A Structured Approach to Post-Silicon Validation and Debug Using Symbolic Quick Error Detection

David Lin[1], Eshan Singh[1], Clark Barrett[3], Subhasish Mitra[1,2]

[1]Department of Electrical Engineering and
[2]Department of Computer Science
Stanford University, Stanford, USA

[3]Department of Computer Science
New York University
New York, USA

***Abstract*—During post-silicon validation and debug, manufactured integrated circuits (ICs) are tested in actual system environments to detect and fix design flaws (bugs). 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. We present Symbolic Quick Error Detection (Symbolic QED), a structured approach to post-silicon validation and debug. 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 even months) 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.

Keywords—Bounded Model Checking, Debug, Formal Debugging, Post-Silicon Validation and Debug, Quick Error Detection, QED, Symbolic Quick Error Detection

I. Introduction

During post-silicon validation and debug, manufactured integrated circuits (ICs) are tested in actual system environments to detect and fix design flaws (bugs). Design bugs can be broadly classified into two categories: *logic bugs* that are caused by (logic) design errors, and *electrical bugs* that are caused by subtle interactions between a design and its "electrical" state. **This paper focuses on logic bugs.**

Traditional pre-silicon verification is inadequate for "difficult" logic bugs. Critical design bugs escape pre-silicon verification and are detected only during post-silicon validation [Adir 11, Friedler 14, Foster 15, Keshava 10, Mitra 10]. Existing post-silicon validation and bug localization practices are often manual and generally *ad hoc*, and the associated costs are rising faster than design costs [Abramovici 06, Friedler 14, Nahir 14, Yerramilli 06,]. *Post-silicon bug localization* involves identifying a *bug trace* (defined as a sequence of inputs, e.g., instructions, that activate and detect the bug) and the hardware design block where the bug is (possibly) located. The effort to localize bugs from observed system failures (e.g., deadlocks, crashes, output errors) dominates the overall cost of post-silicon validation and debug [Amyeen 09, Friedler 14, Keshava 10, Nahir 14]. For example, it might take days or weeks (or even months) of (manual) work to localize and debug a single logic bug [Keshava 10, Reick 12]. New techniques are essential to reverse this trend.

Post-silicon bug localization challenges are primarily caused by long error detection latencies [Hong 10, Lin 12, 14, 15a]. *Error detection latency* is the time elapsed between when a test activates a bug and creates an error and when the error manifests as an observable failure (e.g., system crash, timeout, deadlock, exception). During post-silicon validation and debug, error detection latencies for "difficult" bugs can exceed several millions or even billions of clock cycles [Lin 12, 14]. It is extremely difficult to trace that far back into the history of system operation, especially for large designs consisting of multiple cores, cache / memory controllers, etc.

Traditional post-silicon validation and debug techniques often rely on trace buffers to generate bug traces. *Trace buffers* are small memories 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.

Assertions might be useful for post-silicon debug. However, manual assertion creation is difficult, and it is even more difficult to create assertions that can be efficiently implemented in hardware. While reconfigurable logic can somewhat ease the implementation burden [Abramovici 06], it is also difficult to select the "right" set of assertions to include. This is especially true for automatic assertion generation [El Mandouh 12, Hangal 05, Li 10, Vasudevan 10] which can result in an explosion of assertions, many of which are ineffective at catching bugs.

Many existing 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 we also show in Sec. IV).

The field of post-silicon validation and debug urgently needs a structured, automated, and scalable approach to overcome bug localization challenges. In this paper, we present such an approach called *Symbolic Quick Error Detection* or *Symbolic QED*. 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 core (a generally valid assumption for existing SoCs [Foster 15]); 2) It is broadly applicable for logic bugs inside processor cores, accelerators, and uncore components;[1] 3) It doesn't require failure reproduction; 4) It doesn't require human intervention during bug localization; 5) It doesn't require any additional hardware to localize logic bugs; and, 6) It doesn't require design-specific assertions.

[1] Uncore components refer to components in an SoC that are neither processor cores nor co-processors. Examples include interconnect fabrics, and cache / memory controllers.

978-1-4673-6578-9/15/$31.00 ©2015 IEEE

We demonstrate 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, a 500-million-transistor open-source SoC (see Sec. IV). Such bugs would generally take days or weeks (or even months) 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 small list 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-length bug trace using formal analysis; and, 5) Bug traces generated by Symbolic QED are up to 6 orders of magnitude shorter than those produced by traditional techniques.

Symbolic QED relies on the following two 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 (Sec. II); and 2) Formal techniques that enable bug localization and generation of minimal-length bug traces upon bug detection (Sec. III).

A. Motivating Example

We present a bug scenario that corresponds to a difficult bug found during post-silicon validation of a commercial multicore SoC:

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

The bug is *only* activated when two store operations to adjacent cache lines occur within 2 clock cycles of each other. 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 Fig. 1, the deadlock occurred 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 was activated, processor core 1 performed 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 a deadlock.

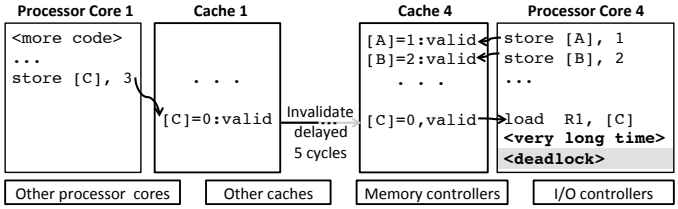

Figure 1. Example bug scenario.

When such a deadlock 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 deadlocked, 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 extraneous instructions that are not needed for activating or detecting the bug. As discussed above, such bugs are extremely challenging to localize using approaches such as trace buffers, failure reproduction, simulation, or traditional formal methods.

As shown in Sec. IV, 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 bug without requiring any failure reproduction, or any additional hardware. 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. III). 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. III and Appendix D). 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 (or even months). Furthermore, the bug traces found by traditional post-silicon techniques can be significantly longer than those found by Symbolic QED (empirically demonstrated in Sec. IV).

While the main focus of this paper is post-silicon bug localization, the Symbolic QED technique can also be used for bug detection and localization during pre-silicon verification, as well as emulation-based verification. We plan to explore these use cases in future work.

The rest of this paper is organized as follows. Section II provides an overview of the previously-published Quick Error Detection (QED) technique. Section III presents the Symbolic QED technique. Results are presented in Sec. IV, followed by related work in Sec. V. We conclude in Sec. VI, with supplemental materials in the appendices.

II. Background: Quick Error Detection (QED)

QED tests have been demonstrated to be highly effective for quickly detecting logic and electrical bugs inside processor cores, uncore components, accelerators, and components related to power-management features [Campbell 15, Hong 10, Lin 12, 14, 15a]. The software-only QED technique automatically transforms existing post-silicon validation tests (*original tests*) into new *QED tests* using various QED transformations, e.g., Error Detection using Duplicated Instructions for Validation (*EDDI-V*) and Proactive Load and Check (*PLC*).[2]

A. EDDI-V

EDDI-V [Hong 10, Lin 14] targets bugs inside processor cores 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,[3] 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 the duplicate instructions. The duplicated instructions execute in the same order as the original instructions.

The EDDI-V transformation also inserts periodic check instructions (referred to as *Normal checks* in this paper) that compare the results of the original instructions against those of the duplicated instructions. For every duplicated load instruction, an additional

[2] Symbolic QED can utilize the CFCSS-V and CFTSS-V QED transformations [Lin 14] as well, which are presented in [Lin 15b].

[3] For EDDI-V, if it is not possible to divide the registers into two halves (i.e., if the original test needs to use all of the available registers), we can use memory to store the register values. The details are in [Lin 14].

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 by the original instruction 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 the original and (corresponding) duplicate registers, respectively. A mismatch in any check instruction indicates an error. In order to minimize any intrusiveness [Lin 12, 14, 15a] 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 (maximum) number of instructions from the original test that must (can) execute before any duplicated or check instructions execute.[4]

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 in Fig. 2. 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). A PLC operation checks all the variables selected for PLC. Various PLC strategies are discussed in [Lin 12, 14, 15a].

```
LOCK(A);
LOCK(A');
Ra  = LOAD(A)
Ra' = LOAD(A')
UNLOCK(A');
UNLOCK(A);
CMP Ra, Ra' // PLC check
```

Figure 2. Example of a PLC operation for a single variable.

III. SYMBOLIC QED

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.

The Symbolic QED approach presented in this paper 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, i.e., a bug trace) is produced. BMC guarantees that if a counter-example is found, it is a minimal-length counter-example [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 Sec. IV). 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.

We address challenge (2) in Sec. III.E. 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 [Hong 10, Lin 12, 14, 15a], QED tests are excellent for detecting a wide variety of bugs; hence, we use QED checks (i.e., Normal checks, Load checks, Store checks, and PLC checks) as the properties, thus addressing challenge (1). QED tests are also designed to detect 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). The details of Symbolic QED are explained in the following subsections.

A. Solving for QED-Compatible Bug Traces Using BMC

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

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, the instruction sequence {MOV R1←1, MOV R17 ←2, CMP R1, R17} results in R1≠R17; the inequality is not caused by a bug. In order to avoid such situations, we require that counter-examples must be *QED-compatible*. We define a *QED-compatible* bug 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 its corresponding duplicate instructions have both been executed.

B. QED Module

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. **The QED module is only used within the BMC tool and is not added to the manufactured IC**; i.e., there is no performance/area/power overhead. The QED module only needs to be designed once for a given ISA, and made available

[4] Examples of EDDI-V and PLC transformations with *Inst_min* and *Inst_max* parameters are presented in Appendix A for the convenience of the reader.

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. IV). Note that, although the QED module is added to processor cores, Symbolic QED is effective not only for bugs inside processor cores, but also for bugs in uncore components, as well as bugs related to power-management features (as demonstrated in Sec. IV).

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."[5] 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 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 instructions are committed. Then, it is executed a second time, but 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 considers 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. Also note 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; [6] and (ii) the behavior of the design during BMC is deterministic. Thus, the original and the duplicate sequences of instructions must compute the same results unless there is a bug.

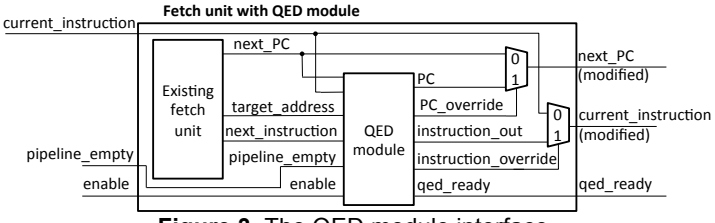

Figure 3. The QED module interface.

Figure 3 shows how the QED module integrates with the fetch unit. The pseudo code of the QED module is shown in Fig. 4. The inputs to the QED module are: 1) *enable*, which disables the QED module if 0 (this signal can be set by the validation engineers to disable the QED module); 2) *current_instruction*, which is the current instruction to be executed in the pipeline of the processor core; 3) *next_instruction*, which is the next sequential instruction after *current_instruction* (i.e., the instruction to be fetched by the fetch unit after *current_instruction*); 4) *next_PC*, which is the PC corresponding to *next_instruction*; 5) *target_address*, which is equivalent to *next_PC* unless the current instruction is a control-flow instruction, in which case it is the control-flow instruction's target address; and 6) *pipeline_empty*, which is a signal that is true if and only if there are no instructions in the pipelines of any of the processor cores and all executed instructions on all cores have been committed (i.e., the results written to registers or to memory).

The outputs from the QED module are: 1) *PC*, which is used to override the value of *next_PC*; 2) *PC_override*, which determines if the processor core should use the *PC* from the QED module or *next_PC* from the fetch unit; 3) *instruction_out*, which is used to override the value of *current_instruction*; 4) *instruction_override*, which determines whether the processor core should use the modified instruction (*instruction_out*) from the QED module or *current_instruction*; and 5) *qed_ready*, which signals when both original and duplicated registers should have the same values (under bug-free conditions). *qed_ready* is false initially; it is only set to true when both original and duplicated instructions have committed.

```
INPUT: enable, current_instruction, next_instruction, next_PC, target_address,
       pipeline_empty
OUTPUT: PC, PC_override, instruction_out, instruction_override,
        qed_ready
// initialization
mode ← ORIG; // "mode" is shared by all QED modules
rewind_address ← PC obtained from initial state (Sec. III.C);
qed_ready ← false;  PC_override_i ← 0;     instruction_override_i ← 0;
// end initialization
PC_override ←  enable ? PC_override_i : 0;
instruction_override ← enable ? instruction_override_i : 0;
if mode == CHECK then
  mode  ← ORIG;                                qed_ready ← true;
  PC ← target_address;                         PC_override_i ← 1;
  rewind_address ← target_address;
end if
if mode == ORIG, then
  qed_ready ← false;  instruction_override_i ← 0; PC_override_i ← 0;
  if is_control_flow_instruction(next_instruction) then
    mode ← WAIT1; // all QED modules go to "WAIT1" when any QED
                  // module gets a control-flow instruction
  end if
end if
if mode == WAIT1 then // wait until pipelines of all processor cores are empty
  mode ← pipeline_empty ? DUP : WAIT1;     qed_ready ← false;
  instruction_out ← NOP;                   instruction_override_i ← 1;
  PC ← rewind_address;                     PC_override_i ← 1;
end if
if mode == DUP then
  qed_ready ← false;   rewind_address ← next_PC;   PC_override_i ← 0;
  if is_control_flow_instruction(next_instruction) then
    mode ← WAIT2; // all QED modules go to "WAIT2" when any QED
                  // module gets a control-flow instruction
  end if
  instruction_out ← create_duplicated_version(current_instruction);
  instruction_override_i ← 1;
end if
if mode== WAIT2 then // wait until pipelines of all processor cores are empty
  mode ← pipeline_empty ? CHECK : WAIT2;   qed_ready ← false;
  instruction_out ← NOP;                   instruction_override_i ← 1;
  PC ← rewind_address;                     PC_override_i ← 1;
end if
```

Figure 4. Pseudo code for QED module.

The QED module has internal variables: 1) *mode*, which tracks if the processor core is executing original instructions (*ORIG*), duplicated instructions (*DUP*), in a wait mode (*WAIT1* or *WAIT2*), or if the BMC tool should do a check (*CHECK*). This variable is shared by all of the QED modules in the design so that they are always in the same mode; 2) *rewind_address*, which holds the address of the first instruction in the sequence of original instructions, (initialized to 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 modules start in *ORIG* mode. When *next_instruction* is a control-flow altering instruction, all QED modules go to *WAIT1*. In *WAIT1, PC* is set to *rewind_address*, and *PC_override_i* is set to 1 (if

[5] One could alternatively use a pseudo-instruction "QED" to trigger instruction duplication; the processor would treat this instruction as a NOP. This would allow the QED module to create sequences that would not be possible otherwise (e.g., an odd number of instructions between two control-flow altering instructions, such as {BRANCH; ADD; BRANCH}).

[6] When executing original instructions, as soon as some QED module encounters a control-flow instruction, all QED modules switch to *WAIT1* (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.

enable is 1, *PC_override* is also set to 1). The QED module also outputs NOP on *instruction_out* and sets *instruction_override_i* to 1 (if *enable* is 1, *instruction_override* is also set to 1). The QED modules stay in *WAIT1* until all of the original instructions have committed (when *pipeline_empty* becomes true, i.e., the pipelines of all processor cores are empty). Then, all QED modules switch to *DUP,* and each processor core then re-executes instructions starting from the address stored in *rewind_address*. In *DUP*, the duplicated instruction is produced 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. In *DUP*, *rewind_address* is constantly updated to *next_PC*. Then, when *next_instruction* is a control-flow altering instruction, all QED modules switch to *WAIT2* and stay in *WAIT2* until the duplicated instructions on all processor cores have committed (the *pipeline_empty* signal becomes true, i.e., the pipelines of all processor cores are empty). In *WAIT2, PC* is set to *rewind_address*. The QED module also outputs NOP on *instruction_out* and sets *instruction_override_i* to 1 (if *enable* is 1, *instruction_override* is also set to 1). After the instructions have committed, the original and corresponding duplicated registers should be equal. Then, the QED modules switch to *CHECK*. In *CHECK*, *qed_ready* is set to true. Each QED module also updates *rewind_address* to *target_address* (i.e., the address of the next instruction to execute) and sets *PC* to *target_address* and *PC_override_i* to 1. After *CHECK,* the QED modules return to *ORIG*.

An example of the transformation performed by the QED module is shown in Fig. 5. Note that, LOAD(A) is transformed into LOAD(A') during the second execution. Thus, comparing the registers (using the BMC tool) is equivalent to a PLC check on variables A and A'. There are 4 events here: (1) store to A by core 1, (2) load from A by core 2, (3) store to A' by core 1, and (4) load from A' by core 2. As explained in [Lin 15b], to avoid false fails without using locks, the QED module ensures that the order of (3) and (4) is the same as the order of (1) and (2), even if multiple cores load from A and A'. Because the BMC tool can choose a wide variety of instructions for the original sequence of instructions, this does not significantly affect the ability of Symbolic QED to activate and find bugs in general (which is empirically demonstrated in Sec. IV). However, in future work, one may want to allow the processor cores to have more freedom when executing the duplicated instructions; in that case, locks may be necessary. Memory initialization is discussed in Sec. III.C.

```
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 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.

C. Initial State

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 to 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 *next_PC* from Sec. III.B) and memory values are read out of the simulator and then used to set the register values, *PC*, *next_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.

D. Finding Counter-Examples using BMC

After inserting the QED module and setting the initial state, we use BMC to find a counter-example to the 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 and Ra' correspond to registers allocated for original instructions and duplicated instructions respectively. As mentioned above (e.g. Fig. 5), because we allow the instructions chosen by BMC to include load and store instructions, our approach can generate PLC checks, and can thus activate and detect bugs in uncore components as well as those in processor cores.

E. 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. A third technique, which uses small hardware structures referred to as *change detectors*, is discussed in Appendix D. Design reduction techniques are important not only for handling large design, but also for better bug localization.

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.

[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 Normal checks and Store checks are omitted, an error caused by a bug inside the core may propagate to an uncore component.

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. *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. Fig. 6 shows the steps for this approach. Once the full set of simplified (partially instantiated) designs is created, they can be analyzed using the BMC tool independently (in parallel). An example is presented below.

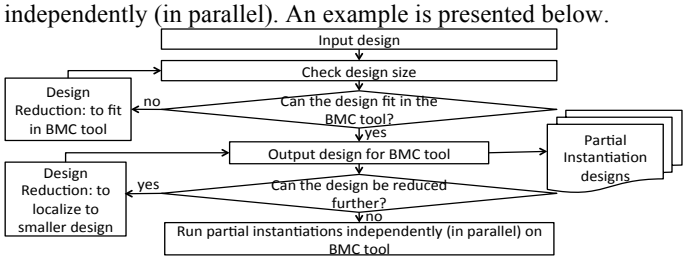

Figure 6. The partial instantiation approach for design reduction.

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 (Fig. 7). This entire design is too big to be analyzed by the BMC tool, so 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 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.

IV. RESULTS

We demonstrate the effectiveness of Symbolic QED using the OpenSPARC T2 SoC [OpenSPARC] (Fig. 7), which is the open-source version of the UltraSPARC T2, a 500-million-transistor SoC with 8 processor cores (64 hardware threads), private L1 caches, 8 banks of shared L2 cache, 4 memory controllers, a crossbar interconnect, and I/O controllers. We simulated logic bug scenarios from [Lin 12, 14, 15a], which represent a wide variety of "difficult" bug scenarios that occurred in various commercial multicore SoCs. The bug scenarios include bugs in the processor cores, bugs in the uncore components, and bugs related to power-management features.[8] They are considered difficult because they took a long time (days to weeks) to localize.

We modified the RTL of the OpenSPARC T2 SoC to incorporate these bug scenarios. For the 80 bug scenarios from [Lin 12, 14], 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 [Lin 14]; note that smaller values for *X* and *Y* imply that the bugs are more difficult to activate and detect. For example, consider the activation criterion 1 from [Lin 14]: "two stores within X clock cycles to different cache lines;" and two sequences of instructions: 1) {STORE [a], Rx; STORE [b], Ry} and 2) {STORE [a], Rx; MOV R0, 0; STORE [b], Ry}. While both sequence 1 and sequence 2 will satisfy the activation criterion when X=3 (i.e., two stores within 3 clock cycles to different cache lines), only sequence 1 will satisfy the activation criterion when X=2. For the 12 power management bug scenarios in [Lin 15a], the activation criterion is set to a sequence of 5 instructions randomly selected from the original test, executed on a designated processor core. This is to emulate a power management controller which puts the system into a power-saving state when it executes a specific sequence of instructions. If a bug is inserted into a component, the bug is included in all instances of that component.

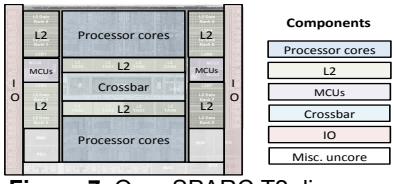

Figure 7. OpenSPARC T2 diagram.

For BMC, we used the Questa Formal tool (version 10.2c_3) from Mentor Graphics on an AMD Opteron 6438 with 128GB of RAM. We used the EDDI-V and the PLC (Sec. II) QED transformations to transform an 8-thread version of the FFT test (from SPLASH-2 [Woo 95]) and an in-house 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 [Hong 10, Lin 12, 14]).[9] Trying additional tests (beyond FFT and MMULT) was deemed unnecessary because both tests (after QED transformation) were able to detect all 92 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. III.B to the RTL of the fetch unit in the OpenSPARC T2 processor core. The resulting fetch unit with the QED module was tested using Questa 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 lengths (1 to 10 instructions long) took approximately 1 minute of runtime. Moreover, we simulated all

[8] Bug scenarios are in the appendix. The bug scenarios were simulated by modifying the RTL of the OpenSPARC T2 SoC design so that, for each bug scenario, if the bug activation criterion is satisfied, the bug effect is simulated.

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

of the bug traces produced by Symbolic QED (which depends on the QED module) to ensure that they indeed activate and detect the corresponding bugs. No additional hardware (e.g., trace buffers or change detectors discussed in Appendix D) was added to the design.

The results are summarized in Table 1. The Original (No QED) column shows results for the original validation tests (FFT or MMULT) using end-result-checks (that check the results of the test vs. pre-computed, known correct results). The QED column shows results from running the same tests after applying QED transformations. Note that, unlike Symbolic QED, both the Original (No QED) and the QED tests (without the analysis techniques discussed in Sec. III.E) are only able to report the existence of a bug; they cannot localize the 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 how the bug is activated. The table is categorized into processor core bugs, uncore bugs (bugs that are inside uncore components as well as in the interface between processor cores and uncore components), and power management bugs. Each entry contains two sets of numbers, corresponding to results obtained from the FFT test (top), and results obtained from the MMULT test (bottom).

Table 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	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	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	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.

In Table 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 completely execute their corresponding instructions to activate and detect the bug (and the cores execute the instructions in parallel).

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), and are significantly shorter than those for QED and No QED. This is because (as discussed in Sec. III) Symbolic QED uses BMC to search through all possible QED-compatible bug traces to find the minimal-length bug trace required to activate and detect the bug. **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 Fig. 10.

In Table 1, "Coverage" is the percentage of the 92 bugs detected. Both Symbolic QED and QED detected all 92 bugs, while the original tests detected only a little more than half of the bugs. This is because 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. 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 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 creation 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 bug traces for all bugs in less than 7 hours, **without relying on trace buffers or any other 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. III.E) 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. III.E) 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.[10] 1) 2

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

processor cores, 2 L2 cache banks, and the I/O controller; 2) 2 processor cores, 2 L2 cache banks, and 1 memory controller; 3) 2 processor cores, and 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; and 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. 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 example reveals that the additional components in partial instance 6 were not required for activating or detecting the bug. Specifically, for this example, while both partial instances 6 and 8 contain processor cores and caches, partial instance 8 does not have a memory controller. Thus the memory controller was not required to activate and detect the bug. Note that this partial instance also provides a small candidate list of components that may contain the bug.

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

Figure 8 reports a breakdown of the bugs localized by Symbolic QED, which correctly localized all 92 bugs. Symbolic QED localized 26.1% 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.

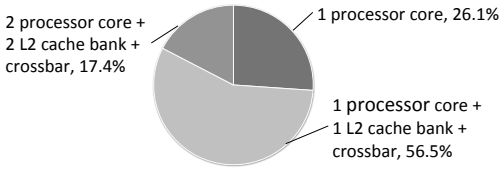

Figure 8. Graph showing the percentage breakdown (by list of candidate modules) of bugs localized by Symbolic QED. All 92 bugs were correctly localized.

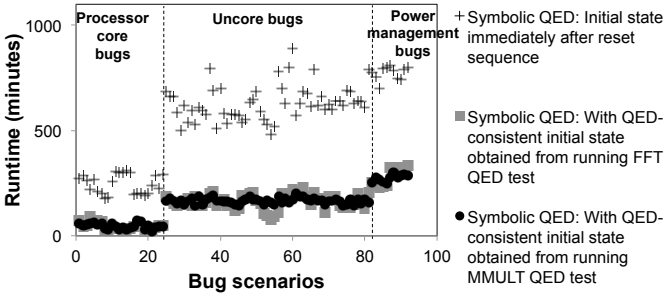

Figure 9. The BMC runtimes for Symbolic QED.

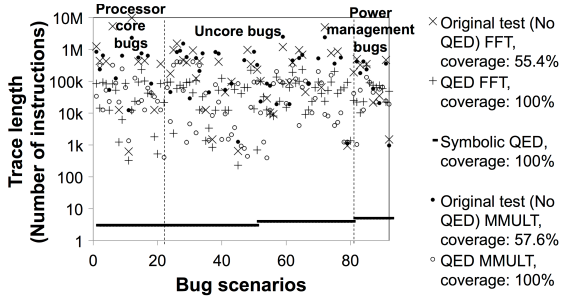

Figure 10. Trace length (in terms of number of instructions).

The BMC runtimes reported in Table 1 for Symbolic QED use the QED-consistent initial state constraint discussed in Sec. III.B. The detailed runtimes for each bug are also presented in Fig. 9. In Fig. 9, we report 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. III.B), and the runtime when similarly seeding BMC after running MMULT. Results show that using a QED-consistent initial state obtained by running a QED test improves runtimes by up to 5X compared to starting from the state immediately after a reset sequence. No significant differences were observed between the results from using the FFT test and those using the MMULT test.

In this paper, we demonstrated the effectiveness of Symbolic QED on the OpenSPARC T2. However, Symbolic QED does not rely on any information about the specific implementation of OpenSPARC T2, making it applicable to a wide variety of SoCs.

V. Related Work

The Symbolic QED technique in this paper mostly relies on QED [Hong 10, Lin 12, 14] 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. IV, 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 may be further enhanced using techniques such as self-consistency checking [Jones 05]. However, [Jones 05] addresses only processor core bugs. Our experience with bugs in commercial SoCs indicate that uncore components are also an important source of difficult bugs in SoCs [Lin 12, 14, 15a].

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

Many post-silicon bug localization approaches rely on trace buffers and assertions. Sec. I 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 doesn't require any trace buffers (or any additional hardware) 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 Sec. I 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 rely solely on formal methods for bug localization (e.g., [De Paula 08, 11, 12, Zhu 11]) are not scalable to large designs (e.g. OpenSPARC T2). Some formal techniques require specific bug models (e.g., [Zhu 11] which targets a specific model for electrical bugs) and may not work for logic bugs, since it is difficult to create 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 requires massive simulations to capture logic values of all flip-flops in the system, which will be difficult. There may be opportunities to use such techniques after Symbolic QED localizes bugs and produces short bug traces (as demonstrated in this paper).

VI. Conclusion

We presented the Symbolic QED technique, a structured and automated approach that overcomes post-silicon validation and debug challenges. It automatically detects and localizes logic bugs in post-silicon validation and provides a list of 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. It is completely automated, does not require human intervention, 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 debugging may take days or fail completely for large designs such as the OpenSPARC T2. As demonstrated in this paper, Symbolic QED is effective for bugs inside processor cores, bugs inside 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]).

There are several directions for future work. Symbolic QED can be expanded to: 1) detect and localize bugs during pre-silicon or emulation-based verification; 2) localize electrical bugs during post-silicon validation (this paper's focus was on logic bugs); 3) perform full system-level bug localization; 4) perform diagnosis of manufacturing defects during system-level testing; 5) localize bugs in analog and mixed signal components; and 6) use a more general QED module that does not require duplicated instructions to start execution on the same clock cycle on all processor cores (locks may be needed to avoid false fails) and that uses a pseudo-instruction "QED" (instead of a control-flow instruction) to trigger duplication.

Acknowledgement

We thank the reviewers for their valuable suggestions.

References

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

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

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

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

[Campbell 15] Campbell, K., D. Lin, S. Mitra, D. Chen, "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.

[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.

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

[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.

[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.

[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.

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

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

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

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

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

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

[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] Lin, D., *et al.*, "Quick Detection of Difficult Bugs for Effective Post-Silicon Validation," *Proc. IEEE/ACM Design Automation Conf.*, pp. 561-566, 2012.

[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," to appear in *IEEE Design Automation and Test in Europe Conf.*, 2015.

[Lin 15b] Lin, D., "QED Post-Silicon Validation and Debug," *Stanford Ph.D. Dissertation, 2015.*

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

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

[OpenSPARC] Available: http://www.opensparc.net

[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 Circuits System*, Vol. 28, No. 10, pp. 1545–1558, 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.

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

[Vasudevan 10] Vasudevan, S., *et al.*, "GoldMine: Automatic Assertion Generation Using Data Mining and Static Analysis," *Proc. IEEE Design Automation and Test in Europe Conf.*, pp. 626-629, 2010.

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

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

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

Appendix

A. EDDI-V Transformation Example

An EDDI-V example is shown in Fig. A1. Fig. A1(a) is the original test. Fig. A1(b) is the transformed test with *Inst_min* = *Inst_max* = 3.

```
//initialization
R1  = 1
R2  = 2
R3  = 3
R4  = 4
R5  = 5
R6  = 6
//code
R1 = R2 + R3
R4 = R5 – R6
R4 = R1 – R4
B label
...
```

(a)

```
//initialization
R1  = 1   R17  = 1
R2  = 2   R18  = 2
R3  = 3   R19  = 3
R4  = 4   R20  = 4
R5  = 5   R21  = 5
R6  = 6   R22  = 6
//code
R1  = R2  + R3
R4  = R5  – R6
R4  = R1  – R4
R17 = R18 + R19
R20 = R21 – R22
R20 = R17 – R20
CMP R4, R20
BNE ERROR_DETECTED
B label
...
```

(b)

Figure A1. EDDI-V example with *Inst_min* = *Inst_max* = *3*.

B. PLC Transformation Example

In Fig. A2(a), *Inst_min* = *Inst_max* = 4 instructions. In Fig. A2(b), *PLC_list* is a list of tuples of the form <*original variable pointer, EDDI-V variable pointer*>, where *original variable pointer* is the address of a variable from the original test selected to be checked by PLC operations and *EDDI-V variable pointer* is the address of the corresponding EDDI-V variable. A PLC operation checks all tuples in *PLC_list*. Variable selection strategies for PLC are given in [Lin 12, 14]. The runtime of PLC-based QED tests can be significantly shortened with a small amount of additional hardware support [Lin 15a].

Transformed Code	PLC Operation

...
<PLC Operation>
R1 = R2 + R3
R4 = R5 – R6
R17 = R18 + R19
R20 = R21 – R22
<PLC Operation>
R7 = R1 – R4
R9 = R7 * R8
R23 = R17 – R20
R25 = R23 * R24
<PLC Operation>
...
	for <A,A'> in PLC_list **do**
	LOCK(A)
	LOCK(A')
	Rt = LOAD(A)
	Rt' = LOAD(A')
	UNLOCK(A')
	UNLOCK(A')
	CMP Rt, Rt'
	BNE ERROR_DETECTED
	end for

(a)	(b)

Figure A2. PLC example. with *Inst_min* = *Inst_max* = *4*.

C. Bug Scenarios

A bug scenario is formed by pairing one bug activation criterion with one bug effect.

Table A1.A. Bug activation criterion from [Lin 12, 14].

Uncore components	1. Two stores within *X* clock cycles to different cache lines.
	2. Two stores within *X* clock cycles to the same cache line.
	3. Two stores within *X* clock cycles to adjacent cache lines.
	4. Two cache misses within *X* cycles.
	5. A sequence of loads and/or stores within *X* clock cycles.
Processor cores	6. Data forwarding between pipeline stages.
	7. Two branch instructions within *X* clock cycles.
Other	8. A randomly chosen clock cycle.

Table A1.B. Bug effects from [Lin 12, 14].

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.

Table A2.A. Power management bug activation criterion [Lin 15a].

ID	Description
1	When exiting from power-saving state.

Table A2.B. Power management bug effects [Lin 15a].

Type	ID	Description
Uncore components	A	The value of the next load operation from data cache is corrupted to all 0's.
	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's.
	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's 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's.

D. Change Detectors for Design Reduction

In this supplemental section, we present a design reduction technique that uses small hardware structures, referred to as change detectors. We insert change detectors (Fig. A3) to record changes in the logic values of signals during validation. These change detectors are inserted at the boundaries of all components that may potentially be removed when creating partial instantiations (e.g., at a certain level in the RTL hierarchy). For example, for the results below, change detectors were inserted on all signals one hierarchical level below the main SoC module of the OpenSPARC T2. They monitored signals between all modules at that level, which includes processor cores, L2 cache banks, memory controllers, and I/O controllers.

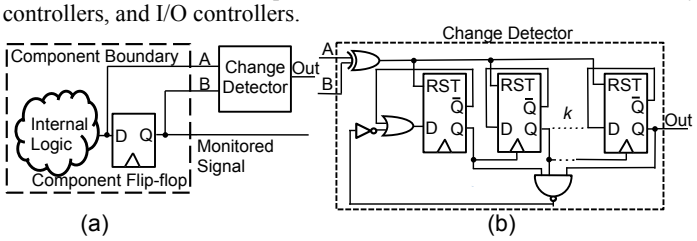

Figure A3. Change detector.

A change detector consists of a *k*-bit ripple counter that is initialized to all 1's and is reset to all 0's whenever a change in signal values is detected. Due to the short error detection latencies of QED tests, $k \approx 10$ is generally sufficient (*change window* of 1,023). When a QED test detects an error, the system is stopped and the change detector counter values are scanned out and saved. Using the recorded values in the change detectors, a reduced design is created for further analysis. A component is excluded from the analysis by BMC if, during the change window, the change detectors did not record any changes in the logic values of the component's input or output signals.

To evaluate the change detectors for reducing the design size, we used the same 92 bug scenarios in OpenSPARC T2 and ran the QED-transformed (EDDI-V and PLC) FFT and MMULT tests. With a change window of 1,023 cycles, we observed that for these tests, only the processor cores, crossbar, L2 caches, and memory controllers were part of the reduced design. Other components such as the I/O controllers could thus be removed from the design. For the 92 bugs, as shown in Fig. A4, the number of L2 cache banks (L2C) and memory controllers (MCU) also varied. For 3 of the bugs (with FFT benchmark) the design size reduced enough to eliminate a component (in each case a memory controller) from the partially instantiated designs.

We performed synthesis using the Synopsys Design Compiler with the Synopsys EDK 32 nm library to calculate the chip-level area overhead of the change detectors on OpenSPARC T2. We inserted change detectors on 1,067 signals (24,214 bits total), requiring 24,214 change detectors for the entire design. This resulted in a 1.86% chip-level area overhead. However, given that the change detectors did not reduce the number of memory controllers or caches enough to completely eliminate a partial instance (Sec. IV) for most of the bugs (Fig. A4), in this example we could have omitted the change detectors that only observe signals between those components. Then the number of signals monitored is 899 (12,734 bits total). The area overhead reduces to 0.98%. The use of change detectors on only peripheral components that see intermittent activity appears to be the most cost-effective strategy; monitoring components such as caches and memory controllers that have high utilization may not add significant value. The overhead is significantly less than the 4% overhead of reconfigurable logic for post-silicon debugging proposed by [Abramovici 06]. This also entirely avoids the use of trace buffers (and the associated area overhead), as implemented by [Chang 09, De Paula 08, 11, 12, Park 08, 09].

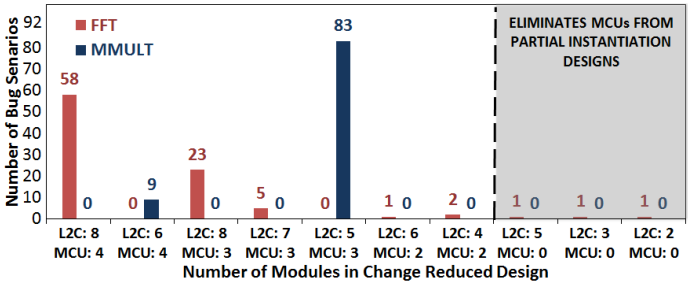

Figure A4. Change detector reduced design results for 92 bugs simulated during the FFT and MMULT benchmark tests.