# Hybrid Quick Error Detection (H-QED): Accelerator Validation and Debug using High-Level Synthesis Principles

Keith A. Campbell[1], David Lin[2], Subhasish Mitra[2,3], Deming Chen[1]

[1]Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, Urbana, IL, USA

[2]Department of EE and [3]Department of CS
Stanford University, CA, USA

## ABSTRACT

Post-silicon validation and debug challenges of system-on-chips (SoCs) are getting increasingly difficult. As we reach the limits of Dennard scaling, efforts to improve system performance and energy efficiency have resulted in the integration of a wide variety of complex hardware accelerators in SoCs. Hence, it is essential to address post-silicon validation and debug of hardware accelerators. High-level synthesis (HLS) is a promising technique to rapidly create customized hardware accelerators. In this paper, we present the Hybrid Quick Error Detection (H-QED) approach that overcomes post-silicon validation and debug challenges for hardware accelerators by leveraging HLS techniques. H-QED improves error detection latencies (time elapsed from when a bug is activated to when it manifests as an observable failure) by 2 orders of magnitude and bug coverage 3-fold compared to traditional post-silicon validation techniques. H-QED also uncovered previously unknown bugs in the CHStone benchmark suite, which is widely used by the HLS community. H-QED incurs less than 2% chip-level area overhead with negligible performance impact, and we also introduce techniques to minimize any possible intrusiveness introduced by H-QED.

## Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids – *automatic synthesis, simulation, verification*; B.7.1 [**Integrated Circuits**]: Types and Design Styles – *advanced technologies, algorithms implemented in hardware, VLSI*; B.8.1 [**Integrated Circuits**]: Performance and Reliability – *Reliability, Testing, and Fault-Tolerance*.

## General Terms

Verification, human factors, design, algorithms, experimentation.

## Keywords

Post-silicon validation, high-level synthesis, accelerators, C simulation, signature generation, timing errors, logic bugs.

## 1. Introduction

During post-silicon validation and debug (*PSV*), manufactured ICs are tested in actual system environments in order to detect and fix design flaws (bugs). Bugs can be broadly classified as: 1) *electrical bugs* that are caused by interactions between a design and its electrical state (this category includes timing errors); and, 2) *logic bugs* that are caused by design errors. This paper focuses on both electrical bugs and logic bugs. PSV is crucial because traditional pre-silicon verification is inadequate for finding "difficult" logic and electrical bugs. However, existing PSV techniques are *ad hoc*, and the associated costs are rising faster than design costs [Friedler 14, Yerramilli 06, Keshava 10]. PSV challenges are further exacerbated by significant increase in design complexity (e.g., hardware accelerators, multiple processor cores, on-chip uncore components such as cache controllers and memory controllers, power management units) to improve energy efficiency and performance of SoCs despite slowdown of silicon CMOS (Dennard) scaling.

PSV is difficult because of long error detection latencies associated with difficult bugs [Hong 10, Lin 12, 14, Reick 12]. *Error detection latency* is defined as the time elapsed from when a bug is activated to when it manifests as an observable failure (e.g., incorrect result, system crash, deadlock, livelock, or exception). As reported in [Hong 10, Lin 12, 14, 15], error detection latencies of difficult bugs during PSV can exceed several millions or billions of clock cycles. It is extremely difficult to trace that far back into the history of system operation for debug purposes. Moreover, such long error detection latencies can also mask bug effects resulting in low bug coverage during PSV.

In this paper, we present the *Hybrid Quick Error Detection (H-QED)* technique to overcome PSV challenges for non-programmable hardware accelerators in SoCs. Such accelerators implement a pre-defined set of functions and are not programmable using software (unlike processor cores or software-programmable accelerators such as GPUs). H-QED is inspired by the QED technique for PSV [Hong 10, Lin 12, 14, 15]. Since QED is (mostly) implemented in software, the error detection latencies of bugs inside hardware accelerators can be very long (e.g., bounded by long execution times of hardware accelerators). H-QED builds on advances in High-level synthesis (*HLS*) [Martin 09, Rupnow 11] to overcome this challenge by automatically embedding small hardware structures inside hardware accelerators. H-QED simultaneously improves error detection latencies and coverage of logic and electrical bugs inside hardware accelerators. H-QED is compatible with QED. By combining H-QED with QED, we provide a systematic solution for PSV of SoCs consisting of processor cores, uncore components, software-programmable accelerators, and hardware accelerators.

The current trend for designing hardware accelerators is to specify them using high-level languages (e.g., C/C++, SystemC or domain-specific languages), and automatically translate the high-level specifications into RTL designs using HLS tools. This methodology can reduce code size by up to 10X (C vs. RTL code) and improve simulation speed by 100X (C-level vs. RTL simulation) [Wakabayashi 00, 04]. The success of HLS for creating accelerators is also demonstrated by a multitude of commercial HLS tools (e.g., Stratus from Cadence, Catapult-C from Calypto, and Vivado HLS from Xilinx).

To the best of our knowledge, H-QED presents the first work that integrates HLS to overcome PSV challenges of SoCs. The input to H-QED is a specification of the hardware accelerator using a high-level language (C/C++ in this paper). H-QED then automatically creates an accelerator with built-in features for hybrid checking using hardware and software techniques. The checking techniques operate in a highly coordinated manner as follows:



*DAC '15,* June 07 - 11, 2015, San Francisco, CA, USA
Copyright 2015 ACM 978-1-4503-3520-1/15/06…$15.00
http://dx.doi.org/10.1145/2744769.2744853

1. During design, our H-QED-aware HLS engine automatically creates an *H-QED-enabled accelerator* from the input specification. Each H-QED-enabled accelerator contains small hardware structures for special hardware signatures that capture the execution behavior of the accelerator during PSV.
2. During design, our H-QED-aware HLS engine also creates a *software version* of the accelerator by inserting additional instructions in the specification of the accelerator (C/C++ source code in this paper). When this software version is executed on a processor (not necessarily on the same SoC being validated), the additional instructions capture the execution behavior of the software version using special software signatures.
3. During PSV, the hardware signatures generated by the hardware accelerator are stored in dedicated on-chip memory[1]. At the end of a PSV run, these signatures are compared against the software signatures obtained from the execution of the software version. We guarantee that, under bug-free situations, the hardware signatures exactly match the software signatures (for the same inputs, obviously). Thus, a mismatch indicates detection of errors (caused by bugs). Note that, the execution of the software version is decoupled from the PSV run.

We demonstrate the effectiveness and practicality of H-QED by showing that: 1) H-QED enables 2 orders of magnitude improvement in error detection latencies for both electrical bugs and logic bugs vs. PSV techniques using end-result-checks that compare accelerator outputs against known correct outputs; 2) H-QED improves electrical bug (timing error) coverage by up to 3X compared to PSV techniques using end-result-checks; 3) H-QED uncovered 2 previously unknown logic bugs in the widely-used CHStone HLS benchmark suite [Hara 09]; 4) H-QED incurs less than 2% SoC-level area overhead, and negligible power and performance costs; 5) H-QED does not require any failure reproduction[2] or low-level simulation (e.g., RTL or netlist) to detect bugs; and, 6) by operating hardware accelerators in *native mode* (similar to normal system operation) and by using dedicated on-chip memory to store hardware signatures during PSV, H-QED minimizes intrusiveness (i.e., incorporation of H-QED continues to detect bugs that are detected by traditional PSV techniques).

The rest of the paper is organized as follows: Section 2 presents our H-QED technique. Section 3 presents our experimental results. Related work is discussed in Section 4. We conclude in Section 5 and discuss ideas for future work.

# 2. H-QED

## 2.1 H-QED Overview

Fig. 1 shows an SoC-level view of our H-QED-enabled accelerators. The SoC typically consists of processor core(s), accelerator(s) (H-QED-enabled in our case), and uncore components. The inputs and outputs of the H-QED-enabled accelerators are supplied by the processor cores inside the SoC. During PSV, the H-QED-enabled accelerators generate hardware signatures that are saved in dedicated on-chip memories (Fig. 1(a)). Fig. 2 shows the overall H-QED flow. It takes as input the high-level design of a hardware accelerator (C/C++ source code in this paper) and produces the RTL implementation of the H-QED-enabled accelerator. This H-QED-enabled accelerator contains embedded hardware structures (*Hardware Signature Generation* in Fig. 2) that generate a sequence of hardware signatures during a PSV run. Care must be taken to ensure that the hardware signatures inside the accelerator do not cause excessive intrusiveness during PSV, e.g., by stalling the accelerator or by interfering with its input and output data traffic. Excessive intrusiveness can prevent activation of bugs inside the accelerator during PSV. In an effort to minimize intrusiveness, H-QED stores hardware signatures in dedicated on-chip memory with dedicated communication channels (Fig. 1(a)). The costs associated with this storage are reported as part of H-QED area costs. It may be possible to minimize signature storage costs (while controlling intrusiveness) by streaming hardware signatures to off-chip memory using JTAG ports.

The H-QED flow also generates a functionally-equivalent *software version* of the hardware accelerator. This software version is compiled from the same C/C++ source code as the hardware accelerator. It is augmented with instructions to generate software signatures when the software version is executed on a processor (*Software Signature Generation* in Fig. 2).

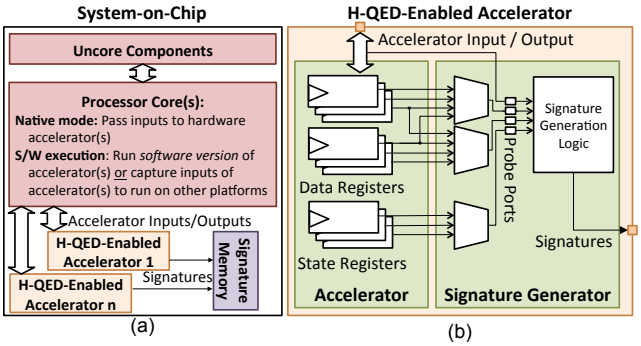


**Figure 1: H-QED-enabled accelerators inside an SoC. (a) SoC-level view, and (b) block diagram of an H-QED-enabled accelerator showing the accelerator and the signature generator.**

During PSV, the sequence of hardware signatures (stored in on-chip memory) is collected at the end of a PSV run. Note that, during the PSV run, the hardware accelerator (and the overall SoC) operates in its native mode. Bugs inside the accelerator are thus expected to be activated during the PSV run.

Next, the software version is executed on a processor; strategies to provide the same inputs to the software version as the hardware accelerator are discussed later in this section. The software version generates a sequence of software signatures during its execution. Bugs may or may not be activated during the execution of the software version. Hence, the execution of the software version can be totally decoupled from the PSV run. For example, the user may choose to execute the software version on a different hardware platform vs. the PSV run.

The sequence of hardware signatures obtained from the PSV run is compared with the sequence of software signatures obtained from the execution of the software version; any mismatch indicates bug detection. Since the execution of the software version and the subsequent signature comparisons are totally decoupled from the PSV run, we minimize possible intrusiveness introduced by H-QED.

In order to ensure that the hardware signatures match the software signatures (under bug-free conditions), we must ensure that the software version receives the same inputs as the hardware accelerator. This can be accomplished in several ways. Two examples include:

1. After a test is executed during a PSV run (in native mode), the SoC may be configured so that the hardware accelerator is disabled and the software version is swapped in. Next, the same test can be executed to generate software signatures. Note that, this is different from failure reproduction because we don't require bugs to be activated (or reproduced) during the second run.

[1] It is possible to stream out the signatures to off-chip memory using on-chip memory interfaces or JTAG ports.

[2] Failure reproduction involves returning the system to an error-free state and re-running the system with the exact input stimuli (e.g., test instructions; test inputs; and operating conditions such as voltage, temperature, and frequency), and is difficult due to Heisenbug effects [Gray 85].

2. After a test is executed during a PSV run (in native mode), the same test may be run again with the SoC (and the test) configured to capture (and store) accelerator inputs at pre-defined memory locations. Using these captured accelerator inputs, the software version can then be executed either on the embedded processor core of the SoC being validated, or on some other processors to generate software signatures. Similar to earlier discussions, we don't require bugs to be activated (or reproduced) after the first PSV run.

We built our framework on top of LLVM [Lattner 04, 11] using a common LLVM Internal Representation (LLVM-IR) to drive the generation of the H-QED-enabled hardware accelerator and the corresponding software version.

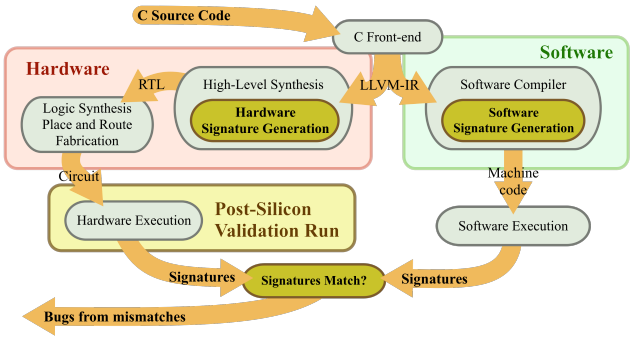


**Figure 2. H-QED flow.**

## 2.2 Hardware Signature Generation

Consider the input pseudo-code shown in Listing 1. It defines two arrays Z and B with element addresses z_ptr and b_ptr. It also defines a single basic block bb1 (a basic block is a basic building block in LLVM-IR representing a piece of code with only one control entry point and only one exit point) that has already been scheduled to execute in hardware across three consecutive clock cycles (bb1.0, bb1.1, bb1.2). Hardware corresponding to this code is shown in Fig. 3. The datapath is controlled by an FSM, where each scheduled clock cycle corresponds to one state in the FSM, and cycle transitions are controlled by the FSM state transitions. In this example, bb1.0, bb1.1, and bb1.2 represent three different FSM states.

```
int Z[100], B[200];
z_ptr = address_of(Z[1]); b_ptr = address_of(B[10])
bb1.0:  z = load mem(z_ptr)
bb1.1:  a = x + y
        b = a × z
bb1.2:  store b → mem(b_ptr)
```

**Listing 1: Input Pseudo-code Example.**

The first step in hardware signature generation is to determine the *probe schedule*: for each clock cycle, we determine which variables should be probed so that these variables contribute to the hardware signature. We perform probing for three kinds of hardware components: memory inputs/outputs, data registers (registers that store intermediate data, such as x and y in Fig. 3), and control state registers storing FSM states. These components then provide probe signals that drive the hardware signature generation logic (consisting of an XOR function and an LFSR). We refer to the physical wires carrying these probe signals as probe ports (Fig. 3).

Since the number of data register bits can be high, we use two strategies to minimize register probe ports: ignore "temporary" variables, and share ports through multiplexers. Both strategies start with variable lifetime analysis: for each variable in the input code, we determine the states in which it is alive. We define *non-temporary lifetime* as one that crosses more than one state transition, at least one of which is a basic block boundary (i.e., the variable is alive across more than one basic blocks). Any variable that does not satisfy these criteria is not probed. In our example, variables x and y meet our criteria, while z, a, and b do not (assuming they are not used in a subsequent basic block).

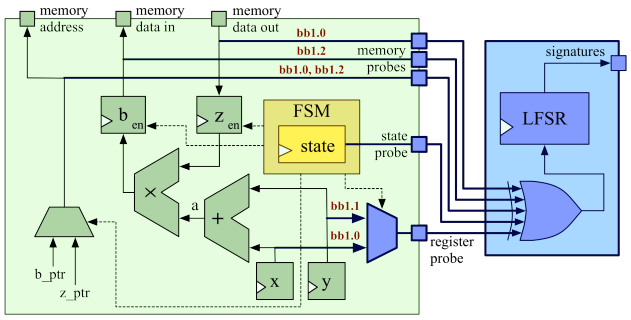


**Figure 3: H-QED-enabled accelerator with hardware signature generation.**

Our scheduler attempts to schedule a probe for a variable in its *use* state. For example, state bb1.1 is a use state of variables x, y, and z because these variables are accessed ("used") in this state. To allocate a minimum number of register probe ports, our algorithm attempts to create a feasible probe schedule using a single register probe initially starting from the first use state. For example, we first schedule y to be probed in state bb1.1; as a result, we are unable to schedule a probe for x in that same state (since there is only a single register probe). To resolve this problem, we probe x in the predecessor state bb1.0 where it is alive as well, generating a multiplexer to share the register probe port. If scheduling fails, we attempt to schedule again with an additional register probe.

We connect each control state register to its own dedicated probe port, allowing us to generate signatures from the control FSM.

We also probe memory inputs and outputs in all states where they are alive and used; i.e., they are used to transfer valid data. In our example, we perform a load in state bb1.0 and a store in state bb1.2. Hence, the memory "data out" is probed in state bb1.0 and the memory "data in" is probed in state bb1.2 as annotated in Fig. 3. The memory address port is probed in both bb1.0 and bb1.2.

Every annotated probe port in Fig. 3 has a MUX associated with it. The MUX output drives the port to logic 0 when it is not probed. The select signals of the MUX are derived from the corresponding states annotated in Fig. 3. All probe ports are fed into an XOR function, which reduces the number of input bits and produces outputs that match the size of the LFSR. We design the XOR function as an XOR tree so it can reduce *n* inputs down to *m* outputs through partitioning *n* inputs into *m* groups and reducing each group into a single bit. The LFSR can output one-bit of hardware signature periodically (the number of cycles in the period can be configured using a counter).

Since memory addresses are included in H-QED signatures, we must ensure that the signatures of memory addresses produced by the hardware accelerator match that of the software version (details in Sec. 2.3).

## 2.3 Software Signature Generation

For H-QED software signature generation, our HLS engine generates a probe schedule file together with hardware memory addresses assigned by the HLS engine, shown in Listing 2 for our earlier example. For each state, the probe schedule provides a state encoding (e.g., 1 for state bb1.0, 2 for bb1.1) as well as a list of variables that are probed in that state. The hardware memory address section provides the statically-assigned address for each memory variable.

Given a probe schedule and hardware memory addresses, software signature generation works as follows. For each state (e.g., bb1.0, bb1.1, bb1.2), we look up the variables probed in that state, and insert into the software an XOR function of the probed variables

and the state encoding, emulating the hardware XOR function (e.g., the one in Fig. 3).

```
// signature output schedule
bb1.0: 1, z_ptr, z, x
bb1.1: 2, y
bb1.2: 3, b_ptr, b
// hardware memory addresses
Z: hardware base address: 0x1000
B: hardware base address: 0x2000
```

**Listing 2: Probe schedule and hardware memory addresses.**

The memory addresses used by the hardware accelerator are not the same as that of the software version. On the hardware side, the address space is mapped by HLS into memory blocks, one for each statically-allocated array. It is desirable to partition the address such that one partition of bits selects the memory block; the remaining bits then select a word within the memory block. Each memory block can also have customized word size to optimize throughput and minimize area cost. On the software side, the statically-allocated variables are packed by a compiler into a static memory segment of the generated executable, typically with the goal of minimizing memory usage. Moreover, all of the variables have the same word size.

For software signatures to match the hardware ones, we implemented a conversion function. This function converts each address used in the software version to the corresponding hardware address before being fed to the XOR function for software signature generation. This is possible because our HLS tool produces a mapping to indicate how the variables are mapped to memory addresses on the hardware side (shown in Listing 2). The additional code for this address conversion on the software side does not do any harm to bug detection; this is because bugs are not required to be activated or reproduced during the execution of the software version.

For an address variable *addr* (e.g., z_ptr or b_ptr) in the software address space, we pass it through the software to hardware address conversion function. First, this conversion function determines which variable *addr* points to (in this case, Z or B) and the address offset into that variable (e.g., 1 for Z and 10 for B). Next, the converter looks up the variable (Z or B) in the hardware memory addresses section of the probe schedule file, and returns the corresponding hardware address with the appropriate offsets. This hardware address drives the XOR function.

The outputs of the XOR function are passed to an LFSR function, which imitates the hardware LFSR. The LFSR function also mimics exactly the signature output interval of the hardware LFSR, enabling the software to generate signatures that match the hardware. Listing 3 shows the resulting H-QED-enabled software version for our example.

```
z = load z_ptr
software_lfsr(1 ⊕ addr_convert(z_ptr) ⊕ z ⊕ x)
a = x + y
b = a × z
software_lfsr(2 ⊕ y)
store b → b_ptr
software_lfsr(3 ⊕ addr_convert(b_ptr) ⊕ b)
```

**Listing 3: H-QED-enabled software version.**

## 2.4 Binding to Minimize Area

Efficient operator and data register sharing is crucial for minimizing H-QED area costs. We implemented a binding engine which aggressively shares operators among instructions and registers among variables, as long as their lifetimes do not overlap, in order to minimize area costs. However, such sharing introduces MUXes. Therefore, we developed heuristics to optimize mux widths for binding by reusing hardware components, wires, and corresponding mux inputs that have already been allocated (we call it *zero-cost binding*). We use a greedy heuristic to exploit zero cost binding opportunities. Instructions and variables are bound to hardware components iteratively. During each iteration for instruction or variable binding, we choose the binding solution with the lowest area cost. We also attempt to share existing probe ports at the register outputs through zero cost binding solutions.

# 3. Results

To demonstrate the effectiveness and practicality of H-QED, we ran a series of simulation and FPGA-based emulation experiments to collect data for area and clock period overheads, and error detection latencies, and coverage estimates for logic and electrical bugs. We used all 12 benchmarks from the CHStone [Hara 09] and 15 benchmarks from the PolyBench [Pouchet 12] benchmark suites. The benchmarks we selected from PolyBench are ones that can be implemented with fixed-point operations because our framework does not support floating-point operations yet.

## 3.1 Hardware Area and Delay Costs

To determine the area and delay costs of adding H-QED signature generation logic to an accelerator, we performed HLS with and without H-QED. We then performed logic synthesis using the Synopsys Design Compiler 2013-12.sp1, mapping to the 45nm ARM standard cell library, and targeting maximum clock frequency. We calculated two area overhead numbers relative to the baseline design without H-QED: one for the accelerator only, and the other for the entire SoC. We calculated the SoC-level (i.e., chip-level) area overhead by using the fraction of the SoC occupied by accelerators in a representative SoC: the Renesas R-Mobile U2 SoC used in many mobile applications [Fujigaya 13]. The fraction is 21.5% for this chip. The SoC-level area and clock period overheads are shown in Fig. 4. Results show a mean SoC-level area cost of 1.8% assuming all the benchmarks follow the same accelerator/SoC area ratio. We observe no clock period overhead on average. The mean accelerator-level area cost is 8.3% across all benchmarks.

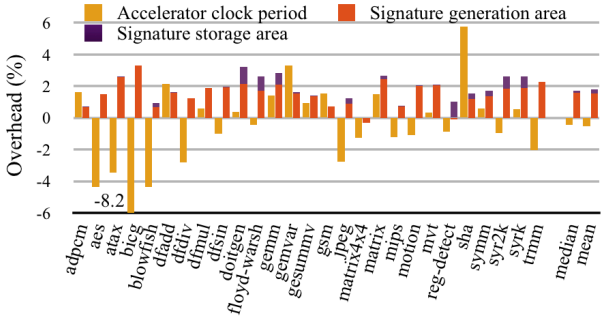


**Figure 4: SoC-level area and performance overheads.**

## 3.2 Logic Bugs

To evaluate the effectiveness of H-QED in detecting logic bugs, we considered bugs in the current and past versions of CHStone [Hara 09], as well as bugs in our HLS engine itself (Table 1). Each bug may have multiple instances, or places in the code where it is activated. For example, there are multiple instances of bit shifts by out-of-bounds amounts in the motion benchmark. The "Benchmarks/Bug Location" column shows the benchmark where the bug is found, as well as the associated C source file and the line. The "Bug Description (# of Instances)" column provides a concise description of the bug with its number of instances in the benchmark. The "Bug Source" column indicates whether it is a bug in the CHStone benchmark suite or a bug in our HLS engine. The "ERC latency" column shows the error detection latency for each bug instance (in terms of the number of clock cycles of operation of the hardware accelerator) using the end-result-check method (where the end results of the accelerator are compared to known correct results

of the accelerator). An "x" in the field indicates that the bug instance is not detected by ERC. The "H-QED latency" column shows the error detection latency value for each bug instance using H-QED. Note that, H-QED naturally includes ERC since H-QED also collects the signature of the accelerator outputs.

Table 1 demonstrates the following results:

1. H-QED improves bug coverage, i.e., it is capable of detecting bugs that escape ERC.
2. For bugs detected by both H-QED and ERC, the error detection latency is significantly shorter (up to three orders of magnitude) using H-QED.
3. H-QED discovered 2 previously unknown bugs (3 bug instances) in the 1.10 version of CHStone (motion in Table 1). H-QED also identified a new bug in the 1.11 release (mips in Table 1). All four new bug instances detected by H-QED were confirmed with the CHStone authors.

We encountered a bug in adpcm, which is detected by ERC (and also by H-QED since H-QED includes ERC). This bug results in a highly repeatable behavior (omission of a variable assignment operation), which makes it very easy to reproduce this bug over multiple runs for debug purposes.

**Table 1. Logic bug detection using H-QED. Previously unknown bugs are highlighted in bold.**

| Benchmark/ Bug Location(s) | Bug description (# of instances) | Bug source | ERC Latency | H-QED Latency |
|---|---|---|---|---|
| gsm/lpc.c 157-158 * | Attempting to access out-of-bounds array index (1) | CHStone | x | 77 |
| **mips/mips.c 255** | Read of uninitialized element of array (1) | CHStone | x | 23 |
| mips/mips.c: 132-135 | Read off the end of array (1) | CHStone | x | 110 |
| motion/mpeg2.c: 225-226 | Read off the end of array (1) | CHStone | 90 | 10 |
| **motion/getbits.c 144, 155, 160 *** | Bit shifts by out-of-bounds amounts (3) | CHStone | 45, x, x | 45, 105, 91 |
| jpeg/huffman.c 86 | Nonzero global register initializers ignored. (1) | HLS tool | 838k | 179 |
| jpeg/huffman.c 45 | RAM blocks for global arrays not initialized (3) | HLS tool | x, x, x | 249, 349, 349 |

*There are multiple instances of various out-of-bounds bugs in the original C code, but the compiler removed some of these bug instances away. We show the instances that were retained.

## 3.3 Electrical Bugs

In this section, we present a study of timing errors as representative electrical bugs. To evaluate the effectiveness of H-QED for detecting such electrical bugs, we injected timing errors into each of our benchmark designs. Such a process begins with running each benchmark through HLS with H-QED, feeding the output RTL code to Design Compiler, and compiling for timing optimization. To identify timing error activations, we use an approach similar to the "ground truth" method in [Gao 12]: for each flip-flop in the logic netlist, add a duplicate flip-flop connected to the same "D" input, but with an additional half-cycle delay on the input. This flip-flop's "Q" output is left unconnected as it is used only to trigger reports of timing violations (by a timing simulator) while the original flip-flops maintain the error free execution of the benchmark. We run timing simulations with the modified netlist and compiled the timing violations reported into a set of (flip-flop, cycle) pair, referred to as "injection candidates." We selected a random subset of these candidates with size $n$ (we set $n$ = 500) to use in our error injection experiments. Starting again from the original netlist, we applied another netlist transform, which inserts XOR gates at the "D" input of flip-flops corresponding to the selected injection candidates. We added additional logic to control each XOR gate, enabling error injection at a specific cycle. We mapped the transformed netlist to an FPGA (Altera Stratix III) for emulation purposes, and performed $n$ full execution runs for each benchmark, injecting one error from the selected "injection candidates" during each run (bit flip at the input of the given flip-flop at the given cycle).

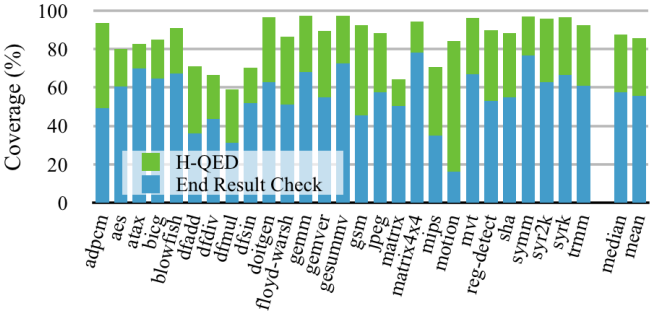


**Figure 5: Timing error detection coverage.**

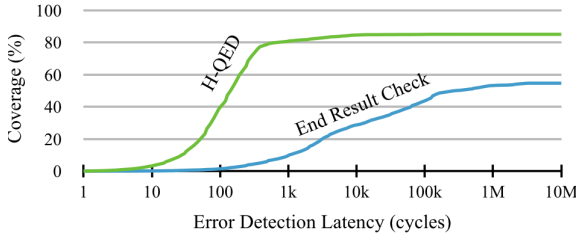


**Figure 6: Overall timing error coverage as a function of error detection latency.**

Timing error coverage (number of errors detected divided by the number of errors injected) is presented in Fig. 5, including both masked (errors that do not propagate to accelerator outputs so they are invisible externally) and unmasked errors (errors that propagate to the primary outputs and affect accelerator results). Note that, the unmasked timing error detection coverage is 100% with H-QED (i.e., we detect all unmasked errors). The overall error detection latency distribution is shown in Fig. 6. We observed mean timing error detection coverage for H-QED of 85.8% compared to 55.8% for the end result check, resulting in 3.1x improvement (i.e., reduction) in undetected timing errors. We also observed a mean error detection latency of 705 cycles for H-QED, compared to 124,490 cycles for end result check, resulting in 176x improvement (i.e., reduction) in error detection latency.

# 4. Related Work

As discussed in Sec. 1, H-QED represents the first PSV work that integrates HLS to overcome PSV challenges for hardware accelerators in SoCs.

PSV techniques that target processors (e.g., [Adir 11, Wagner 08] and others) are inadequate for bugs inside accelerators. While H-QED is inspired by QED [Hong 10, Lin 12, 14, 15], there are important differences between H-QED and QED (discussed in Sec. 1). A combination of QED and H-QED provides an integrated and systematic solution for quickly detecting bugs inside processor cores, programmable accelerators, hardware accelerators, and uncore components in complex SoCs.

Although H-QED may appear to be similar to tracing techniques used in PSV (e.g., using trace buffers or system memory [Abramovici 08, ARM CoreSight, Ko 08, Park 09, 10], there are important differences: 1. H-QED systematically collects signatures unlike tracing techniques that are often ad-hoc or based on heuris-

tics; 2. H-QED doesn't require extensive low-level (e.g., RTL) simulation; 3. H-QED doesn't require designer-crafted assertions; 4. H-QED enables very short error detection latencies and high bug coverage unlike tracing techniques that become ineffective for difficult bugs with long error detection latencies.

H-QED is distinct from fault-tolerant computing techniques for processors (e.g., using watchdog processors, DIVA, multi-threading and signature techniques for duplex systems [Austin 99, Lu 82, Mahmood 88, Mukherjee 02, Saxena 00, Smolens 04, Sogomonyan 01]). Many of these techniques only check the register values as defined by the Instruction Set Architecture (ISA). In contrast, H-QED is effective for arbitrary hardware accelerators created using the HLS and automatically identifies signals to check in the resulting designs. Unlike time redundancy and cycle stealing techniques for enhancing reliability of designs created using HLS [Karri 93, Mitra 00, Saxena 98], H-QED utilizes unique aspects of the PSV environment (where the generation of software signatures after a PSV run is acceptable vs. reliability techniques that focus on quick error recovery) to minimize area/performance costs and intrusiveness.

Given a high-level specification and a design produced by HLS (referred to as an *implementation*), there is a large class of techniques that check if the implementation is equivalent to the high-level specification, often relying on formal techniques [Feng 06, Fujita 05, Mathur 09]. The goal is to detect bugs in the implementation that are caused by the HLS tool. However, equivalence checking techniques cannot detect bugs that are in the high-level specification itself. In contrast, H-QED detects bugs in the high-level specification (e.g., the C source code in this paper) as well as bugs in the implementation caused by the HLS tool.

## 5. Conclusion

H-QED utilizes HLS principles for quickly detecting bugs inside hardware accelerators in SoCs. Our results demonstrate the effectiveness and practicality of H-QED: up to 2 orders of magnitude improvement in error detection latency, up to 3-fold improvement in coverage, less than 2% chip-level area overhead, and with negligible performance overhead. Furthermore, H-QED also discovered previously unknown bugs in the widely-used CHStone HLS benchmark suite. Through hybrid hardware/software signatures, H-QED minimizes intrusiveness during PSV. Thus, the combination of QED and H-QED provides a systematic approach to PSV of complex SoCs consisting of processor cores, uncore components, programmable accelerators, and hardware accelerators. Future directions related to H-QED include: 1. Use of H-QED for a wide variety of high-level descriptions beyond C and C++ (e.g., various domain-specific languages). 2. Use of H-QED for programmable accelerators; 3. Integration of H-QED with formal analysis tools for automatic debug.

## References


[Abramovici 08] M. Abramovici, "In-System Silicon Validation and Debug," *IEEE Design & Test of Computers*, Vol. 25, No. 3, pp. 216-223, May 2008.

[Adir 11] Adir, A., *et al.*, "Threadmill: A Post-Silicon Exerciser for Multi-Threaded Processors," *DAC,* 2011.

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

[Austin 99] Austin, T. M., "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. IEEE/ACM MICRO*, pp. 196-207, 1999.

[Bohr 09] "The New Era of Scaling in an SoC World," *Proc. ISSCC* 2009.

[C11] International Organization for Standardization, "ISO/IEC 9899:201x - Programming languages - C," Dec 2011.

[Feng 06] Feng, X., and A. J. Hu, "Early Cutpoint Insertion for High-Level Software vs. RTL Formal Combinational Equivalence Verification," *DAC,* pp. 1063-1068, 2006.

[Friedler 14] Friedler, O., *et al.*, "Effective Post-Silicon Failure Localization Using Dynamic Program Slicing," *DATE*, pp. 1-6, 2014.

[Fujigaya 13] M. Fujigaya, *et al.*, "A 28nm High-κ Netal-Gate Single-Chip Communications Processor with 1.5GHz Dual-Core Application Processor and LTE/HSPA+-Capable Baseband Processor," *Proc. Intl. Solid State Circuits Conf.*, pp. 156-157, Feb. 2013.

[Fujita 05] Fujita, M., "Equivalence Checking Between Behavioral and RTL Descriptions with Virtual Controllers and Datapaths," *ACM Trans. Design Automation Electronic Systems*, Vol. 10, No. 4, pp. 610–626, Oct. 2005.

[Gao 12] M. Gao, *et al*., "On Error Modeling of Electrical Bugs for Post-silicon Timing Validation," *ASPDAC*, pp. 701-706, 2012.

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

[Hara 09] Y. Hara, *et al*., "Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis", *Information Processing,* 2009.

[Ho 09] Ho, R. C., *et al*., "Post-Silicon Debug Using Formal Verification Waypoints," *Proc. Design Validation Conf.*, 2009.

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

[Karri 93] Karri, R., A. Orailoglu, "High-Level Synthesis of Fault-Secure Microarchitectures," *DAC,* pp. 429-433, 1993.

[Lattner 04] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *CGO,* pp.75-86, 2004.

[Lattner 11] C. Lattner "LLVM and Clang: Advancing Compiler Technology", Keynote Talk*, FOSDEM,* 2011.

[Lin 12] Lin, D., *et al.*, "Quick Detection of Difficult Bugs for Effective Post-Silicon Validation," *DAC*, pp. 561-566, 2012.

[Lin 14] Lin, D., *et al.*, "Effective Post-Silicon of System-on-Chips Using Quick Error Detection," *IEEE Trans. CAD*, Vol. 33, No. 10, pp. 1573-1590, Oct. 2014.

[Lin 15] Lin, D., *et al.*, Quick Error Detection Tests with Fast Runtimes for Effective Post-Silicon Validation and Debug, *DATE*, 2015.

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

[Mathur 09] Mathur, A., *et al*., "Functional Equivalence Verification Tools in High-Level Synthesis Flows," *Proc. IEEE Design and Test of Computers*, pp. 88-95, 2009.

[Mahmood 88] Mahmood, A., and E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors – A Survey," *IEEE Trans. Computers*, Vol.37, No.2, pp. 160-174, Feb. 1988.

[Mitra 00] Mitra, S., N. R. Saxena and E. J. McCluskey, "Fault Escapes in Duplex Systems," *Proc. IEEE VLSI Test Symp*., pp. 453-458, 2000.

[Mitra 10] Mitra, S., S.A. Seshia, and N. Nicolici, "Post-Silicon Validation Opportunities, Challenges and Recent Advances," *DAC*, pp. 12-17, 2010.

[Martin 09] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design & Test of Computers*, Vol: 26, Issue:4, 2009.

[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. CAD*, pp. 1545-1558, Oct. 2009.

[Park 10] Park, S.-B., *et al.*, "BLoG: Post-Silicon Bug Localization in Processors Using Bug Localization Graph," *DAC,* pp. 368-373, 2010.

[Pouchet 12] L. N. Pouchet. PolyBench/C 3.2. http://www.cse.ohio-state.edu/~pouchet/software/polybench/

[Reick 12] Reick, K., "Post-Silicon Debug," DAC Workshop on Post-Silicon Debug: Technologies, Methodologies, and Best-Practices. *DAC*, 2012.

[Rupnow 11] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A Study of High-Level Synthesis: Promises and Challenges," Proc. IEEE Intl. Conf. on ASIC, pp. 1102-1105, 2011.

[Saxena 00] Saxena, N. R., S. Fernandez-Gomez, W. J. Huang, S. Mitra, S.-Y. Yu and E. J. McCluskey, Saxena, N. R., S. Fernand, Online Testing in Adaptive and Configurable Systems, *IEEE Design and Test of Computers*, Vol. 17, No. 1, pp. 29-41, Jan.-Mar. 2000.

[Saxena 98] Saxena, N. R., and E. J. McCluskey, "Dependable Adaptive Computing Systems,c *Proc. IEEE Systems, Man, and Cybernetics Conf.,* pp. 2172-2177, 1998.

[Smolens 04] Smolens, J. C., *et al.*, "Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth," *Proc. ACM ASPLOS*, pp. 224-234, 2004.

[Sogomonyan 01] Sogomoyan, E. S., *et al.*, "Early Error Detection in System-on-Chip for Fault-Tolerance and At-Speed Debugging," *Proc. IEEE VLSI Test Symp.*, pp. 184-189, 2001.

[Wagner 08] Wagner, I., and V. Bertacco, "Reversi: Post-Silicon Validation System for Modern Microprocessors," *ICCD* 2008.

[Wakabayashi 00] Wakabayashi, K., and T. Okamoto, "C-based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective," *Proc IEEE Trans. CAD,* Vol. 19, No. 12, pp. 1507-1522, Dec. 2000.

[Wakabayashi 04] Wakabayashi, K., "C-Based Behavioral Synthesis and Verification Analysis on Industrial Design Examples," *Proc. IEEE/ACM Asia and South Pacific Design Automation Conf.*, pp. 344-348, 2004.

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

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