that compare actual program outputs to expected outputs). EDDI-V and EDDI have different tradeoffs and requirements. EDDI strikes a balance between performance impact and the need for error containment and recovery. As a result, duplicated instructions and checks are

inserted before each store and branch instruction [Oh 02a]. Targeting post-silicon validation, the performance impact of EDDI-V's frequent checking is not a primary concern. Instead, we support flexible configurability in EDDI-V to trade off target error detection latency for less intrusiveness. This is achieved by varying two parameters: *Inst_min* and *Inst_max* that correspond to the the minimum and maximum number of original instructions executed before any instructions inserted by QED execute (*Inst_min* must be less than or equal to *Inst_max* by definition). Increasing *Inst_min* decreases intrusiveness, and vice-versa. Decreasing *Inst_max* increases the target 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 less than *Inst_min* instructions.

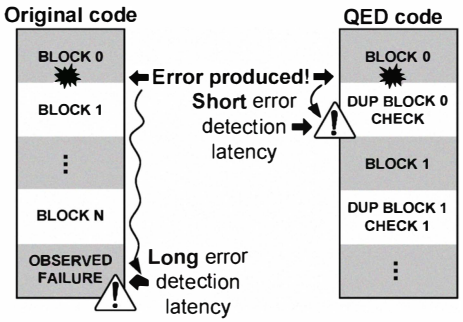


Figure 3. Error detection latency of original vs. EDDI-V-based QED test.

EDDI-V is implemented by reserving half of the general-purpose registers and memory space for the original instructions, while the other half is used by the duplicated instructions. For example, in Fig. 4a, the two original instructions in the body section of the code use four registers (A, B, C, and D). These two instructions are duplicated, and another set of registers (A', B', C', and D') is used by the duplicated instructions. In situations where there are insufficient registers, values can be stored temporarily in memory, which is also partitioned into two halves to be used by the original and duplicated instructions, respectively. Values stored in memory are then re-loaded for comparison (Fig. 4b). Note that, even with large *Inst_min*, some intrusiveness may remain due to the effects of the above code changes. Each half of the general-purpose registers and memory space are identically initialized so that original and duplicated instructions perform identical operations and obtain identical results in a bug-free system. Checking is performed by comparing the results of the original instructions vs. their duplicates. In the case of arithmetic and logic operations, the contents of the destination registers in both the original and duplicated instructions are compared; in the case of memory access operations, values loaded from or stored to memory are compared. Any mismatch in the comparison indicates an error.

Special analysis is required for certain code structures such as loops, conditionals, 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* and less than *Inst_max* instructions can be constructed. These blocks are then duplicated and checks are inserted. Likewise, it may not be possible to divide a loop body containing more than *Inst_max* instructions into blocks with more than *Inst_min* instructions each. In this case, the loop can also be unrolled until the unrolled loop body can be divided into blocks that satisfy the desired *Inst_min*. Figure 5 shows an example of loop unrolling, where *Inst_min* = *Inst_max* = 4. The original code has a loop body containing only two instructions, which is less than *Inst_min*. It is unrolled so that the body has four instructions. The unrolled loop body is then duplicated, and checks are inserted. For conditionals, we consider each execution path (including the paths of any nested conditionals) separately, and divide the instructions of each path into blocks of instructions satisfying both *Inst_min* and *Inst_max*. Note that in order to create such blocks, we may need to copy or move some instructions before or after the conditional into each execution path of the conditional. For locks, we ensure that the original and duplicated code blocks are protected by the same lock. In the case where synchronization primitives are implemented using custom code, some manual intervention may be necessary.

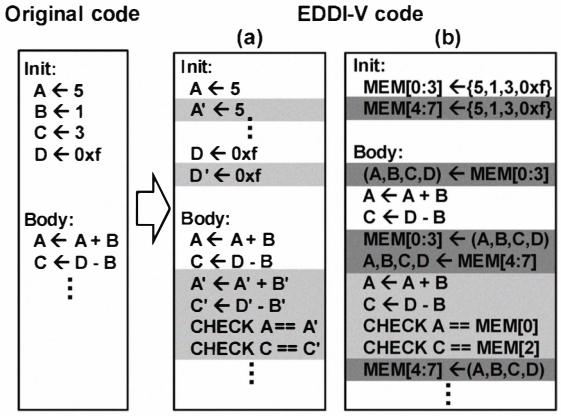


Figure 4. EDDI-V transformations: (a) with half of all general-purpose registers reserved, (b) with no registers reserved and register values stored in memory.

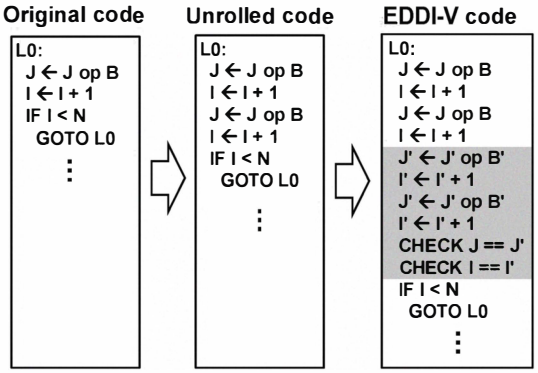


Figure 5. EDDI-V loop unrolling. The number of instructions in the original loop body is less than *Inst_min* = *Inst_max* = 4. Additional code (not shown due to space limitations) checks if N is odd and groups an extra loop iteration with two instructions outside of the loop to satisfy the constraints.

2.2 Redundant Multi-Threading for Validation (RMT-V)

Redundant Multi-Threading for Validation (RMT-V) is another QED transformation inspired by fault-tolerant computing [Mukherjee 02, Rotenberg 99, Saxena 98, 00, Wang 07]. Unlike EDDI-V, which executes the original, duplicated, and check instructions on the same thread, RMT-V executes the original instructions on one thread and uses an additional thread to execute the duplicated and check instructions. The two RMT-V threads can be simultaneously executed on different cores. As a result, the

execution of the original instructions would potentially be less affected by the execution of the duplicated and check instructions compared to EDDI-V, implying reduced intrusiveness. Like EDDI-V, RMT-V also provides flexible configurability to trade off target error detection latency and intrusiveness.

RMT-V is implemented by creating two copies of the original test code: a main thread and a check thread. The *main thread*, containing the original instructions, is instrumented with additional instructions to transmit its results to the check thread. The *check thread*, containing the duplicated instructions, is instrumented with additional instructions to receive the main thread's results and to compare them against its own results. This communication occurs via FIFO queues that can be implemented in software or hardware; a separate queue is needed for each main thread and check thread pair. Figure 6 illustrates the RMT-V mechanism. In this example, the main thread executes a block of three instructions, enqueues its results (J and K), and continues to run ahead because there is no need for error containment. The check thread concurrently executes its duplicated block and then dequeues the two values to compare with its own results. Techniques exist to ensure that neither the main thread nor the check thread execute too far apart from each other [Mukherjee 02, Rotenberg 99]. For example, the check thread can speed up its execution by using results transmitted from the main thread (this way, some dependencies in the check thread's execution can be eliminated, so more instructions can be executed in parallel). Therefore, the target error detection latency of RMT-V, as illustrated in Fig. 7, is bounded by the sum of two delays:

1. The time it takes the main thread to execute a block of instructions.

2. The delay till after the check thread checks the results of the block, which is the sum of the time it takes the main thread to enqueue the results, the time the results wait in the queue, and the time it takes the check thread to perform the checks.

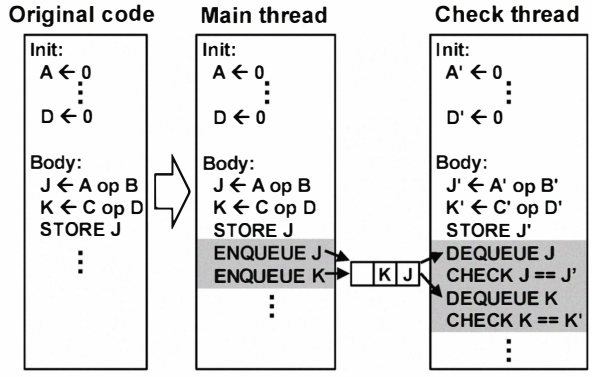


Figure 6. RMT-V transformation.

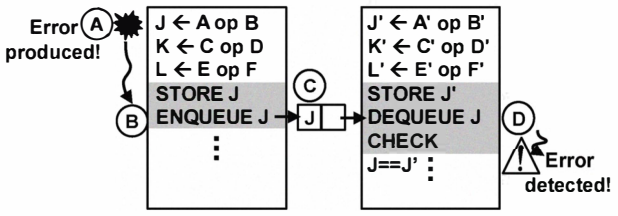


Figure 7. RMT-V error detection latency illustration.

RMT-V can be flexibly configured to trade off intrusiveness and target error detection latency by varying three parameters:

1. *Inst_min* and *Inst_max*, the minimum and maximum number of instructions executed before an enqueue is encountered, respectively. The effects of varying these two parameters on intrusiveness and target error detection latency are the same as described in Sec. 2.1 for EDDI-V. Note that, even with large *Inst_min,* some intrusiveness may remain, since the main thread may still be perturbed by the check thread, e.g., via a shared cache.

2. *Transmit overhead*, the number of instructions needed to enqueue a single register value into the FIFO. A single enqueue is required to transmit each register value to be checked from the main thread to the check thread. A lower transmit overhead means that QED interrupts the flow of the original test for a shorter amount of time per check, leading to potentially reduced intrusiveness while maintaining target error detection latency.

Transmit overhead can range from a few instructions (RMT-V without any hardware support) to zero (RMT-V with minor hardware modifications). We present three possible implementations of RMT-V below, with various options ranging from software-only techniques to hardware-assisted techniques. A comparison of these three techniques is presented in Table 1.

Software RMT-V (S-RMT-V): RMT-V can be implemented entirely in software with a small transmit overhead of three instructions per enqueue operation: an add instruction to increment a pointer to the next empty queue location, and two store instructions that store the data into the queue and mark it as valid. Our implementation utilizes the idea of lock-free queues [Michael 96] and does not require any locks to access the FIFO queues.

S-RMT-V with Hardware Queues (S-RMT-V-HQ): With queues implemented as hardware FIFOs (Fig. 8), the transmit overhead can be reduced to a single store instruction per enqueue operation: a single store instruction is able to specify the value to transmit in its data field and the destination FIFO in its address field. Small hardware modifications are needed to implement the FIFO, but no modifications to the processor cores are needed since the FIFOs can be accessed through memory-mapped I/O.

Hardware RMT-V (H-RMT-V): The intrusiveness of QED can be greatly reduced by implementing RMT-V in hardware – since no additional instructions are inserted into the main thread of H-RMT-V, the main thread's execution would be "similar" to that of the original test thread, although slight deviations between the execution of the two threads may still be possible due to cache effects, non-deterministic events (e.g., interrupts), and so on. To implement H-RMT-V, each core is augmented with a *monitor* (Fig. 8) that automatically enqueues the results of instructions to be checked. The monitor, similar to that used in [Mahmood 88, Nakka 04], observes committed instructions, determines if any instruction should be checked (e.g., the monitor can be implemented to check every store instruction, or every other add instruction, etc.), and, if so, directly sends an enqueue command to the appropriate hardware FIFO. This results in zero transmit overhead.

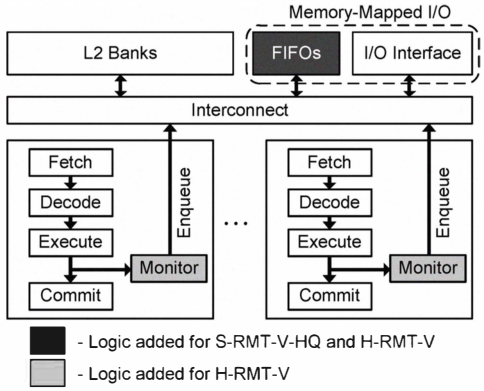


Figure 8. Hardware support for S-RMT-V-HQ & H-RMT-V.

Table 1. Comparsion of RMT-V implementation techniques.

	S-RMT-V	S-RMT-V-HQ	H-RMT-V
Transmit overhead (per enqueue)	3 instructions	1 instruction	0
Intrusiveness	Small	Smaller	Smallest
Error detection latency	Flexible	Flexible	Small
Hardware modifications	None	Very small	Some

3 Hardware and Simulation Experiments and Results

Key metrics for QED evaluation include error detection latency and coverage. Both of these metrics are very difficult to measure in hardware. Without sophisticated debug equipment, it is extremely difficult to identify the exact point in time when an error occurs. Similarly, it is very difficult to determine whether a QED test alters the system's internal electrical state in a way that can adversely affect coverage compared to the original test.

3.1 Hardware Experiments and Results

Figure 9 shows a quad-core Intel® Core™ i7 processor 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, and operating voltage and frequency values).

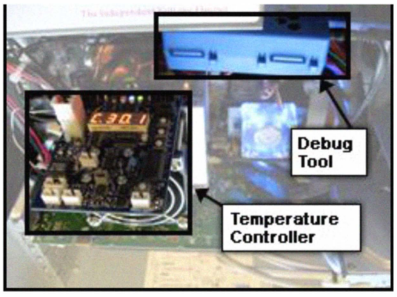


Figure 9. Quad-core Intel® Core™ i7 system with a DX58SO motherboard, a temperature controller, and a debug tool.

3.1.1 Error Detection Latency Experiment

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 serves as 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., is pessimistic) for error detection latency.

By sweeping frequency, voltage, and temperature values, we first identified conditions under which the system would operate with and without errors. By programming the desired settings using the debug tool, the window of vulnerability is created by temporarily switching 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. Therefore, any error must have occurred during the window of vulnerability, which lasts for no more than a few hundred million cycles.

In our vulnerability-window-injection experiment, the reliable and unreliable operating conditions (voltage-frequency pairs) 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.

The validation test used in this experiment is the Linpack benchmark, which is a widely-used high-performance computing 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 instrumenting EDDI-V at the source code level. For every arithmetic or logic statement, we duplicated the statement, stored the result in a different variable, and compared this result to the original (Fig. 10).

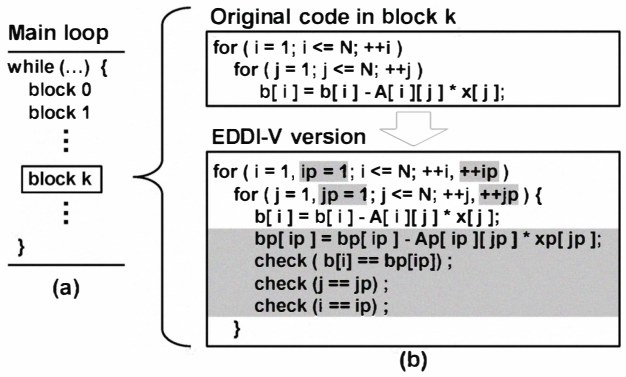


Figure 10. (a) Linpack program structure. (b) Source code level EDDI-V transformation.

Figure 11a shows the experiment flowchart:

1. We first selected t and injected a vulnerability window t cycles after each main loop iteration. The value of t is arbitrarily chosen to be t_0 initially.

2. When the Linpack test finished execution after two hours[1], we examined the test log file to determine if any QED check detected an error in any main loop iteration. Note that, although we injected vulnerability windows in the same way for all main loop iterations, errors were not necessarily observed in all iterations.

3. If no error was detected, we selected another arbitrary initial value of t and repeated from Step 1. Otherwise, we performed an

[1] The system may crash within the two hours, before the QED test detects and logs an error. In these cases, we restarted the test until the cumulative test run time reached two hours. We are not able to report the cases where the injection of a vulnerability window resulted in a crash – the limitations of our experiment setup prevent us from accurately capturing injection-to-detection latencies in these cases.

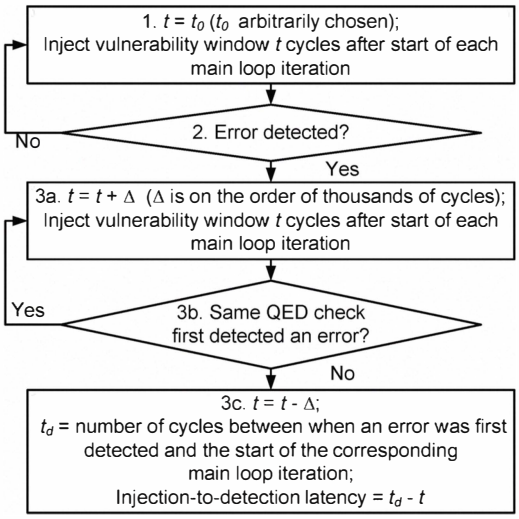
iterative procedure to move the start of the vulnerability window closer to when an error was first detected:

3a. We incremented t by Δ (Δ is on the order of thousands of cycles) and ran the Linpack test for two hours again. Vulnerability windows were injected for every main loop iteration t cycles after the start of the main loop iterations.

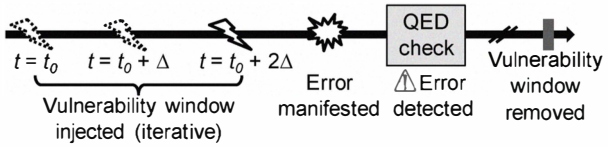
3b. We examined the test log files to determine if the same QED check continued to be the first to detect an error.

3c. If errors continued to be first detected by the same QED check, we repeated from Step 3a. Otherwise, we decremented t by Δ, and obtained injection-to-detection latency by subtracting t from t_d, the cycle count from the start of the main loop iteration to when an error was first detected. t_d is the absolute difference between the cycle count when an error is first detected and the cycle count at the start of the corresponding main loop iteration; both cycle counts were obtained by reading a processor timestamp counter and recorded in the test log file.

Fig. 11b illustrates the iterative procedure (Steps 3a - 3c described above). For a specific initial value $t = t_0$, if any error was detected by a QED check, we iteratively move the start of the vulnerability window closer to when the error was detected, while ensuring that the QED check that first detected an error remains the same. This allows us to obtain an accurate estimate of the injection-to-detection latency, i.e., a tight upper bound for error detection latency, because we ensure that manifestation of errors occurred between the time when vulnerability windows were injected and the time when the QED check first detected any error.



(a) Vulnerability window injection experiment flow chart.



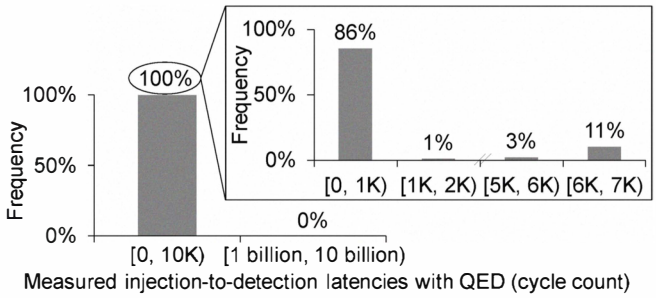
(b) Illustration of the iterative procedure to obtain accurate injection-to-detection latencies.

Figure 11. Vulnerability window injection experiment on an Intel® Core™ i7 platform.

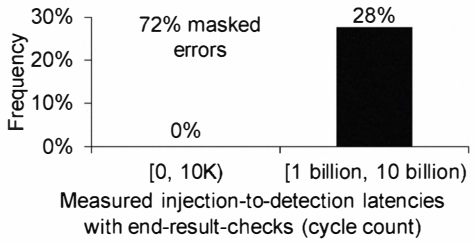
3.1.2 Error Detection Latency Results

Following the systematic experiment procedure described in Sec. 3.1.1, we obtained 75 injection-to-detection latency values. The distribution for these 75 data points is shown in Fig. 12. Figure 12a shows the results of the EDDI-V-based QED Linpack test when we take into account the QED checks. Results of the "non-QED" Linpack test shown in Fig. 12b were obtained by ignoring the QED checks and only taking into account the program'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, injection-to-detection latencies are all very short, ranging from fewer than 1,000 cycles to ~ 6,000 cycles, as shown in Fig. 12a. (Actual error detection latencies are even shorter because injection-to-detection latency is only an upper bound). On the other hand, without the QED checks (Fig. 12b), 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 cycles (even after we subtracted the latency overhead introduced by QED instrumentation, including both the duplicated and check statements).



(a) Linpack test using QED checks.



(b) Linpack test using only end-result-checks and not taking into account QED checks.

Figure 12. 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 QED checks but not crashes.

Two key observations can be made from these results:

Observation (1): QED significantly reduces error detection latencies by six orders of magnitude compared to the original (non-QED) validation test. With QED, error detection latencies are reduced from billions of cycles to a few thousand cycles or less. These short latencies enable many existing debug techniques (such as on-die trace buffers and IFRA) to be effectively used, not only for single-core processors, but also for multi-core SoCs.

Observation (2): QED detects errors that would otherwise be undetected by the original test program due to masking effects.

In addition to reducing error detection latency, QED significantly improves a test program's ability to detect errors.

3.1.3 Electrical Bug Coverage Analysis

We quantified the impact of QED on Fmax, which is often used to detect electrical bugs [Josephson 06], by generating shmoo plots across a wide range of voltage and frequency operating points. We used two versions of the Linpack program: the original non-QED version and an EDDI-V-based QED version (EDDI-V was instrumented the same way as described in Sec. 3.1.1). The non-QED version contains end-result-checks, which compare the program's final results with golden expected outputs. Note that, the coverage of non-QED test with end-result-checks is optimistic: expected values needed by end-result checks may not be available for all test programs (e.g, operating systems or games). Without end-result-checks, silent errors may impair the coverage of non-QED tests. Since all QED tests are valid stimuli (i.e., they do not introduce illegal states in the system), the Fmax values obtained using QED tests are not pessimistic.

Figure 13 details the procedure for the shmoo experiment. Both the QED and non-QED version of the test were run at least 10 times for each voltage and frequency operating point. The voltages and frequencies were specified in the BIOS, and the package temperature was fixed at 30°C. For each test run, the system was reset and the program was executed for an hour or until a system crash. Program outputs, including any errors detected by QED checks or end-result-checks, were logged to a file for later analysis.

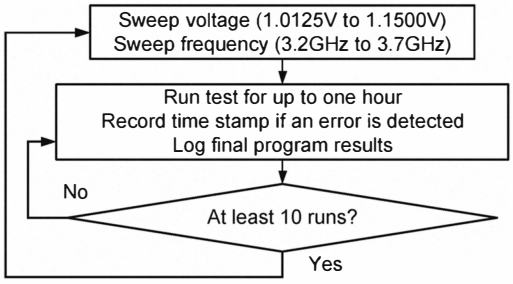


Figure 13. Shmoo experiment to evaluate the coverage of QED on an Intel® Core™ i7 platform. Two tests were run (Linpack with EDDI-V-based QED and the original Linpack with end-result-checks).

3.1.4 Electrical Bug Coverage Results

Shmoo plots for the original Linpack (with end-result-checks) and the EDDI-V-based QED Linpack test are presented in Fig. 14. Each frequency and voltage operating point is classified as:

1. Did not boot - the machine could not boot and run the test.

2. Error detected - during at least one of the runs, an error was detected by a check (an end-result-check or a QED check), or a system crash occurred.

3. Passed - no errors were observed.

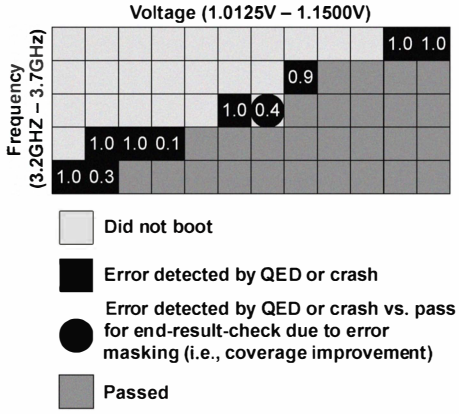
By comparing the two shmoo plots, we make three observations in addition to the two presented in Sec. 3.1.3:

Observation (3): QED does not degrade coverage as quantified by Fmax. As shown in Fig. 14, QED does not increase Fmax – under no operating point did the QED test pass when the non-QED test did not pass. This empirically establishes the fact that the QED test continues to create and detect errors for the cases where the original test creates and detects errors. We also observed the same behavior with the MPrime stress test that performs the Lucas-Lehmer primality test [Mersenne 10] (results are not shown due to space limitations).

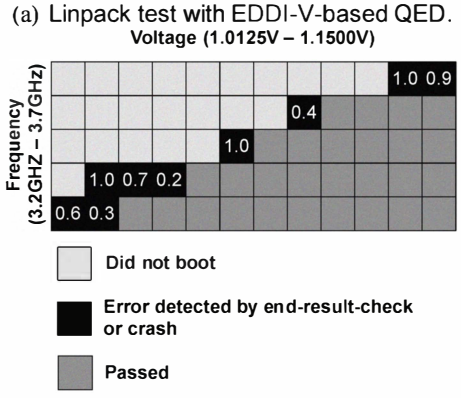
Observation (4): QED can improve coverage while significantly improving error detection latency. This is demonstrated by the voltage and frequency operating point in Fig. 14 that passes with end-result-checks, but resulted in detected errors with QED (labeled with ● in Fig. 14a).

QED tests also detect errors more readily compared to non-QED tests. In Fig. 14, we annotated any operating point classified as "error detected" with the fraction of runs in which an error was detected or a crash occurred. For example, one operating point was annotated with 0.9 in Fig. 14a, because QED checks detected errors in 5 runs, the system cashed in 4 runs, and the test passed in 1 run. The same operating point in Fig. 14b was annotated with 0.4, because end-result-checks detected errors in 3 runs, the system crashed in 1 run, and the test passed in 6 runs. For only one operating point, the QED test detected errors 1 out of 10 runs while the non-QED version detected errors 2 out of 10 runs. For all other operating points, the QED test had more than or the same number of "error detected" runs as the non-QED version.

Observation (5): Coarse-grained assertions alone <u>may not</u> be sufficient to reduce error detection latencies and achieve high coverage. By examining the test logs in which the time of each error detection was logged, we computed the average difference between the times it took end-result-checks to detect errors since the start of the test run from the times it took QED checks to detect errors. We observed that end-result-checks, which can be considered as a type of coarse-grained assertion, took several billion cycles longer to detect errors than QED.



(a) Linpack test with EDDI-V-based QED.



(b) Non-QED Linpack test with end-result-checks.

Figure 14. Linpack shmoo plots. Numbers in figures refer to the fraction of runs in which errors were detected or a crash occurred for a voltage and frequency operating point.

3.2 Simulation Results

We modeled a 4-core 4-way out-of-order MIPS processor using both the SESC microarchitectural simulator [Renau 05], and the RTL processor model from [Wang 05] that we modified to support the MIPS instruction set architecture. We performed simulations to achieve two objectives:

1. To estimate error detection latency values for the EDDI-V-based QED Linpack test program to confirm the hardware experiment results in Sec. 3.1.2.

2. To characterize error detection latency values for the H-RMT-V-based QED technique from Sec. 2.2. Since H-RMT-V requires hardware modifications, it cannot be evaluated on our existing hardware platform.

The Linpack test program, along with two applications (FMM and Radix) from the multi-threaded Splash2 benchmark suite [Woo 95], were used to create four different tests: Linpack with EDDI-V-based QED (described in Sec. 3.1); and Linpack, FMM, and Radix with H-RMT-V-based QED. The microarchitectural simulator was modified to support the H-RMT-V mechanism: memory-mapped hardware FIFOs were added, and each core was modified to automatically enqueue the values to be checked to a FIFO. In this implementation of H-RMT-V, only the operands of store and branch instructions are enqueued and sent to the check thread. The implementation can be extended to support checking for other types of instructions such as arithmetic and logic operations. Our simulation results demonstrate that our current H-RMT-V implementation enables sufficiently short error detection latencies (Fig. 15). All H-RMT-V threads were run on different cores.

We used the RTL simulator to determine the time it takes an error to affect the architectural state (general-purpose registers or main memory). We injected a single-bit-flip error into one randomly chosen flip-flop of the RTL processor model (out of a total of 18,142 flip-flops). For 10,000 such random error injections, errors were not masked in 36 cases (i.e., the errors injected eventually caused the architectural state to be different from that obtained from the error-free executions); an average of 70 cycles and a maximum of ~ 300 cycles elapsed before the injected errors propagated to the architectural state. (Note that, we disregard one case in which the injected error resulted in a deadlock before the architectural state was affected.) These latencies are consistent with those presented in [Wang 05]: almost all errors that eventually affect the architectural state have affected the architectural state within 500 instructions.

In a separate experiment, we determined the time it takes for an error in the architectural state to reach a QED check. For each of the four tests, we ran 15,000 experiments using the microarchitectural simulator. For each experiment, we chose a random instruction, flagged the instruction's result, and propagated the flag via data dependencies until a QED check was reached. Note that, we are not able to consider all cases of error masking and the effects errors have on execution paths in these experiments. However, with QED, short error detection latencies reduce the amount of error masking that can occur before a check, and extensive checking along all execution paths means that no matter which path the execution follows, any error would quickly encounter a check. Therefore, the error detection latency values obtained from these experiments are realistic.

Figure 15 shows the distribution of latencies for errors in the architecture state to reach a QED check. For the Linpack test with EDDI-V-based QED, 99.4% of all non-masked errors reached a QED check within 1,000 cycles. For the other three tests with H-RMT-V-based QED, 99.9% of all non-masked errors reached a QED check within 1,000 cycles. A small portion of non-masked errors (0.6% for EDDI-V and 0.1% for H-RMT-V) did not reach a QED check within 1,000 cycles due to the limitations of our specific implementations of QED. EDDI-V-based QED was implemented at the source code level; therefore, we were not able to instrument QED checks within system or library function calls. Our implementation of H-RMT-V did not check all instructions, but only the operands of store and branch instructions.

The distribution of error detection latencies would be similar to that presented in Fig. 15. This is because the additional time it takes for an error to propagate to the architectural state is very small – even taking into account the average delay of 70 cycles or even the maximum delay of ~ 300 cycles in addition to the latency values presented in Fig. 15, the distribution remains similar. This distribution is consistent with that obtained from the hardware experiment discussed in Sec. 3.1.2.

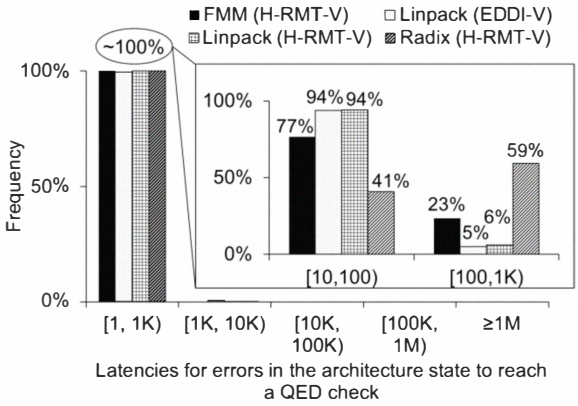


Figure 15. Simulated distribution of latencies for errors in the architecture state to reach a QED check.

3.3 Summary of Hardware and Simulation Results

Results obtained from experiments on Intel® Core™ i7 hardware platforms demonstrate that, QED not only significantly improves error detection latencies from billions of cycles to a few thousand cycles or less, but also reduces error masking and enables a test to detect more errors. Results from simulations on a multi-core MIPS processor design confirmed QED's effectiveness in improving error detection latencies. Our hardware experiments also demonstrate that QED transformations do not degrade, but improve, coverage of validation tests as estimated empirically by measuring the maximum operating frequency values over a wide range of operating voltage points: we observed that QED tests detected errors at operating points in which non-QED tests were unable to detect any errors.

4 Related Work

Prior research related to QED can be categorized into:

a. Post-silicon debug techniques: Long error detection latencies pose major barriers to the effectiveness of post-silicon debug techniques, especially for multi-core SoCs. As discussed in Sec. 3.1.2, by significantly reducing error detection latencies, QED can enable more effective post-silicon debug. Many bug localization techniques can benefit from QED, including those that rely on failure reproduction [Yang 09b], simulation [Krstic 03], and analysis of recorded signals [Park 09, 10]. Furthermore, QED can complement many other techniques that facilitate post-silicon debug, such as the selection of signals and intervals to be recorded [Liu 09, Yang 08, Yang 09a], compression of recorded events

[Anis 07, Vishnoi 09], and logic analysis or formal methods [De Paula 08, Ko 08].

b. Post-silicon validation stimulus generation: QED is applicable to and can transform a wide range of test programs into corresponding QED tests. For example, QED can be applied to automatically-generated functional tests [Benardi 08, Benso 08, Krstic 02, Parvathala 02, Shen 98].

A further benefit of applying QED is that the transformed test is "self-checking", i.e., it does not require a separate golden response (e.g., created through simulation). Techniques presented in [Raina 98] and [Wagner 08] can also generate such "self-checking" tests by using inverse operations to check the correctness of test execution. QED can utilize these techniques to incorporate diversity in its checking (which may be desirable as discussed in Sec. 2). However, these techniques alone do not provide mechanisms to ensure short error detection latencies, and may not be able to check all operations (e.g., shift operations).

c. Post-silicon validation assertions: The creation and use of assertions for validation is non-trivial: a design may have upwards of 10,000 separate assertions, many of which are manually created and must be kept up-to-date, and validated [Bentley 01]. While automatic assertion generators [Ernst 07, Hangal 05, Li 10] have been developed, the assertions they generate may not be applicable for all system and program inputs. Furthermore, assertions may not be able to detect all errors (for example, assertions may not be able to check the outputs of an ALU unit). Reconfigurable logic can ease implementation of assertions in hardware [Abramovici 06, Boule 07, Gao 08]; however, one must be careful about selecting the "right" set of assertions to be implemented in hardware. If assertions are not carefully inserted, they may not be able to achieve the desired error detection latencies (as shown in Sec. 3.1.4 for the case of end-result-check assertions).

By using transformation techniques such as EDDI-V and RMT-V, QED overcomes the difficulties with assertions by providing general and extensive checks that can be automatically generated. Moreover, if any hardware assertions are available, QED may benefit from them by "off-loading" some of its checking to these assertions, thus introducing less intrusiveness to the original validation test while achieving target error detection latency.

d. Checking techniques for fault-tolerant computing: As discussed in Sec. 2, the constraints and requirements for fault-tolerant computing and post-silicon validation are very different, though both can utilize similar checking techniques [Lu 82, Mahmood 88, Oh 02a]. For fault-tolerant computing, performance impact and error recovery are major concerns. For post-silicon validation, we must ensure that any instrumentation added to validation tests does not adversely affect coverage and is sufficient for low error detection latency. We may be able to reduce the intrusiveness of QED by incorporating application-specific checks [Huang 84, Saxena 94] developed for fault-tolerant computing.

5 Conclusions

Quick Error Detection (QED) is an effective technique that overcomes the challenges of long error detection latencies in the context of post-silicon validation of processors. In this paper, we have presented results from comprehensive hardware experiments and simulations to demonstrate that QED drastically improves error detection latencies by six orders of magnitude, from billions of cycles to a few thousand cycles or less. Such improvement in error detection latencies can enable significant gains in post-silicon validation productivity as well as significant reduction in the cost of debug equipment. Moreover, our results empirically demonstrate that QED improves coverage of post-silicon validation tests: by applying QED to existing validation tests, we are able to detect errors that would otherwise be undetected by the original non-QED tests.

Future research directions of QED include: 1. Development of a fully-automated framework for QED that includes an optimal mix of a wide range of assertions and QED transformations, in addition to the QED techniques presented in this paper. 2. Analysis and optimization of QED's effectiveness over a wider range of validation test suites and platforms. 3. Generalization of QED to logic bugs. 4. Generalization of QED to uncore components of SoCs.

6 Acknowledgements

This research is funded in part by the FCRP GSRC, SRC, NSF, and Intel Corporation. The authors thank Rahima Mohammed of Intel Corporation for help with hardware thermal management and overall methodology.

7 References

[Abramovici 06] Abramovici, M., *et al.*, "A reconfigurable design-for-debug infrastructure for SoCs," *Proc. Design Automation Conf.*, pp. 7-12, 2006.

[Anis 07] Anis, E., and N. Nicolici, "On using lossless compression of debug data in embedded logic analysis," *Proc. Intl. Test Conf.*, pp. 1-10, 2007.

[Barton 90] Barton, J., *et al.*, "Fault injection experiments using FIAT," *IEEE Trans. Computers*, 39(4), pp. 575-582, 1990.

[Bayazit 05] Bayazit, A. A., and S. Malik, "Complementary use of runtime validation and model checking," *Intl. Conf. on Computer-Aided Design*, pp. 1052-1059, 2005.

[Bernardi 08] Bernardi, P., *et al.*, "An effective technique for the automatic generation of diagnosis-oriented programs for processor cores." *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* 27(3), pp. 570-574, 2008.

[Bentley 01] Bentley, B., and R. Gray, "Validating the Intel Pentium 4 processor." *Intel Technology Journal*, 5(1), 2001.

[Benso 08] Benso, A., *et al.*, "March test generation revealed," *IEEE Trans. Computers,* 57(12), pp. 1704-1713, 2008.

[Boule 07] Boule, M., J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," *Intl. Symp. on Quality Electronic Design*, pp. 613-620, 2007.

[De Paula 08] De Paula, F.M., *et al.*, "BackSpace: formal analysis for post-silicon debug," *Proc Intl. Conf. Formal Methods in Computer-Aided Design*, pp.1-10, 2008.

[Dongarra 03] Dongarra, J., P. Luszczek, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: Practice and Experience.* 15(9), pp. 803-820, 2003.

[Ernst 07] Ernst, M.D., *et al.*, "The Daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, 69(1-3), pp. 35-45, 2007.

[Gao 08] Gao, M., *et al.*, "Time-multiplexed online checking: a feasibility study," *Proc. Asian Test Symp.*, pp. 371-376, 2008.

[Hangal 05] Hangal S., *et al.*. "IODINE: a tool to automatically infer dynamic invariants," *Proc. Design Automation Conf.*, pp. 775-778, 2005.

[Ho 09] Ho, C.R., *et al.*, "Post-silicon debug using formal verification waypoints," *DV-Con,* 2009.

[Huang 84] Huang, K.-H., and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Computers*, 33(6), pp. 518-528, 1984.

[Intel 03] Intel Corp 2003, "Intel Platform and Component Validation," http://download.intel.com/design/chipsets/labtour/PVPT_WhitePaper.pdf.

[ITRS 09] *International Technology Roadmap for Semiconductors*. 2009 ed.

[Josephson 01] Josephson, D. D., S. Poehhnan, and V. Govan, "Debug methodology for the McKinley processor," *Proc. Intl. Test Conf.,* pp. 451-460, 2001.

[Josephson 02] Josephson, D. D., "The manic depression of microprocessor debug," *Proc. Intl. Test Conf.*, pp. 657-663, 2002.

[Josephson 06] Josephson, D., "The good, the bad, and the ugly of silicon debug," *Proc. Design Automation Conf.*, pp. 3-6, 2006.

[Ko 08] Ko, H. F., and Nicolici, N., "On automated trigger event generation in post-silicon validation," *Proc. Design Automation and Test in Europe*, pp. 256-259, 2008.

[Krstic 02] Krstic, A., *et al.*, "Embedded software-based self-test for programmable core-based designs," *IEEE Design & Test of Computers*, 19(4), pp.18-27, 2002.

[Krstic 03] Krstic, A., *et al.*, "Diagnosis-based post-silicon timing validation using statistical tools and methodologies," *Proc. Intl. Test Conf.*, pp. 339-348, 2003.

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

[Liu 09] Liu, X., and Q. Xu, "Trace signal selection for visibility enhancement in post-silicon validation," *Proc. Design Automation and Test in Europe*, pp. 1338-1343, 2009.

[Lu 82] Lu, D., "Watchdog Processors and Structural Integrity Checking," *IEEE Trans. Computers*, 31(7), pp. 681-685, 1982.

[Mahmood 88] Mahmood, A., and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey." *IEEE Trans. Computers*, 37(2), pp. 160-174, 1988.

[McLaughlin 09] McLaughlin, R., S. Venkataraman, and C. Lim, "Automated debug of speed path failures using functional tests," *Proc. VLSI Test Symp.*, pp. 91-96, 2009.

[Mersenne 10] "Great Internet Mersenne Prime Search." http://www.mersenne.org/.

[Michael 96] Michael, M. M., and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," *Proc. Symp. Principles of Distributed Computing*, pp. 267-275, 1996.

[Mitra 02] Mitra, S., N. R. Saxena, and E. J. McCluskey, "A design diversity metric and analysis of redundant systems," *IEEE Trans. Computers*, 51(5), pp. 498-510, 2002.

[Mukherjee 02] Mukherjee, S. S., M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," *Proc. Intl. Symp. Computer Architecture*, pp. 99-110, 2002.

[Nakka 04] Nakka, N., *et al.*, "An architectural framework for providing reliability and security support," *Proc. Intl. Conf. on Dependable Systems and Networks*, pp. 585-594, 2004.

[Oh 02a] Oh, N., P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Reliability*, 51(1), pp. 63-75, 2002.

[Oh 02b] Oh, N., S. Mitra, and E. J. McCluskey, "ED^{4}I: error detection by diverse data and duplicated instructions," *IEEE Trans. Computers,* 51(2), pp. 180-199, 2002.

[Olukotun 98] Olukotun, K., M. Heinrich, and D. Ofelt, "Digital system simulation: methodologies and examples," *Proc. Design Automation Conf.*, pp. 658-663, 1998.

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

[Park 09] Park, S.-B., T. Hong, and S. Mitra, "Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA)," *IEEE Trans Computer-Aided Design Integrated Systems*, 28(10), pp. 1545-1558, 2009.

[Park 10] Park, S.-B., *et al.*, "BLoG: post-silicon bug localization in processors using bug localization graphs," *Proc. Design Automation Conference*, 2010.

[Patra 07] Patra, P., "On the cusp of a validation wall," *IEEE Design & Test of Computers*, 24(2), pp. 193-196, 2007.

[Raina 98] Raina, R., and R. Molyneaux, "Random self-test method - applications on PowerPC™ microprocessor caches," *Proc. Great Lakes Symp. on VLSI*, pp. 222-229, 1998.

[Renau 05] Renau, J., *et al.*, "SESC Simulator." http://sesc.sourceforge.net, 2005.

[Rotenberg 99] Rotenberg, E., "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," *Proc. Intl. Symp. Fault-Tolerant Computing*, pp. 84-91, 1999.

[Saxena 94] Saxena, N., and McCluskey, E. J., "Linear complexity assertions for sorting," *IEEE Trans. Software Engineering*, 20(6), pp. 424-431, 1994.

[Saxena 98] Saxena, N., and McCluskey, E. J., "Dependable adaptive computing systems – the ROAR project," *Proc. Intl. Conf. Systems Man and Cybernetics*, pp. 2172-2177, 1998.

[Saxena 00] Saxena, N., *et al.*, "Dependable computing and online testing in adaptive and configurable systems," *IEEE Design & Test of Computers*, 17(1), pp. 29-41, 2000.

[Shen 98] Shen, S., and J. A. Abraham, "Native mode functional test generation for processors with applications to self-test and design validation," *Proc. Intl. Test Conf.*, pp. 990-999, 1998.

[Vishnoi 09] Vishnoi, A., P. R. Panda, and M. Balakrishnan, "Cache aware compression for processor debug support," *Proc. Design Automation and Test in Europe*, pp. 208-213, 2009.

[Wagner 08] Wagner, I., and V. Bertaco, "Reversi: post-silicon validation system for modern microprocessors," *Intl. Conf. on Computer Design*, pp. 307-314, 2008.

[Wang 05] Wang, N. J., and S. J. Patel, "ReStore: symptom based soft error detection in microprocessors," *Proc. Intl. Conf. Dependable Systems and Networks*, pp. 188-201, 2005.

[Wang 07] Wang, C., *et al.*, "Compiler-managed software-based redundant multi-threading for transient fault detection," *Proc. Intl. Symp. Code Generation and Optimization*, pp. 244-258, 2007.

[Woo 95] Woo, S. C., *et al.*, "The SPLASH-2 programs: characterization and methodological considerations," *Proc. Intl. Symp. Computer Architecture*, pp. 24-36, 1995.

[Yang 08] Yang, J., and N. Touba, "Expanding trace buffer observation window for in-system silicon debug through selective capture," *Proc. VLSI Test Symp.*, pp. 345-351, 2008.

[Yang 09a] Yang, J., and N. Touba, "Automated selection of signals to observe for efficient silicon debug," *Proc. VLSI Test Symp.*, pp. 79-84, 2009.

[Yang 09b] Yang, Y., N. Nicolici, and A. Veneris, "Automated data analysis solutions to silicon debug," *Proc. Design Automation and Test in Europe*, pp. 982-987, 2009.