# LOW-COST ERROR DETECTION THROUGH HIGH-LEVEL SYNTHESIS

BY

KEITH A. CAMPBELL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Deming Chen

# ABSTRACT

System-on-chip design is becoming increasingly complex as technology scaling enables more and more functionality on a chip. This scaling and complexity has resulted in a variety of reliability and validation challenges including logic bugs, hot spots, wear-out, and soft errors. To make matters worse, 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. Thus the challenge is to design complex, custom hardware that is efficient, but also correct and reliable.

High-level synthesis shows promise to address the problem of complex hardware design by providing a bridge from the high-productivity software domain to the hardware design process. Much research has been done on high-level synthesis efficiency optimizations. This thesis shows that high-level synthesis also has the power to address validation and reliability challenges through two solutions.

One solution for circuit reliability is modulo-3 shadow datapaths: performing lightweight shadow computations in modulo-3 space for each main computation. We leverage the binding and scheduling flexibility of high-level synthesis to detect control errors through diverse binding and minimize area cost through intelligent checkpoint scheduling and modulo-3 reducer sharing. We introduce logic and dataflow optimizations to further reduce cost. We evaluated our technique with 12 high-level synthesis benchmarks from the arithmetic-oriented PolyBench benchmark suite using FPGA emulated netlist-level error injection. We observe coverages of 99.1% for stuck-at faults, 99.5% for soft errors, and 99.6% for timing errors with a 25.7% area cost and negligible performance impact. Leveraging a mean error detection latency of 12.75 cycles (4150x faster than end result check) for soft errors, we also explore a rollback recovery method with an additional area cost of 28.0%, observing a 175x increase in reliability against soft errors.

Another solution for rapid post-silicon validation of accelerator designs is Hybrid Quick Error Detection (H-QED): inserting signature generation logic in a hardware design to create a heavily compressed signature stream that captures the internal behavior of the design at a fine temporal and spatial granularity for comparison with a reference set of signatures generated by high-level simulation to detect bugs. Using H-QED, we demonstrate an improvement in error detection latency (time elapsed from when a bug is activated to when it manifests as an observable failure) of two orders of magnitude and a threefold improvement in bug coverage 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 10% area overhead for the accelerator it validates with negligible performance impact, and we also introduce techniques to minimize any possible intrusiveness introduced by H-QED.

*To my parents, for their love and support.*

# ACKNOWLEDGMENTS

I would like to thank my advisor Prof. Chen for showing me the meaning of "brute-force" effort and for operating like a true scientist: being convinced once presented with sufficient evidence. I would like to thank my friends in the lab who have kept me company over the years: in particular Yun Heo for giving me insight into the hardware world, Ashutosh Dhar for stepping up to help me maintain our critical lab infrastructure, and Yao Chen for complimenting my ideas and treating me like a professional.

# TABLE OF CONTENTS

LIST OF ABBREVIATIONS . . . . . vii

CHAPTER 1 BACKGROUND . . . . . 1
1.1 Root Causes for Hardware Failure . . . . . 2
1.2 Root Cause Effects . . . . . 7
1.3 Error Propagation . . . . . 9
1.4 Modulo Arithmetic . . . . . 10
1.5 Execution Signatures . . . . . 12
1.6 High-Level Synthesis . . . . . 14

CHAPTER 2 RELATED WORK . . . . . 16
2.1 Hardware Reliability . . . . . 16
2.2 Post-Silicon Validation . . . . . 17

CHAPTER 3 ERROR DETECTION THROUGH MODULO-3 SHADOW DATAPATHS . . . . . 19
3.1 Method . . . . . 20
3.2 Experimental Results . . . . . 30

CHAPTER 4 VALIDATION THROUGH HYBRID SIGNATURE GENERATION . . . . . 35
4.1 Method . . . . . 37
4.2 Experimental Results . . . . . 44

CHAPTER 5 CONCLUSIONS . . . . . 52

REFERENCES . . . . . 54

# LIST OF ABBREVIATIONS

| | |
|---|---|
| ARM | Company that designs CPU cores, initially an acronym for Acorn RISC Machine |
| ASIC | Application Specific Integrated Circuit |
| BTI | Bias Temperature Instability |
| CED | Concurrent Error Detection |
| CHStone | C-based High-level Synthesis benchmark suite |
| CPU | Central Processing Unit |
| DAC | Design Automation Conference |
| DIVA | Dynamic Implementation Verification Architecture, a fault-tolerant CPU architecture |
| DMR | Double Modular Redundancy |
| DRAM | Dynamic Random Access Memory |
| ERC | End Result Check |
| FPGA | Field Programmable Gate Array |
| FSM | Finite State Machine |
| GPU | Graphics Processing Unit |
| HLS | High-Level Synthesis, also known as behavioral synthesis |
| IR | Intermediate Representation |
| ISA | Instruction Set Architecture |
| JPEG | Joint Photographic Experts Group, develops image compression standards |

| | |
|---|---|
| JTAG | Joint Test Action Group, develops on-chip instrumentation standards |
| LFSR | Linear Feedback Shift Register |
| LLVM | An open-source compiler development framework, initially an acronym for Low-Level Virtual Machine |
| MOSFET | Metal-Oxide-Semiconductor Field-Effect Transistor |
| MSB | Most Significant Bit |
| MUX | MUltipleXer |
| PSV | Post-Silicon Validation |
| QED | Quick Error Detection, detecting errors by fine-grained duplication |
| RISC | Reduced Instruction Set Computing |
| RTL | Register Transfer Level, referring to the Verilog or VHDL hardware description languages |
| SEC | Statistical Error Compensation |
| SoC | System on a Chip |
| SRAM | Static Random Access Memory |
| SSA | Single Static Assignment |
| TMR | Triple Modular Redundancy |
| VHDL | VHSIC Hardware Description Language |
| VHSIC | Very High Speed Integrated Circuit, a U.S. government program |
| XOR | Exclusive OR, addition in modulo-2 space |

# CHAPTER 1

# BACKGROUND

Designing hardware is hard. A system designer chooses a custom hardware design when a pure software solution is inadequate for power consumption and/or performance reasons. Thus problems that require a hardware solution already come with demanding power and performance constraints. With the end of Dennard scaling, improvements in power consumption and performance for CPU-based software platforms have slowed down, pushing more and more system designers to custom hardware solutions.

The result is an explosion in system complexity with increasing effort and chip area dedicated to custom hardware on SoCs. To make matters worse, designers often have additional constraints: limited time to get into a market, complex functionality demanded by that market, and limited chip area budgets due to fabrication costs.

As if this were not enough, the continuation of Moore's law scaling has resulted in new hardware reliability problems. Reliably operating billions of transistors is not easy when power "brown outs" start occurring and thermal hot spots start forming as transistors are packed closer together. Reliably fabricating smaller wires and devices is also not easy, resulting in more permanent defects. Smaller devices are more vulnerable to particle strikes, which manifest as soft errors. Physical effects cause smaller transistors to wear out, resulting in longer gate propagation delays leading to timing errors after prolonged use. All of this does not even consider that designers themselves, without needing any help from circuit physics, are more than capable of creating their own logic bugs to trip over in their complex designs.

Clearly there is a need for effective methods to manage the complexity of hardware design. High-level synthesis, also known as behavioral synthesis, is one such approach. HLS provides a bridge from the high-productivity software world to the hardware design world, enabling hardware designers to create behavioral specifications of their design in dialects of traditionally

software languages. HLS frees hardware designers from the tedious details of hardware resource allocation, scheduling, and binding, allowing them to focus on meeting design requirements and designing effective hardware algorithms. From a research point of view, starting from a behavioral specification provides the synthesis engine with richer information about the behavior and architecture of a design, enabling scheduling and binding optimization potential not possible with RTL design entry, and giving the synthesis engine more freedom to exploit this flexibility to meet multiple optimization goals.

In this thesis, I discuss my research to leverage this power of HLS to address the aforementioned hardware reliability and validation problems. In Chapter 3, I propose creating a redundant, but smaller "shadow" datapath based on modulo arithmetic to detect reliability problems in a design's main datapath. HLS is critical here because it provides a clear picture of the datapath of the design and enables effective sharing of expensive checksum computing resources. In Chapter 4, I propose the insertion of signature generation logic in a hardware design to create a heavily compressed signature stream that captures the internal behavior of the design at a fine temporal and spatial granularity. By comparing the generated sequence of signatures to a reference set generated by high-level simulation, I can detect both logic and electrical bugs in hardware designs. HLS also plays a critical role here by identifying important variables to capture and enabling the sharing of expensive signature generation logic. Before these main chapters, I will provide some background on the reliability and validation problems hardware designers face in the rest of this chapter and discuss related work in Chapter 2.

This thesis is based on my two publications in DAC 2015: "High-Level Synthesis of Error Detecting Cores through Low-Cost Modulo-3 Shadow Datapaths" [1] and "Hybrid Quick Error Detection (H-QED): Accelerator Validation and Debug using High-Level Synthesis Principles" [2].

## 1.1 Root Causes for Hardware Failure

Figure 1.1 provides an overview of the hardware engineering process, which consists of the following steps:

1. The designer writes a Verilog and/or VHDL description of the design.

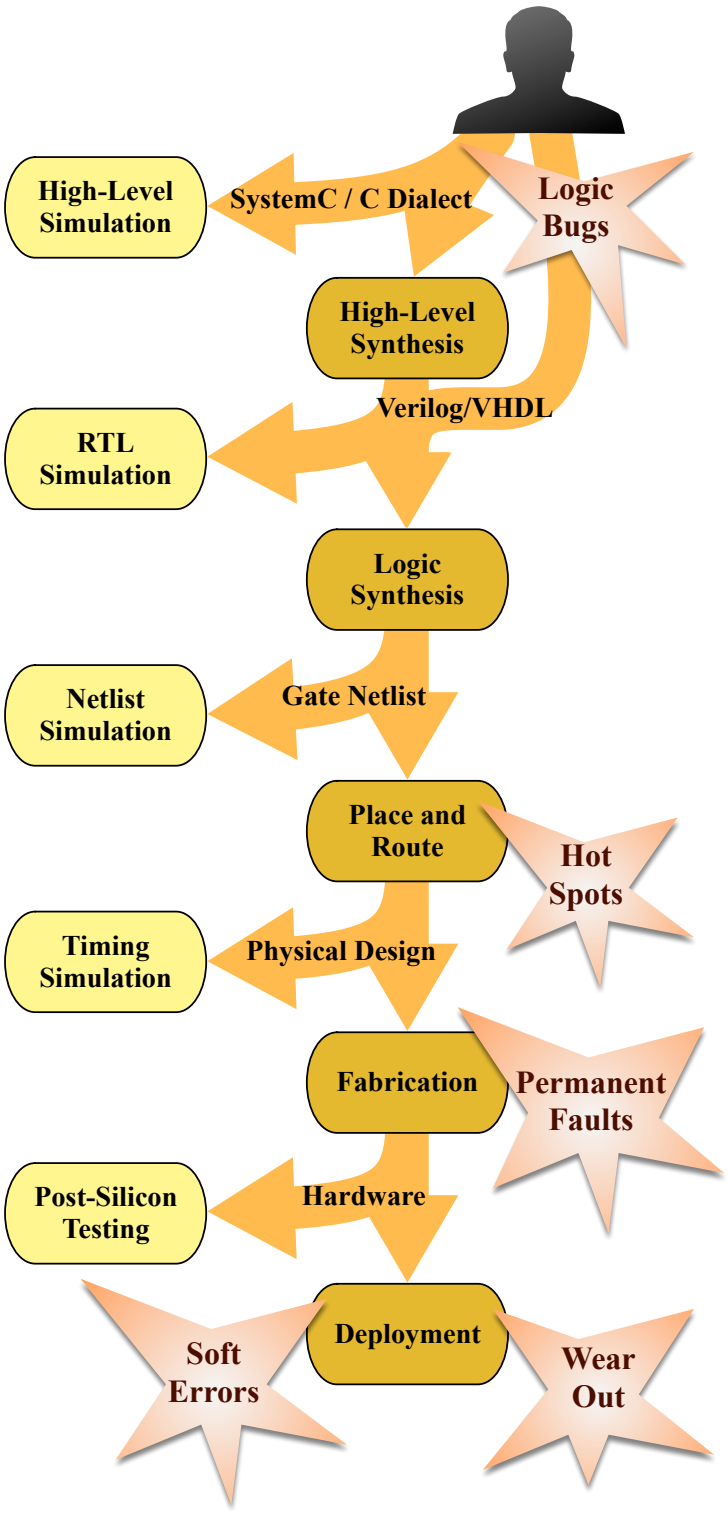



Figure 1.1: The hazards inherent in designing custom hardware

For improved productivity, the designer may also elect to specify design blocks at the behavioral level in SystemC or his HLS-tool's proprietary C dialect.

2. The designer simulates behavioral design blocks using a software compiler.

3. The designer uses a high-level synthesis tool to generate an RTL implementation of behavioral design blocks.

4. The test engineer runs the resulting RTL implementations through an RTL simulation tool.

5. The designer runs the RTL blocks through logic synthesis to generate a technology mapped gate netlist.

6. The test engineer may simulate the netlist with a netlist simulation tool. Simulation at this stage is very slow.

7. The designer runs the gate netlist through a placement and routing engine, which produces a physical design.

8. The test engineer may simulate the physical design with a chip simulation that takes wire and gate delays into account. This simulation is extremely slow.

9. The designer sends the physical design to a foundry, which fabricates the chip.

10. Test engineers test the actual hardware to verify that it meets specifications and validate that it implements the correct design.

11. Hardware that passes post-silicon testing is sent to end-users who deploy it in their systems.

Figure 1.1 also shows what can go wrong during the hardware engineering process, which we now discuss in the following subsections.

### 1.1.1 Logic Bugs

Logic bugs are mistakes that the hardware designer makes in writing the C or RTL version of a design that cause it to function in violation of the design specification. Most of these bugs are caught in high-level simulation or RTL simulation. Due to the complexities of system design it is difficult to design these tests such that they exercise every possible interaction between a design block under test and other design blocks around it. Thus some logic bugs escape high-level and RTL simulation and can make it into the physical design. Some of those bugs evade detection in post-silicon testing and survive all the way to deployment. We define two primary classes of logic bugs:

- **Deterministic logic bugs** have well defined behavior that is not compiler or synthesis tool dependent. For input languages with well defined standards, semantics that are defined in the standard are deterministic for tools that conform to the standard. An example of a deterministic logic bug is a memory copy operation for input data that simultaneously (for faster performance) copies the first half of an input array to both halves of an output array when the programmer intended to copy corresponding halves of the whole input array to the whole output array.

- **Nondeterministic logic bugs** do not have well defined behavior; the behavior can depend on the compiler or synthesis tool used, how the tool was configured, what environment the tool was run in or the design was tested in, and even other parts of the design that are seemingly unrelated; the behavior of these bugs can depend on almost anything! For input languages with well defined standards, nondeterministic semantics may be specified as resulting in "undefined behavior." An example of a nondeterministic logic bug is a read from uninitialized memory.

### 1.1.2 Hot Spots

Hot spots are regions on a chip that exceed local heat dissipation capacity and/or power supply capacity under certain operating conditions. Hot spots happen when a large amount of transistor switching activity is concentrated in a small region of a chip. An excess current demand that lasts long enough

causes voltage drops on power supply wires, resulting in longer than expected transistor delays. High power consumption exceeding the thermal dissipation capability of a region of a chip that lasts long enough results in excess heat that causes the transistors in that region, which are not designed to operate at high temperature, to slow down. The net effect is that signal propagation delays increase, leading to timing errors (defined in Section 1.2.1).

### 1.1.3 Fabrication Defects

Fabrication defects result in gates implementing the wrong logic function (or being permanently bypassed) due to wire or transistor fabrication failures. These permanent defects typically manifest as stuck-at faults: wires that are supposed to be the output of a logic gate are stuck at logic 0 or logic 1 and never change regardless of circuit input.

### 1.1.4 Soft Errors

Soft errors are caused by a particle striking a transistor with enough energy and the right timing to cause bit-flips in storage elements including flip-flops, SRAM cells, and DRAM cells. The victim transistor can be part of the storage element, or an upstream gate that propagates a resulting logic glitch. These particles are typically part of a shower of particles that results when a cosmic ray strikes the Earth's atmosphere. Thus these events are random and unpredictable in nature.

### 1.1.5 Wear Out

Like mechanical systems, MOSFETs can wear out from prolonged, heavy use. High-energy charge carriers can build up over time in a MOSFET's insulating dielectric, increasing the threshold voltage which causes the transistor to switch more slowly. Bias temperature instability (BTI) is another effect that can charge the insulating dielectric over time, although some of its effects are temporary [3]. Like hot spots, both of these problems can lead to timing errors (defined in Section 1.2.1). Unlike hot spots, these aging effects can take years to develop.

Worse problems can occur when the dielectric layer breaks down, which can result in a short that causes a permanent failure of a transistor. Another effect called electromigration causes atoms in wires to slowly "flow" downstream, thinning the wire upstream until it becomes a permanent open circuit defect [3].

## 1.2 Root Cause Effects

The effects of many of the above root causes are predictable enough that they can be modeled. For each effect, there are *activation conditions*, or conditions required for the effect to occur. More precisely, an activation condition is the condition required for an error, fault, or bug to change the internal behavior of a design. Thus if an error, fault, or bug is not activated, then it is undetectable even with perfect observability of the internal behavior of a design.

### 1.2.1 Timing Errors

Power and thermal hot spots, charge carrier injection, and bias temperature instability all result in transistors switching more slowly than they normally would. The result is that signal propagation delays along chains of gates increase, resulting in a signal taking so long to propagate from a launch flip-flop to a latch flip-flop that it misses the latch window. The result is that the wrong value can be latched at the latch flip-flop; when this occurs it is known as a timing error.

We can model this timing error as a bit flip at the latch flip-flop, given these four activation conditions for a timing error to occur along a given combinational path at a given cycle from a launch flip-flop to a latch flip-flop:

1. The sum of the arrival time of the launch flip-flop output and delays of each gate along the path must exceed the required arrival time for the latch flip-flop input.

2. The path must be *sensitized*, meaning that all logic values are such that a flip in the logic value of the launch flip-flop results in a flip along each

segment of the path up to and including the latch flip-flop.

3. The launch flip-flop toggles at the given cycle.

4. The latch flip-flop latches the wrong value. Favorable glitches may cause the latch flip-flop to latch an intermediate value that happens to be correct even though the final value arrives too late.

### 1.2.2 Stuck-at Faults

Fabrication defects result in gate outputs being stuck at either a 0 or a 1. The more dramatic wear-out problems that cause permanent defects can also have this effect. Modeling these faults is straightforward: disconnect a net from its original driver and connect it to a constant logic 0 or 1 instead. Stuck-at 0 (1) faults have one activation condition, which is that the input logic values to the gate with the stuck-at fault are such that the output should be 1 (0). The result is an internally detectable deviation in the behavior of a design.

### 1.2.3 Soft Errors

Soft errors cause random logic values to be injected into storage elements of a design, overwriting the previous value. For this event to be internally observable, the activation condition is that the value injected must differ from the value that would otherwise be latched at the storage element at the time of injection. Thus we model these events as random bit-flips at random cycles in randomly selected storage elements, using the value that would normally be latched as the reference for the flip.

### 1.2.4 Logic Bugs

While logic bug activation conditions and effects are in general more difficult to pin down than the above electrical bug scenarios, they still exist. Logic bugs have activation conditions, which are the conditions under which the internal behavior of a design deviates from what the designer expects, and effects, which are the actual behavior of the bug as compared to a designer's expectations.

## 1.3 Error Propagation

When an error, fault, or bug is activated, it has by definition begun to change the internal behavior of a circuit. This change in behavior is not necessarily externally observable, however. Errors that are activated have multiple possible outcomes:

- The error effects are *masked* before they affect any output of the circuit. This means the error changes the internal behavior of the circuit temporarily, but that eventually the circuit reverts to behaving as if the error had never activated. Externally (i.e. observing the circuit outputs), there is no way to know a masked error has activated. An example of a masked error is a value that is computed incorrectly, but is then ignored because it is not selected by a multiplexer.

- The error effects change the output of the circuit. In this case we say that the error is *unmasked*.

- For effects that are not quickly masked or unmasked but instead make it to internal storage elements, there can be a third "limbo" state known as *silent data corruption*. In this state, the error has changed the internal behavior of the circuit, but whether the error will be masked or unmasked depends on the next access to the corrupted storage elements. For example, the corrupted elements may be overwritten, in which case the error becomes masked or the corrupted elements may be read and outputted, in which case the error becomes unmasked. Since data can be stored in memory indefinitely, there is no limit to how long silent data corruption can last.

While unmasked errors are clearly the most problematic, one should be careful about considering masked errors to be benign. In the same way that errors have activation conditions, errors are also sensitive to masking conditions that can turn a masked error into an unmasked one. A particularly insidious case is a masking condition that cause an error to be masked in testing mode, but unmasked in production mode. Thus for circuit validation, increasing observability to detect masked errors is also important.

## 1.4 Modulo Arithmetic

Modulo-$b$ arithmetic is arithmetic defined in a finite field with $b$ possible values, where each possible value corresponds to a remainder when an integer is divided by $b$ (using Euclidean division so that remainders are always positive). Addition, subtraction, and multiplication are defined with "wraparound" arithmetic where the result is immediately divided by $b$ and the remainder taken as the result.

For example, in modulo-3 space the possible values are $\{0, 1, 2\}$ and $2+2 = 1$ since in integer space $(2 + 2) \bmod 3 = 1$ where $a \bmod b$ is the remainder after dividing $a$ by $b$. Table 1.1 shows the mapping from integer space to modulo-3 space and Table 1.2 provides the modulo-3 addition, subtraction, and multiplication tables.

### 1.4.1 Properties

Since equivalent lightweight computations can be performed in modulo-$b$ space as in integer space, modulo-$b$ arithmetic can be used as a way to independently check integer computation. This works because we have defined a homomorphism from integer arithmetic to modulo-$b$ arithmetic. In other words, given integers $\{x, y, z\}$ and corresponding modulo-$b$ variables $\{x', y', z'\} = \{x, y, z\} \bmod b$ we observe the following properties:

$$x + y = z \implies x' + y' = z' \pmod{b} \tag{1.1}$$

$$x - y = z \implies x' - y' = z' \pmod{b} \tag{1.2}$$

$$xy = z \implies x'y' = z' \pmod{b} \tag{1.3}$$

where $(\text{mod } b)$ next to an equation indicates that the arithmetic is performed in modulo-$b$ space. Thus for Equations (1.1), (1.2), (1.3), $z'$ can be independently computed two ways: by mapping $z$ to modulo-$b$ space or by mapping $x'$ and $y'$ to modulo-$b$ space and performing the "shadow computation" in each equation.

Note that this "shadow computation" property holds for arbitrarily com-

Table 1.1: Integer to Modulo-3 Space Mapping

| Integer value | -6 | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Modulo-3 value | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 | 0 |

Table 1.2: Modulo-3 Addition, Subtraction and Multiplication Tables

| + | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 1 | 2 |
| 1 | 1 | 2 | 0 |
| 2 | 2 | 0 | 1 |

| − | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 2 | 1 |
| 1 | 1 | 0 | 2 |
| 2 | 2 | 1 | 0 |

| × | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 |
| 2 | 0 | 2 | 1 |

plex integer arithmetic involving addition, subtraction, and multiplication. For example, $x^2 - 4xy + 2y^2 = z \implies x'^2 - x'y' + 2y'^2 = z' \pmod{b}$. Exploiting the ability of homomorphisms such as this integer to modulo-$b$ mapping to scale to arbitrarily complex expressions is the key to implementing cost-effective error detection.

### 1.4.2 Aliasing

When using modulo-$b$ arithmetic as an error detection technique, aliasing occurs when the integer result of an erroneous computation corresponds to the same modulo-$b$ checksum as the correct result. For example, for modulo-3 arithmetic, if the correct integer result of a computation is 5, but the value -4 is produced instead, since both values map to 2 in modulo-3 space (Table 1.1) the error may not be detected since the correct "checksum" was produced.

One should be particularly wary of the aliasing that can occur when multiplying by a multiple of $b$. For example, for modulo-3 arithmetic, if any erroneous integer value is multiplied by 6, then the result will be 0 in modulo-3 space (Tables 1.1 and 1.2). Thus, in my application of modulo-3 arithmetic, I pay special attention to multiplication operations (see Section 3.1.2).

### 1.4.3 Modular Base

To use modulo-$b$ arithmetic to detect errors effectively in binary logic, I choose $b$ such that $z' = z \bmod b$ is a function of all of the bits in $z$. For example, $b = 4$ would fail this test because now $z'$ is just the last two bits of $z$, ignoring the higher order bits (and any errors in those bits). I also want

each bit in $z$ to have the ability to affect any bit in $z'$ to reduce to probability of aliasing. For example, $b = 6$ would fail this test because the last bit of $z'$ would only be affected by the last bit of $z$. The choice of $b$ will pass both of these tests if $b$ is odd and $b \geq 3$. In this thesis, I choose $b = 3$ to minimize the hardware cost, as only two bits are needed to represent the three possible modulo-3 values.

## 1.5 Execution Signatures

A software program contains variables that will have dynamic values during the program execution. Similarly, a hardware design has storage elements such as flip-flops that will have dynamic values during hardware execution. An execution signature is a hashed trace of the dynamic value of variables during software or hardware execution. Comparing the trace of hardware to be validated with a reference execution trace is a useful way to catch bugs. As one might imagine, tracing all variables at all times during software or hardware execution is expensive. We can use the following complementary techniques to reduce that cost:

1. Select a subset of all variables to trace. This reduces overhead, but also observability.

2. Create a diverse tracing schedule (i.e. different variables are traced in different execution states). This allows tracing resources such as buffers and I/O ports to be shared, reducing overhead.

3. Hash some of the traced variables. In order for the hash to be reproducible to detect errors, the values of the traced variable must be known (i.e. if there is an unknown or "x" value, then the hash cannot be reproduced and false bug detection positives will occur).

4. Compute a running hash to combine variables across cycles. Again all of the values that go into this running hash must be known.

In Chapter 4, I use all four of these techniques, and hash *all* of the traced variables to detect errors, using the high-level synthesis binding solution to identify when register values are known.

### 1.5.1 Catching Logic Bugs

If a design contains a nondeterministic logic bug and is run in a reference simulation and in hardware, the dynamic trace of the variable values will likely be different. The simulation would involve a different process (e.g. compilation by a high-level C compiler) than the hardware synthesis process, so the undefined behavior would likely manifest itself differently. For example, the values stored in uninitialized memory in hardware could be the device physics dependent power-on state, while uninitialized memory in a reference simulation might contain values from when it was used by another software process.

If a design only contains deterministic logic bugs and the simulation and synthesis tools correctly interpret the input code, the dynamic hardware and reference trace of the variable values will be identical. Thus hybrid tracing techniques will not catch deterministic logic bugs. The good news is that due to their deterministic nature, these bugs are easily reproducible in both hardware and reference executions. Furthermore, for hardware designs written in software input languages, we can leverage traditional software debugging techniques to debug hardware designs.

### 1.5.2 Hash Functions

In order to minimize hardware cost, I select the following xor-based hash functions:

$$H(x_1, x_2, ..., x_n) = x_1 \oplus x_2 \oplus ... \oplus x_n \tag{1.4}$$

$$S_n = \begin{cases} H_0 \oplus C & \text{if } n = 0 \\ H_n \oplus \text{rotate}(S_{n-1}, r) & \text{if } n > 0 \end{cases} \tag{1.5}$$

where $H$ is the reduction function that reduces a set of multi-bit variable values (technique 3 above) to a single hash. Similarly, $S_n$ is my running hash that combines the values of $H$ across execution cycles (technique 4 above) ($H$ in cycle $n$ is denoted $H_n$). The function rotate$(v, r)$ denotes bit rotation to the left of the bit vector $v$ by $r$ bits. $C$ and $r$ are constants. In Section 4.1.2, I refer to the hardware that implements these hash functions as an XOR tree and an LFSR, respectively.

Both of these functions have the desirable property that a change in any bit of the input variables will result in a change in at least one bit of the output. Equation (1.5) has the additional desirable property that $S_n$ depends on the number of cycles that have passed, $n$, even if all $H_n = 0$.

## 1.6 High-Level Synthesis

High-level synthesis, also known as behavioral synthesis, is a process that turns a software behavioral specification with an architectural description into hardware that implements that specification. The input to a high-level synthesis tool is typically a C language dialect with language extensions (e.g. pragmas and directives) and libraries to annotate the behavioral description with architectural specifications. The output is a hardware description, typically specified in Verilog or VHDL. A typical synthesis engine will perform the following steps:

1. **Compilation**: The synthesis engine parses the input code and converts it to an intermediate representation (IR).

2. **Transformation and Optimization**: The synthesis engine runs the IR through a series of optimization passes, similar to software compiler optimizations. The engine also does architectural transformations such as loop unrolling and pipelining.

3. **Allocation**: For each hardware resource—memories, ports, registers, and functional units—the synthesis engine determines what kind and how many of each to use. Larger allocations usually increase performance at the cost of area.

4. **Scheduling**: The engine creates a state machine corresponding to the control flow of the software specification. For each state, the engine determines what operations—computations, memory access, and/or I/Os—will occur in that state. The engine may insert extra states to provide sufficient cycles to complete complex chains of operations.

5. **Binding**: For each operation, the engine determines which hardware resource(s) will be involved in performing the operation. Operations

that can never occur at the same time can share a common hardware resource. The engine inserts multiplexers at this stage to facilitate such sharing.

6. **RTL Generation**: The engine generates a complete RTL description of the final state machine and datapath solution.

# CHAPTER 2

# RELATED WORK

## 2.1 Hardware Reliability

A traditional approach to error detection in hardware designs is by duplicating each component, also called dual modular redundancy (DMR) [4]. But this approach comes with a 2x area cost that eliminates the area and power reduction benefits of Moore's law scaling. DIVA [5] is another popular technique which uses an extra checker core to verify the correctness of the main core computation and commit only non-faulty results. Concurrent error detection (CED) [6] uses HLS to introduce redundancy at the functional unit level. Although each component is fully duplicated, this technique aims at reducing area and performance overhead through resource sharing. But this technique can incur at least 75% area cost for simple and small datapaths.

Another approach is time-redundancy, where we re-compute results using the same hardware units to detect errors. [7] uses a time redundancy-based concurrent error detection scheme with diverse binding solutions in its re-computation stage but has performance overheads even though it incurs low area cost. Argus [8] is a prototype processor with a modulo-3 arithmetic checker that can detect up to 98.0% and 98.8% of unmasked transient and permanent errors respectively. Argus has low area (17%) and performance (4%) costs but it is limited to the Von Neumann processor architecture and, to the best of our knowledge, there is no similar work in high-level synthesis that targets application-specific custom logic and accelerator designs.

The traditional approach to reliable hardware is triple modular redundancy (TMR) [9] where two additional units are added to the main unit and a majority voting unit. The three units perform the same computation and if any of the three units fail, the other two units can correct and mask the fault. Although TMR has a high fault coverage, it has a 3x area cost. [10] integrated

modular redundancy into high-level synthesis and presented techniques to increase reliability with cost and performance constraints and decrease cost given reliability constraints, but not both together. New approaches to modular redundancy such as statistical error compensation (SEC) involving pairing an estimator module with unreliable hardware still come with high (50-100%) area cost [11]. Razor [12] is a gate level transformation that adds a shadow latch for each flip-flop to detect timing errors. Although it has a low area and performance overhead (<3%), it is limited to only detecting timing errors. [13] proposes a technique to recover from soft errors but does not perform any error injection experiments and has a passive approach to masking errors whereas we actively detect and correct errors.

Compared to our reliability solution in Chapter 3, the techniques mentioned above have one or more of the following limitations:

1. Not automated;
2. Does not protect state machine and control logic;
3. Limited to a single fault model;
4. Has significant performance cost;
5. Has area costs approaching 2x or more.

## 2.2 Post-Silicon Validation

The inspiration for my H-QED PSV solution (Chapter 4) is QED [14, 15, 16, 17], which is a software technique for the post-silicon validation of programmable microprocessors. In general, PSV techniques that target processors (e.g., [18, 19] and others) are inadequate for bugs inside accelerators.

Although H-QED may appear to be similar to tracing techniques used in PSV (e.g., using trace buffers or system memory [20, 21, 22, 23]), there are important differences:

1. H-QED systematically collects signatures, unlike tracing techniques that are often ad-hoc or based on heuristics;
2. H-QED does not require extensive low-level (e.g., RTL) simulation;

3. H-QED does not 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 [5, 24, 25, 26, 27, 28]). 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 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 [29, 30, 31], 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 [32, 33, 34]. 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.

# CHAPTER 3

# ERROR DETECTION THROUGH MODULO-3 SHADOW DATAPATHS

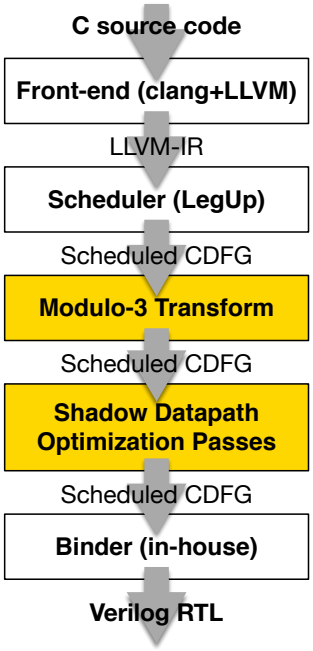
In this chapter, I propose creating a redundant, but smaller "shadow" datapath based on modulo arithmetic to detect reliability problems in an HLS design's main datapath. I automate the creation of this "shadow" datapath through a series of modulo-3 shadow datapath HLS transformations. Our main innovations are:

1. Intelligent scheduling of intermediate register consistency checks for maximum coverage with minimum checker allocation;
2. Support for mixed arithmetic/non-arithmetic data paths;
3. A register-duplication based checkpointing technique to demonstrate the error correction potential of our approach;
4. An FPGA accelerated, fully automated error injection framework using a gate-netlist transformation to enable accelerated injection for three fault models;
5. Error detection latencies three orders of magnitude faster than an end result check;
6. Unmasked error detection coverage of 99.42% for an assortment of three different kinds of fault models.

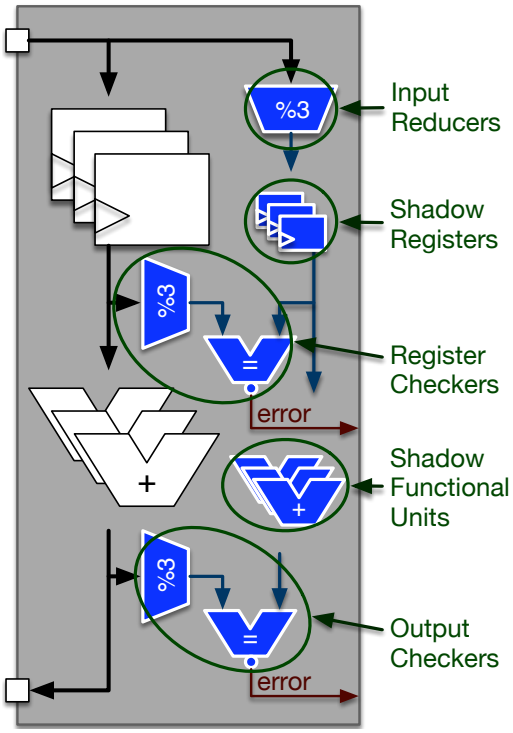
The rest of this chapter is organized as follows: Section 3.1 explains the method we use to perform our error detection and correction transformations and Section 3.2 discusses our experimental setup and results.

## 3.1 Method

Our approach to protecting a hardware design is a series of modulo-3 shadow datapath HLS transformations. An overview of how these transformations fit into the HLS process is illustrated in Figure 3.1a. We use the LegUp HLS scheduling engine [35] to schedule the original datapath, and perform binding with our in-house binding engine. Our transformations involve some additional scheduling steps (see Section 3.1.2). We perform our error detection transformations after scheduling but before binding to insure that the latency of the hardware function does not increase.




(a) HLS Overview




(b) Modulo-3 Transform

Figure 3.1: Overview of our method. (a) Integration of our reliability transformations into the high-level synthesis process. (b) Illustration of our core mod-3 transform. The original datapath is colored black/white and the shadow datapath is in blue.

Figure 3.1b provides an overview of our basic modulo-3 shadow datapath transformation. For each input port, we add a mod-3 reducer to compute the input value mod-3 residue, effectively creating a shadow mod-3 input. For each arithmetic functional unit (e.g. add, subtract, multiply), we add

a corresponding shadow mod-3 functional unit. For each datapath flip-flop, we add a corresponding 2-bit flip-flop to store and propagate the mod-3 checksum in a parallel datapath. For each output port, we add a mod-3 checker which consists of a reducer and 2-bit equality comparator, which then drives shared error ports. The result is that each main computation is independently performed in mod-3 space, and the two results are checked for consistency. In the following two subsections, we discuss the design of these mod-3 functional units and the transformation that inserts them into high-level synthesized designs.

### 3.1.1 Modulo-3 Functional Units

#### Basic Functional Units

Mod-3 functional units represent the types of functional units which operate in the mod-3 space. Since only two bits are required to encode three possible values in mod-3 space, a simple approach is to use two representations for 0: 00 and 11, which is the approach taken for previous designs of mod-3 functional units. Our key innovation is to ignore the 11 encoding (we name it the **U** value) and optimize it as a don't care.

Table 3.1: Modulo-3 Adder Functional Specification Table

| **value** | **encoding** |
|---|---|
| **0** | 00 |
| **1** | 01 |
| **2** | 10 |
| **U** | 11 |

| $+_3$ | **0** | **1** | **2** | **U** |
|---|---|---|---|---|
| **0** | 0 | 1 | 2 | X |
| **1** | 1 | 2 | 0 | X |
| **2** | 2 | 0 | 1 | X |
| **U** | X | X | X | X |

Table 3.2: Optimization Results for Shadow Mod-3 Units

| **Function** | **32-bit unit** | | **naive shadow** | | **optimized shadow** | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| **Add** | 163 | 1.30 | 17.6 | 0.15 | 9.30 | 0.08 |
| **Multiply** | 2381 | 2.05 | 10.9 | 0.08 | 5.75 | 0.05 |

Thus if either input is the **U** value, then the output does not matter as the **U** case will never occur in normal operation. As illustrated in Table 3.1

for the mod-3 adder, there are 9 fixed output cases and 7 *don't care* output cases for each two-input mod-3 unit. Through the use of Karnaugh maps, we optimally exploited these don't cares to find a low area cost design expressed as a sum of products. We verified the optimality of our sum of products solution through exhaustive search of all $4^7$ possible *don't care* assignments (i.e. to check for better solutions involving compound gates). Table 3.2 shows the effects of our optimization. For logic synthesis, we implemented our designs in Verilog, used Synopsys Design Compiler 2013-12.sp4 with an ARM 45nm standard cell library, and optimized for minimum area. We measure area in square micrometers and delay in nanoseconds.

Constant Functional Units

We also consider an additional class of constant operation units generated by high-level synthesis, units that have a constant as one input. We can think of this constant as "baked-in" to the logic of the unit so that structurally the unit has a single input and a single output. For example, a "+10" constant operation unit takes some value $x$ as input and outputs $x + 10$.

Table 3.3: Shadow Unit Metrics for Operation with Constant $c$

| **Function** | **c = 0** | | **c = 1** | | **c = 2** | |
|---|---|---|---|---|---|---|
| | area | delay | area | delay | area | delay |
| **Add c** | 0 | 0 | 0.96 | 0.02 | 0.96 | 0.02 |
| **Multiply by c** | 0 | 0 | 0 | 0 | 0 | 0 |

Table 3.3 shows the cost of the constant operation versions of our mod-3 units. Since we can reduce each constant to its mod-3 residue at compile time, there are only three versions of each constant unit. We observe that the operations +0 and x1 have no area cost since they lower to the identity function and x0 lowers to the constant zero for multiplication. As discussed in Section 3.1.2, such operations are optimized out by our high-level synthesis optimization passes.

With such functional unit optimizations, our method has an even greater area-cost advantage over double or triple modular redundancy for arithmetic datapaths.

Modulo-3 Reducers

Mod-3 reducers are our modulo-3 residue computing units. They are implemented as a tree of $\lceil \log n/2 \rceil$ stages of modulo-3 adders where $n$ is the input width, similar to the tree approach in [36]. An example reducer for $n = 16$ is illustrated in Figure 3.2. The design works by grouping the input bits into pairs and effectively constructing a base $2^2 = 4$ representation of the input value. Since $4^n \bmod 3 = 1$ for all $n \geq 0$, each base 4 digit has the same weight in mod-3 space and thus we can compute the mod-3 sum of all of the digits in a straightforward tree reduction.

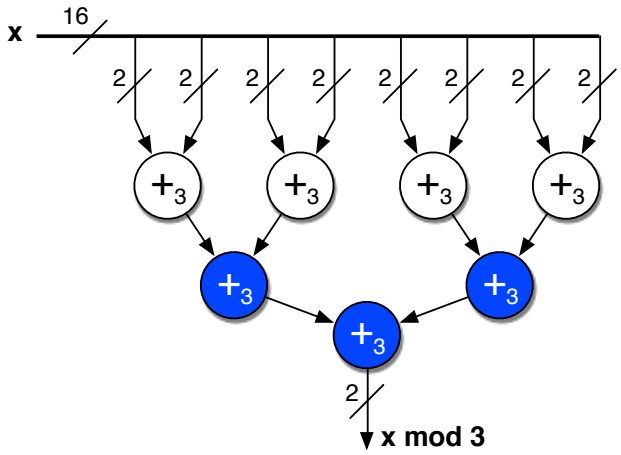



Figure 3.2: Optimized mod-3 reducer topology for a 16-bit unsigned reducer. Optimized mod-3 adders are colored blue.

Table 3.4: Optimization Results for 32-bit Mod-3 Reducer

| **Reducer Type** | **[36]** | | **ours** | |
|---|---|---|---|---|
| | area | delay | area | delay |
| **Unsigned** | 263 | 0.62 | 203 | 0.46 |
| **Signed** | 267 | 0.66 | 207 | 0.51 |

Since the first stage adders must take all possible values (0, 1, 2, and 3) as inputs, we cannot perform don't care optimizations for those units. But since we design the first stage adders to normalize their output to be 0, 1, or 2, all subsequent stages can optimize the fourth ("3" or U) value as a don't care. To the best of our knowledge, this optimization was not previously explored. With this optimization, we observe a 22-23% area cost reduction and a 23-26% delay reduction compared to [36].

Thus far, we have assumed that the original datapath uses an unsigned bit encoding for all variables. To modify our reducers to handle a signed (2s complement) variable, we leverage that the only difference between the unsigned and signed (2s complement) encodings is the weight of the most significant bit (MSB). In the unsigned encoding, the MSB has a weight of $2^{n-1}$ while in the signed encoding, it has a weight of $-2^{n-1}$ where $n$ is the number of bits. Without loss of generality, if we assume $n$ is even, then $2^{n-1} \bmod 3 = 2$ and $-2^{n-1} \bmod 3 = 1$. Since the second most significant bit always has a weight of 1, the insertion of a half-adder is sufficient to normalize the two most significant bits for a signed reducer. Table 3.4 shows the small cost of this extra half-adder.

### 3.1.2 High Level Synthesis Transformations

Our HLS transformations, as illustrated in Figure 3.1 on page 20, consist of a core mod-3 transform that generates the shadow datapath as well as some dataflow-level optimization passes on the generated mod-3 logic. Our transformations operate on a scheduled control/data flow graph.

By leveraging the state machine and data flow graph information available in this HLS stage, we can perform transformations and optimizations not possible at the RTL or gate-level stage. In the following subsections, we discuss how we handle mixed arithmetic-nonarithmetic datapaths, the scheduling of intermediate register consistency checks for maximum coverage with optimized sharing, pipelining for deferred shadow datapath scheduling to eliminate clock period overhead and lower area cost, and binding diversity between the main and shadow datapaths for improved fault coverage.

#### Handling Non-arithmetic Components

HLS generated designs involve non-arithmetic components including state machine logic, bitwise operations, and comparators that have single bit outputs. Each non-arithmetic component is duplicated such that each component has a redundant counterpart. However, such units have low area overhead. For example, bitwise operations have very low area cost and shifts by a constant have zero area cost. We also observe low overheads for duplication of non-arithmetic units (Area and Delay overheads are mentioned in

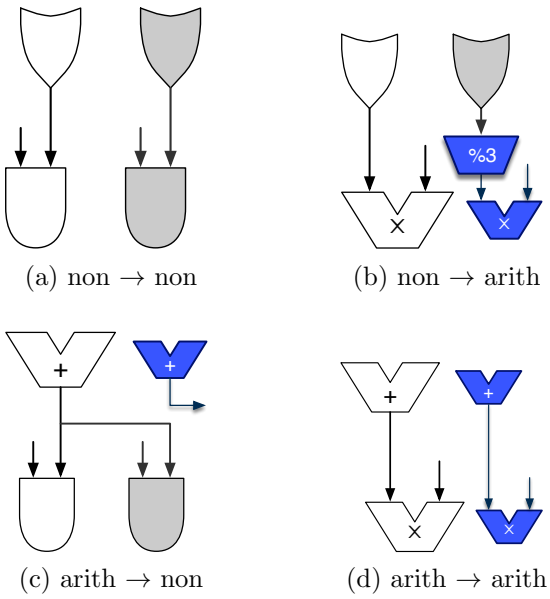



Figure 3.3: Shadow/duplicate connection cases. For each subfigure, the original graph is on the left and the redundant logic is on the right. For the redundant logic, nonarithmetic components ("non") are duplicated with the duplicates in grey. Arithmetic components ("arith") are mod-3 shadowed with the shadows in blue. The unit labeled "%3" is a mod-3 reducer.

Table 3.5 on page 32).

There are a number of cases to deal with when we generate shadow connections for arithmetic and non-arithmetic components, which are illustrated in Figure 3.3. Connections between two duplicate components and between two mod-3 components are straightforward: just make connections corresponding to those in the original datapath (Figures 3.3a and 3.3d). We can connect a duplicate component output (full bit width) to a mod-3 component input (2 bit) through a mod-3 reducer (Figure 3.3b). Connecting a mod-3 component output to a duplicate component input is not possible since information lost in the mod-3 reduction cannot be recovered. Thus the duplicate component input is connected to the same output as the original component (Figure 3.3c).

Making connections this way can leave some mod-3 components with outputs unconnected, which we call *mod-3 sinks*. For example, the mod-3 adder in Figure 3.3c may not have a mod-3 component to connect to in its fanout. Such mod-3 sinks may output an inconsistent mod-3 checksum due to an error that occurred in the main datapath, but there would be no way to detect it. Thus we add a mod-3 checker for each mod-3 sink to insure such errors

are detected.

We deal with constant multiplication by multiples of three in a similar way since the mod-3 result is always zero (Section 1.4.2). Our optimization passes will replace such a shadow multiplier with a constant, leaving no pin to connect its original input to. Thus we treat constant multiplication by a multiple of three as an additional shadow datapath barrier: if it results in a mod-3 sink then we add a mod-3 checker.

### Register Consistency Check Scheduling

Some errors may be masked in the main datapath (and thus masked in the shadow datapath) before they reach the primary output. Other errors may be unmasked, but undetected due to aliasing (see Section 1.4.2) that occurs in the shadow datapath. To maximize our chances of detecting such errors, we insert checkers on the output of datapath registers, using strategic scheduling of check operations to share as many mod-3 reducers as possible.

Compared to the rest of the shadow datapath, reducers are expensive (Compare Tables 3.2 and 3.4). Reducers are scheduled in fixed states for use at output ports and mod-3 sinks to produce residues for checkers as well as at input ports to provide shadow inputs (Figure 3.1b). Intermediate register checkpoints, on the other hand, have flexible scheduling constraints corresponding to their liveness state machine subgraph.

To exploit this flexibility and minimize reducer allocation, we select register liveness intervals that are more than one cycle long and that extend across a basic block boundary (control flow divergence or convergence). For each liveness interval, we attempt to schedule a checkpoint at each *use* (read) of the corresponding SSA variable[1] with the constraint that we cannot schedule more reducers at a state than have been allocated. The intuition behind this method is that we want to catch errors right before they leave a register to go through functional units where they may be masked or aliased. If the checkpoint cannot be scheduled at a state, we attempt to recursively schedule it at each of the state's predecessors.

The core recursive algorithm is listed in Algorithm 1. In the event of a scheduling failure, we allocate an additional reducer and try again until check

[1]Single-static assignment variable which is written only once and thus corresponds to one liveness interval for a variable.

**Algorithm 1** Core recursive scheduling algorithm

```
function SCHEDULE(var, state)
    if (var, state) has not been visited or scheduled then
        if reducer_count[state] = max_reducers then
            preds ← state predecessors that var is live in
            if preds = ∅ then
                increment max_reducers
                restart scheduling process
            end if
            for each pred in preds do
                schedule(var, pred)
            end for
        else
            schedule check for (var, state)
            increment reducer_count[state]
        end if
    end if
end function
```

scheduling succeeds.

Pipelining for Deferred Shadow Datapath Scheduling

While our mod-3 shadow functional units have low latency (Tables 3.2 and 3.3), our mod-3 reducers have high latency (Table 3.4). In addition, the insertion of a mod-3 checker on a mod-3 sink's corresponding main component can cause severe timing violations if the main component is part of an operation chain. Even if the timing violations are corrected through gate sizing, the area cost can be quite large as 1x transistors are replaced with 4x and 8x transistors to meet timing requirements. Ideally, we want all of the mod-3 components to be mapped to 1x gates for minimum area overhead.

Thus our solution is to insert pipeline flip-flops both in front of and behind each mod-3 reducer. The shadow datapath schedule is then deferred by two cycles, adding two cycles of error detection latency in exchange for reduced area cost.

Shadow Datapath Optimization Passes

Our mod-3 transformation can create no-op identity operations and redundant components. This superfluousness motivated us to add a shadow datapath optimization pass to eliminate them as shown in Figure 3.1a which consists of two components:

1. **Constant propagation and identity elimination**: A +6 adder results in the generation of a +0 mod-3 component, which is an identity. A x6 multiplier evaluates to a constant 0 in mod-3 space, which could then propagate to other operations and make their result evaluable at compile time.

2. **Redundant component elimination**: A x8 and a x11 multiplier both result in the generation of a x2 mod-3 component. If both multipliers are connected to the same input, the second x2 mod-3 component is redundant and can be removed.

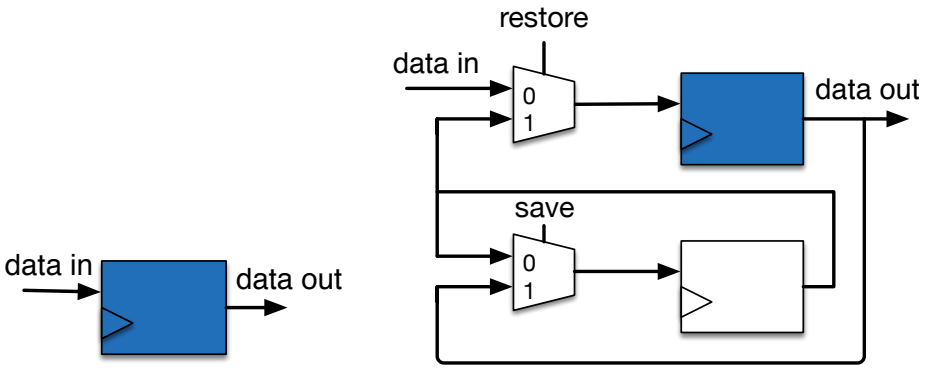
Diverse Binding

We perform binding of our optimized and scheduled control and data flow graph with our in-house binding engine, which creates diverse (different) binding solutions between the original and duplicate / mod-3 datapaths. Such diverse binding makes it difficult for control errors and stuck-at faults to affect both redundant datapaths in the same way. Further state machine checking is enabled by comparing the state registers of the redundant state machines and using one state machine to control the main datapath and another one to control the duplicate and shadow datapaths. Both the shadow datapath and the duplicate state machine run two cycles behind the main computation, so synchronization is not an issue. The binding engine's primary goal is to maximize sharing where profitable for area cost, minimizing the number of reducers allocated.

### 3.1.3 Recovery

To enable error recovery for soft errors, we use a checkpoint and recovery register transformation, illustrated in Figure 3.4. For each state and datapath

register, we add a duplicate register to store checkpoint data. At regular intervals (configurable), we assert the "save" signal to take a snapshot of the state of each datapath and state register in a corresponding duplicate. Error detection triggers a "restore" signal which recovers the state from the previously recorded checkpoint, i.e. the cycle where the "save" signal was asserted.




(a) Original flop (b) Transformed flop

Figure 3.4: Flip-flop transformation for soft error recovery.

Our error recovery technique will work for soft errors as long as the error has not made it into the checkpoint snapshot. A checkpoint is corrupted when an error is activated before, but detected after the checkpoint. We consider an error to be masked if it does not affect the primary outputs of the generated core or the timing of those outputs. Otherwise, it is an unmasked error. The probability of checkpoint corruption, $P_{CC}$, is defined as in Equation (3.1), where $l$ is the unmasked error detection latency, $P_l$ is the probability of that particular latency (i.e. $\sum_l P_l = 1$) and $CI$ is the checkpoint interval (configurable). An error is removed if either it is masked to begin with or it is unmasked, detected, and successfully recovered by rolling back to an uncorrupted checkpoint; we formally define the *error removal rate* as the number of removed errors divided by number of total errors, as formalized in Equation (3.2). In this equation, $E$ is the error removal rate; $M$ is the *error masking rate* (defined as number of masked errors divided by number of total errors); and $U$ is the *unmasked error detection rate* (defined as number of unmasked errors detected divided by number of total errors). An error is detected ($ED$) in a given cycle if an error occurred in that cycle and it was detected by our detection logic, as formalized in Equation (3.3), where $P_{error}$

stands for the probability of error activation in each cycle and *det* stands for total error detection rate given error activation. *Avg.rollback* is the number of cycles, on average, that we would rollback on detection of an error. Since the rollback length distribution is uniform, the average is approximately half the checkpoint interval (Equation (3.4)). Thus, the average rollback cycle overhead is the product of the average rollback length and the probability of an error being detected in a given cycle (Equation (3.5)).

$$P_{\mathrm{CC}} = \sum_{l} P_l \frac{\min(l, \mathrm{CI})}{\mathrm{CI}} \leq \frac{l_{\mathrm{avg}}}{\mathrm{CI}} \tag{3.1}$$

$$\mathrm{E} = \mathrm{M} + \mathrm{U}(1 - P_{\mathrm{CC}}) \tag{3.2}$$

$$\mathrm{ED} = P_{error} \times \mathrm{det.} \tag{3.3}$$

$$\text{Avg. Rollback} = \sum_{r=1}^{\mathrm{CI}} \frac{r}{\mathrm{CI}} = \frac{\mathrm{CI}+1}{2} \tag{3.4}$$

$$\text{Cycle Overhead} = \mathrm{ED} \times \text{Avg. Rollback} \tag{3.5}$$

## 3.2 Experimental Results

### 3.2.1 Setup

Our experimental setup is illustrated in Figure 3.5. We performed logic synthesis with Synopsys Design Compiler 2013-12.sp1 with an ARM 45nm standard cell library, and optimized for maximum clock frequency. We evaluated the detection coverage of our approach with error injection enabling netlist transformations which support stuck-at, transient, and timing errors.

To inject stuck-at faults, the netlist transform inserts AND (for stuck-at 0) or OR (for stuck-at 1) gates at randomly selected gate outputs. To inject transient errors, we insert XOR gates at the "D" inputs of randomly selected flip-flops. For timing errors, we induce setup time violations by performing timing simulations with a fast clock to collect flop-cycle pairs where timing errors are activated while continuing error-free execution with the use of a razor flip-flop like transformation, similar to the activation detection method of [37]. Then we pass these flop-cycle pairs as a subset of transient errors to our error injection enabling netlist transformation.

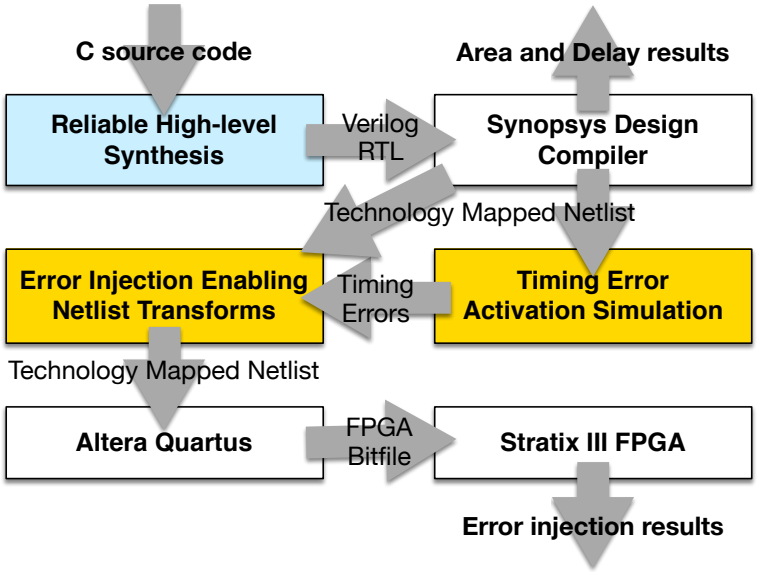



Figure 3.5: Our error detection coverage evaluation framework. Our "reliability-centric" high-level synthesis process is elaborated in Figure 3.1a. Our customized steps are highlighted in yellow.

To accelerate fault effect evaluation, we map the ASIC netlist to an Altera Stratix III FPGA for emulation. A hardware test driver module mapped to the FPGA communicates with the host system to facilitate thousands of rapid (<1 second each) back-to-back full runs of the design under test, injecting one error from the sample list at a time. As one would expect, stuck-at faults are activated for the duration of the design execution, while transient errors are activated for one cycle.

### 3.2.2 Results

We used benchmarks from the PolyBench/C 3.2 benchmark suite [38] and modified the benchmarks to use fixed-point encodings for originally floating-point encoded values as our transformations currently do not support floating-point operations. We implemented fixed point arithmetic with C integer arithmetic operations with shifts for binary point alignment. "Matrix 4x4" is a tiled version of the matrix multiply benchmark that completely unrolls $4 \times 4$ tiles to explore performance/area tradeoff. We synthesized our benchmarks using our method (Section 3.1.2) and used our experimental setup (Section 3.2.1).

To determine the area cost of our error detection approach, we compare

Table 3.5: Area and Clock Period Overhead Results

| Benchmark | Baseline | | Detection | | Total | |
|---|---|---|---|---|---|---|
| | area ($\mu m^2$) | period (ns) | area ov.(%) | period ov.(%) | area ov.(%) | period ov.(%) |
| **Atax** | 13 434 | 0.89 | 28.3 | −2.4 | 52.7 | 2.0 |
| **Bicg** | 13 923 | 0.90 | 27.4 | −5.2 | 57.6 | −0.9 |
| **Floyd-Warsh** | 12 764 | 0.70 | 26.9 | 0.3 | 57.4 | 5.8 |
| **Gemm** | 13 380 | 0.84 | 30.3 | 1.7 | 56.4 | 6.3 |
| **Gemver** | 18 855 | 1.00 | 26.8 | 1.5 | 55.4 | 5.4 |
| **Gesummv** | 13 230 | 0.84 | 30.0 | 1.9 | 57.1 | 6.6 |
| **Matrix 4 × 4** | 65 258 | 1.03 | 5.7 | 8.8 | 29.5 | 12.6 |
| **Matrix** | 11 151 | 0.80 | 22.1 | 1.0 | 55.6 | 5.9 |
| **Mvt** | 16 212 | 0.88 | 40.2 | −1.1 | 67.9 | 3.3 |
| **Symm** | 16 943 | 0.84 | 24.9 | 2.9 | 57.2 | 7.5 |
| **Syr2k** | 15 183 | 0.85 | 23.0 | 1.2 | 48.9 | 5.8 |
| **Syrk** | 13 975 | 0.89 | 23.1 | 0.1 | 48.9 | 4.5 |
| **Median** | 13 949 | 0.86 | 26.8 | 1.1 | 56.0 | 5.8 |
| **Mean** | 18 763 | 0.87 | 25.7 | 0.9 | 53.7 | 5.4 |

the core area of an unprotected baseline benchmark synthesized without our mod-3 shadow datapath transformations against our experimental version synthesized with the mod-3 transforms. Table 3.5 shows the area and clock period overhead for both the detection logic and estimated overhead (through characterization of the hardware in Figure 3.4) for the total logic which includes both detection and recovery. We observe on average an area cost of 25.7% for detection and estimate 53.7% for both detection and recovery. Interestingly, we observe a 5.7% detection area cost for the highly parallelized "Matrix 4x4" benchmark, suggesting that lower overheads are achievable in large high-throughput accelerator designs.

To observe fault coverage, we injected a sampling of 2,000 stuck-at, 10,000 transient and 10,000 timing errors into each synthesized core. The outcome of our fault injection experiments is shown in Table 3.6.

For unmasked errors, we observe an average stuck-at fault coverage of 99.1%, soft error coverage of 99.5%, and timing error coverage of 99.6%. To provide some context, Argus, which we consider to be a state-of-the-art error detecting microprocessor, can detect 98.0% of transient errors and 98.8% of stuck-at faults [8].

It is difficult to make a direct comparison with previous HLS work since

Table 3.6: Fault Coverage

| Benchmark | Unmasked (%) | | | Masked (%) | | |
|---|---|---|---|---|---|---|
| | stuck | trans. | timing | stuck | trans. | timing |
| **Atax** | 99.7 | 99.8 | 99.8 | 68.6 | 28.3 | 65.0 |
| **Bicg** | 98.9 | 97.1 | 100 | 73.9 | 31.1 | 57.4 |
| **Floyd-Warsh** | 99.9 | 100 | 100 | 64.8 | 40.9 | 73.4 |
| **Gemm** | 98.5 | 100 | 100 | 100 | 31.8 | 77.2 |
| **Gemver** | 99.5 | 99.9 | 100 | 78.0 | 18.8 | 77.5 |
| **Gesummv** | 99.9 | 99.3 | 100 | 67.6 | 38.4 | 56.1 |
| **Matrix 4X4** | 98.8 | 98.7 | 99.5 | 67.7 | 48.9 | 76.5 |
| **Matrix** | 100 | 100 | 100 | 76.1 | 25.9 | 54.1 |
| **Mvt** | 96.7 | 100 | 100 | 73.4 | 17.0 | 66.9 |
| **Symm** | 99.6 | 99.0 | 97.7 | 76.8 | 36.4 | 47.7 |
| **Syr2k** | 99.5 | 99.7 | 98.9 | 73.5 | 33.5 | 81.7 |
| **Syrk** | 98.5 | 100 | 100 | 71.4 | 31.9 | 73.2 |
| **Median** | 99.5 | 99.9 | 100 | 72.8 | 31.8 | 70.0 |
| **Mean** | 99.1 | 99.5 | 99.6 | 72.0 | 31.9 | 67.2 |

high-level synthesis benchmarks with experimental error injection and area cost are quite limited. For reference, Concurrent Error Detection [6] uses HLS to fully duplicate each component but attempts to compensate for area cost through resource sharing and has around 75% area cost for a simple, fully arithmetic datapath which in theory is not susceptible to aliasing.

Figure 3.6 shows the estimated soft error removal rate and rollback cycle overhead for our error recovery method with checkpoint intervals ranging from 10 to 100k cycles calculated through Equations (3.1)-(3.5).

The baseline average masking rate of the unmodified designs is 70.2% (indicated by the lower dotted line), and we achieve an total error removal rate (indicated by the "Error Removal Rate" curve) arbitrarily close to the theoretical upper bound (all errors detected are corrected) which is 99.83% (indicated by the upper dotted line).

We cannot achieve an error removal rate of 100% as we have a small percentage of undetected, unmasked errors. The 4 parallel lines represent rollback cycle overheads for different soft error rates. For reference, [39] reports a worst case error rate of around $10^{-16}$ errors / cycle for a space environment assuming a clock frequency of 1GHz.

What is interesting to observe is the tradeoff between the error removal rate and rollback cycle overhead. Larger checkpoint intervals reduce the

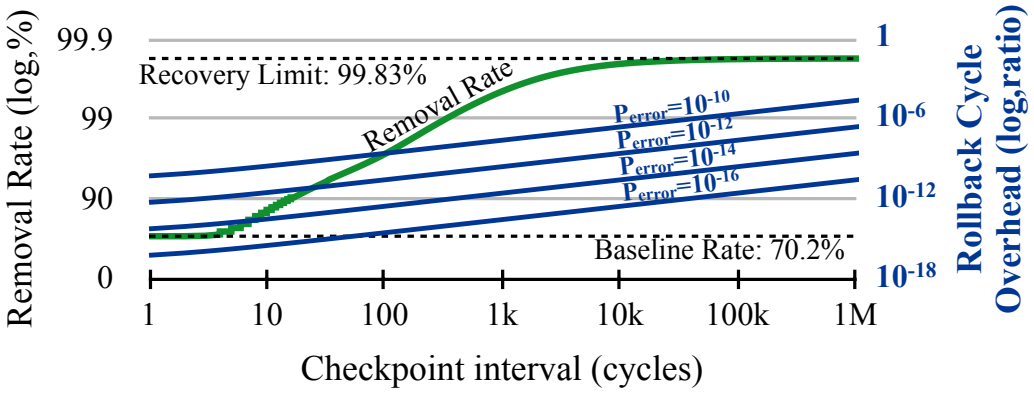



Figure 3.6: Error removal rate and rollback cycle overhead

chance of checkpoint corruption, resulting in higher error removal rates. At the same time large checkpoint intervals result in larger jumps back in time for each error detection triggered rollback, resulting in larger cycle overheads. To pick a number, 1000 cycles is a reasonable tradeoff as we are at the point of diminishing returns for the error removal rate (98.6%).

Figure 3.7 shows the soft error detection latency distribution for unmasked errors, masked errors and both. "End Result Check" (ERC) is a basic error detection method involving comparing the benchmark's output with its expected output once execution is complete. We observe mean latencies of 8.72, 17.14, 12.75, and 36.2k cycles for unmasked, masked, both and ERC respectively, for an error detection latency improvement of 4150x over the ERC.

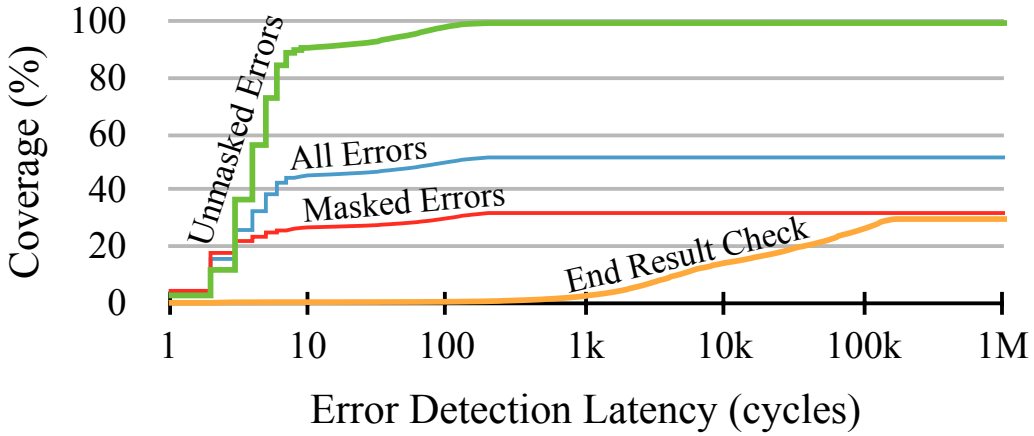



Figure 3.7: Soft error detection latency distribution

# CHAPTER 4

# VALIDATION THROUGH HYBRID SIGNATURE GENERATION

In this chapter, I 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 [14, 15, 16, 17]. 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) [40, 41] 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.

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:

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). 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 two 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 four previously unknown logic bugs in the widely used CHStone HLS benchmark suite [42];

4. H-QED incurs less than 10% overhead for the accelerator it validates, and negligible performance costs;

5. H-QED does not require any failure reproduction[2] or low-level simulation (e.g., RTL or netlist) to detect bugs;

[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 [43].

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 this chapter is organized as follows: Section 4.1.1 presents our H-QED technique and Section 4.2 presents our experimental results.

## 4.1 Method

### 4.1.1 H-QED Overview

Figure 4.1 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 (Figure 4.1(a)). Figure 4.2 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 Figure 4.2) 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 (Figure 4.1(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 Figure 4.2).

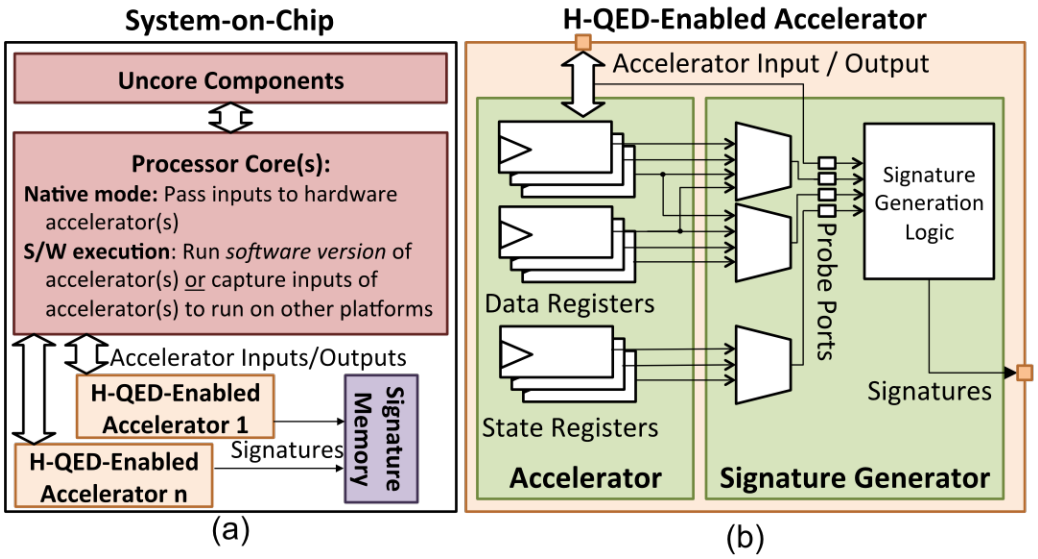



Figure 4.1: H-QED-enabled accelerators inside an SoC. (a) SoC-level view, and (b) block diagram of an H-QED-enabled accelerator showing the accelerator and the signature generator.

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,

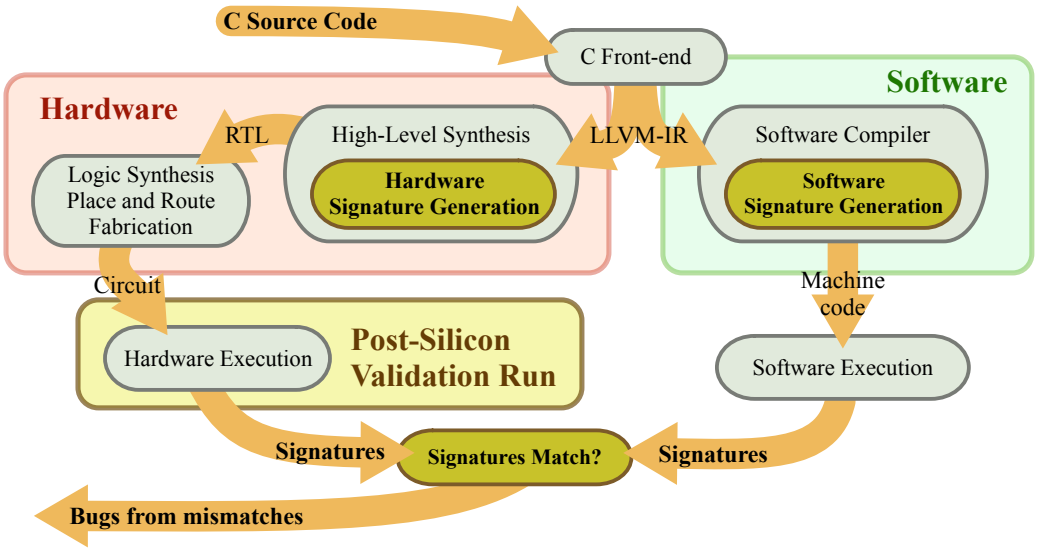



Figure 4.2: H-QED flow

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 do not require bugs to be activated (or reproduced) during the second run.

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 do not require bugs to be activated (or reproduced) after the first PSV run.

We built our framework on top of LLVM [44, 45] using a common LLVM

Internal Representation (LLVM-IR) to drive the generation of the H-QED-enabled hardware accelerator and the corresponding software version.

### 4.1.2 Hardware Signature Generation

Consider the input pseudo-code shown in Listing 4.1. 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 Figure 4.3. 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.

Listing 4.1: Input Pseudo-code Example

```
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)
```

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 Figure 4.3), 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 (Figure 4.3).

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:

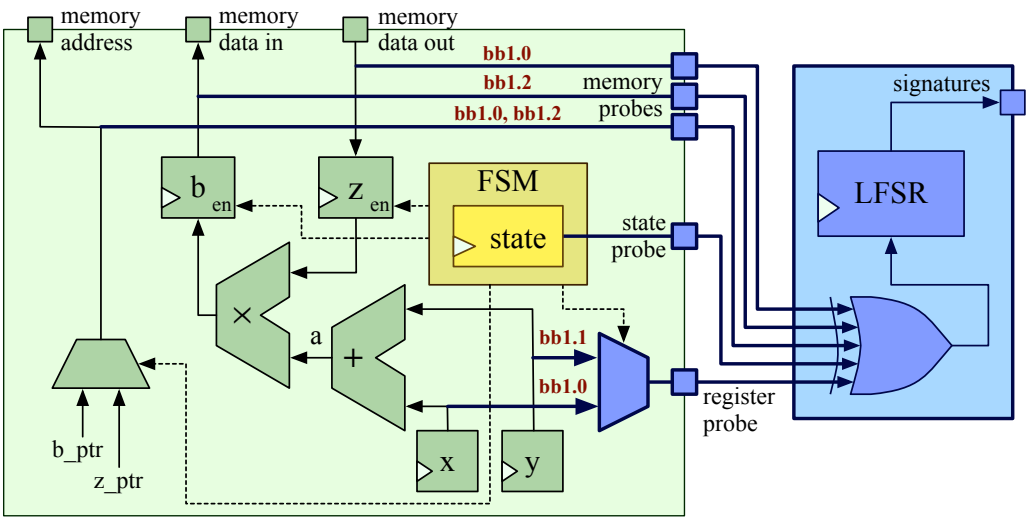



Figure 4.3: H-QED-enabled accelerator with hardware signature generation

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 block). 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).

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" port is probed in state bb1.0 and the memory "data in" port is probed in state bb1.2 as annotated in Figure 4.3. The memory address port is probed in both bb1.0 and bb1.2.

Every annotated probe port in Figure 4.3 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 Figure 4.3. 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).

To avoid clock period overhead, we register each probe port output. This effectively pipelines the signature generation logic, adding a cycle delay in exchange for avoiding gate upsizing (and thus minimizing the area cost) of our signature generation logic. 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 Section 4.1.3).

### 4.1.3 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 4.2 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.

Listing 4.2: Probe schedule and hardware memory addresses

```
// 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
```

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 Figure 4.3).

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 4.2). The additional code for this address conversion on the software side does no 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 4.3 shows the resulting H-QED-enabled software version for our example.

Listing 4.3: H-QED enabled software version

```
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)
```

### 4.1.4 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.

## 4.2 Experimental 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, error detection latencies, and coverage estimates for logic and electrical bugs. We used all 12 benchmarks from CHStone [42] and 15 benchmarks from the PolyBench [38] 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.

We used a 16-bit LFSR and outputted the least significant bit from the LFSR as a signature at a regular interval. We fixed the signature output interval of each benchmark at 100 cycles or the interval that would result in 5% signature storage area cost, whichever interval is larger. While aliasing will occur with 50% probability in the single-bit signatures, the LFSR maintains a running hash (see Section 1.5.2) that captures the internal behavior of the accelerator with negligible aliasing probability. If the software and hardware LFSRs mismatch, a single-bit signature mismatch will soon follow. At the end of benchmark execution, we dump the full contents of both LFSRs as signatures to insure that any late mismatch in the LSFRs is detected.

### 4.2.1 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 Synopsys Design Compiler 2013-12.sp1, mapping to the 45nm ARM standard cell library, and targeting maximum clock frequency. The area and clock period overheads for each accelerator core are shown in Figure 4.4. Results show a mean accelerator-level area cost of 8.3%. We observe no clock period overhead on average.

### 4.2.2 Logic Bugs

To evaluate the effectiveness of H-QED in detecting logic bugs, we considered bugs in the current and past versions of CHStone [42], as well as bugs in our HLS engine itself.

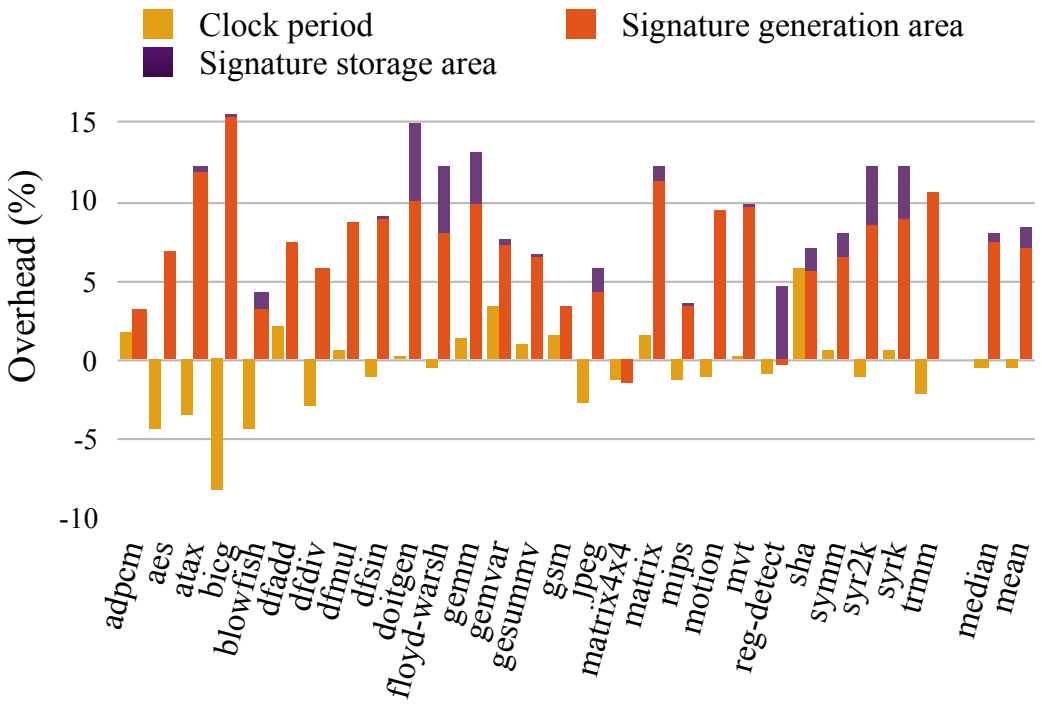



Figure 4.4: Area and performance overheads

CHStone Bugs

For CHStone bugs, we identified all of the bug fixes in the version history and confirmed those bugs with the CHStone authors. For each bug found, we isolated it by fixing all of the other bugs in the last version of CHStone with that bug, creating bug benchmarks containing one known bug each. We ran each buggy benchmark through our H-QED process (Figure 4.2 on page 39), producing hardware and software versions for the same buggy code that we executed to produce signatures for internal behavior and benchmark outputs. We also maintained a third, known correct result for each CHStone benchmark that we used as an additional check for the results of the hardware execution. Table 4.1 enumerates the results of our logic bug experiments. The columns are as follows:

- **Benchmark** indicates the CHStone benchmark containing the bug.
- **Version** indicates the versions of CHStone in which the bug is present.
- **File** and **Line** denote the location of the bug in the last version of the source code that contains the bug.

Table 4.1: Evaluation of H-QED against Logic Bugs in CHStone
(previously unknown bugs highlighted in bold)

| Bench | Vers. | File | Lines | Type | ND | ERC | HQ | Both |
|---|---|---|---|---|---|---|---|---|
| adpcm | 1.1-1.8 | adpcm.c | 686-690 | MLU | no | 21.8k | - | 21.8k |
| jpeg | 1.6-1.9 | decode.c | 206 | *++ | unact | | | |
| | | | 207 | *++ | unact | | | |
| | | | 211 | *++ | unact | | | |
| gsm | 1.1-1.4 | lpc.c | 87-88 | OOB | C only | - | - | - |
| | | | 150-151 | OOB | C only | - | - | - |
| | | | 157-158 | OOB | yes | - | 77 | 77 |
| mips | **1.1-1.10** | **mips.c** | **255** | **INIT** | yes | - | 23 | 23 |
| | **1.11** | **mips.c** | **132-135** | **OOB** | yes | - | 110 | 110 |
| motion | 1.1-1.2 | mpeg2.c | 225-226 | OOB | yes | 90 | 10 | 10 |
| | 1.1-1.4 | getbits.c | 113 | SHFT | C only | 100 | - | 100 |
| | | motion.c | 155 | SHFT | C only | - | - | - |
| | | | 160 | SHFT | yes | 45 | 45 | 45 |
| | | | 166 | SHFT | unact | | | |
| | **1.1-1.10** | **getbits.c** | **134** | **SHFT** | yes | - | 105 | 105 |
| | | | **144** | **SHFT** | yes | - | 91 | 91 |
| | | | 155 | SHFT | unact | | | |
| | | | **Counts** | **17** | **11** | **4** | **7** | **9** |

MLU = Manual loop unrolling omits one iteration
*++ = Wrongly assuming postincrement (++) has lower precedence than dereference (*)
OOB = Out-of-bounds array access
INIT = Read of uninitialized variable
SHFT = Bit shift by out-of-bounds amount

- **Type** provides information about the nature of the bug.
- **ND** indicates if the bug is nondeterministic. Some bugs are nondeterministic at the C-level, but not at the LLVM level. Others are never activated during benchmark execution.
- **ERC** denotes the result of comparing the hardware result with the known correct result. If the bug is caught (i.e. the results do not match), this column notes the latency (in cycles) from bug activation to the end of benchmark execution.
- **HQ** indicates the error detection latency (in cycles) from comparing hardware and software executions through the H-QED process.
- **Both** provides the result of combining (i.e. taking the better of) the ERC and HQ columns.

Of the seventeen bugs we identified in CHStone, only twelve were activated during benchmark execution. Unactivated bugs, by definition, are undetectable. Some unactivated bugs correspond to code that is not used or not needed, in which case such code should be removed since it adds overhead to the hardware accelerator. Other unactivated bugs are the result of coverage limits of the existing test vectors in CHStone. For both cases, traditional code coverage evaluation techniques will allow the hardware designer to identify uncovered code or operand ranges and either remove the functionality or improve the test vectors for full coverage so that all bugs in the code are activated.

Of the twelve activated bugs, one bug involving the omission of an iteration in loop unrolling was deterministic, resulting in identical hardware and software behavior. Comparing the benchmark output with the known correct result catches the bug. As mentioned in Section 1.5.1, this bug could be isolated with traditional software debugging techniques.

For the remaining eleven bugs, all are non-deterministic according to standard C semantics, but for four of these bugs, our compiler infrastructure performed optimization transforms that made the behavior deterministic at the LLVM-IR level. In some cases, compiler transformations actually "fixed" the bug, producing LLVM-IR isomorphic to that generated by the bug-free version of the benchmark. In other cases, the compiler transform replaced the nondeterministic bug with a deterministic variant. In both cases, the compiler identified an undefined operation, and silently, arbitrarily assigned the result (which is legal because the behavior is undefined). Whether the bug is "fixed" or made deterministic is a matter of whether the arbitrary choice the compiler makes happens to correspond to the behavior the programmer intended. Bugs of this nature could be isolated if the compiler optimization passes emitted warnings when they identify nondeterminism. Since the four bugs in question are deterministic at the LLVM level, the hardware and software execution are identical and thus signature comparison does not detect these bugs. One of the bugs in "getbits.c" that was made deterministic, however, was caught by the end result comparison. Provided that the compiler transformations can be reproduced, this bug can also be isolated with traditional software debugging techniques using an LLVM debugger.

The remaining seven bugs are nondeterministic at the LLVM-level and all are caught by H-QED. Of these seven bugs, five are masked, rendering

the end result check unable to detect those bugs. Recall from Section 1.3 that masked bugs are not necessarily benign in validation testing as masking conditions can be affected by the testing environment. Of those five bugs, four (highlighted in bold) were previously unknown bugs in the CHStone benchmark suite that we discovered by using H-QED, suggesting that these nondeterministic, masked bugs are the most difficult bugs to find. We confirmed these new bugs with the CHStone authors.

Overall, in our CHStone bug experiments we observe that of the twelve activated bugs, the end result check catches four of the bugs, H-QED catches seven of the bugs, and that combining the two techniques results in a coverage of nine out of twelve.

HLS Engine Bugs

For HLS engine bugs, we considered bugs that we fixed during the development of our HLS engine. The bugs involve initialization errors for global variables in the JPEG benchmark (mapped to hardware memory). We injected each bug, one at a time, into our hardware design for the JPEG benchmark by modifying the memory initialization procedure for the corresponding variable. Table 4.2 enumerates the bugs and detection results. The "variable" column indicates the global variable affected by the bug. The other columns have the same meaning as Table 4.1.

Table 4.2: Evaluation of H-QED against HLS Engine Logic Bugs

| **Bench** | **Vers.** | **Variable** | **Type** | **ERC** | **HQ** | **Both** |
|---|---|---|---|---|---|---|
| jpeg | 1.1-1.11 | read_position | ZERO | 838k | 179 | 179 |
| | 1.1-1.4 | p_dhtbl_maxcode | NOINIT | - | 249 | 249 |
| | | p_dhtbl_mincode | NOINIT | - | 349 | 349 |
| | | p_dhtbl_valptr | NOINIT | - | 349 | 349 |
| | | **Counts** | **4** | **1** | **4** | **4** |

ZERO = Global variable initialized to zero, ignoring nonzero initializer
NOINIT = Global variable not initialized, ignoring implicit zero initializer

Our experiments show that for the "ZERO" bug, H-QED error detection latency is over three orders of magnitude faster than the end result check. The "NOINIT" bugs are masked, so the end result check is unable to detect them, but H-QED does. Overall, H-QED dominates in error detection coverage

and latency for HLS engine bugs because an HLS engine bug results in a hardware design with different internal behavior, but does not affect the software version. Differences in internal behavior between the hardware and software versions are quickly caught by H-QED.

### 4.2.3 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 [37]: 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 ran 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).

Timing error coverage (number of errors detected divided by the number of errors injected) is presented in Figure 4.5, 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

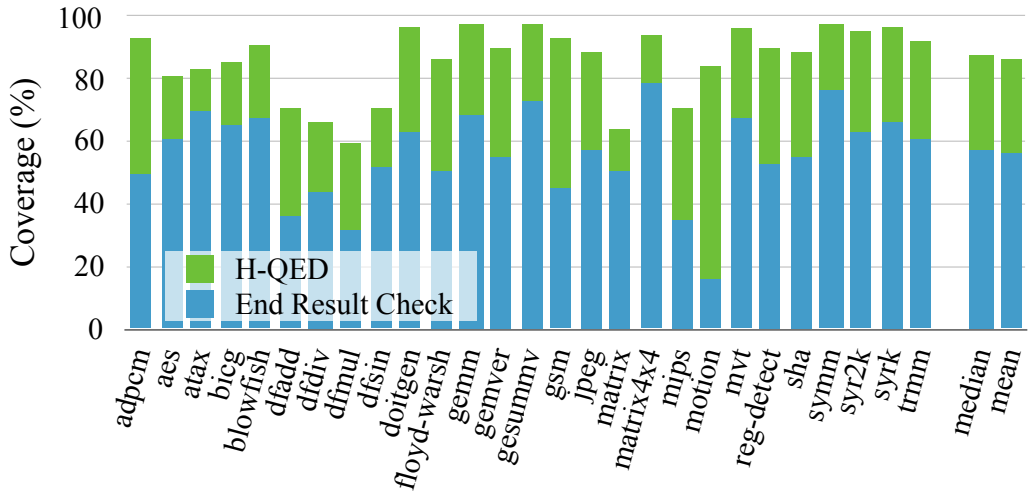



Figure 4.5: Timing error detection coverage

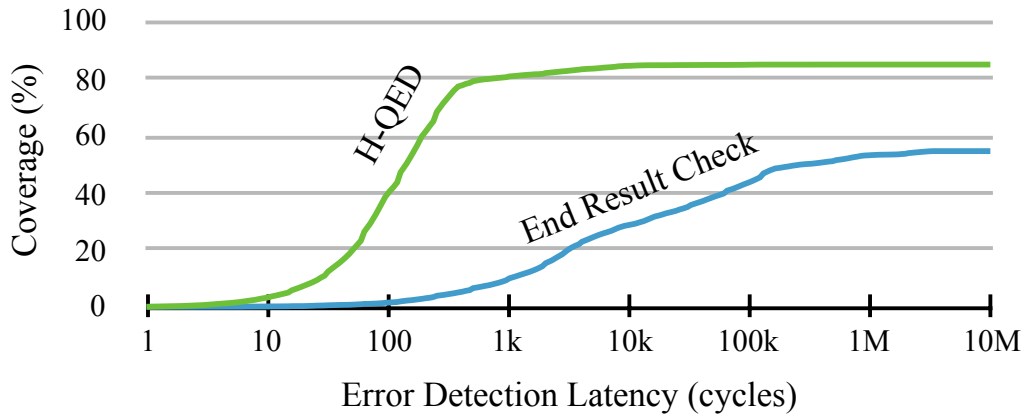



Figure 4.6: Overall timing error coverage as a function of error detection latency

is 100% with H-QED (i.e., we detect all unmasked errors). The overall error detection latency distribution is shown in Figure 4.6. 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.

# CHAPTER 5

# CONCLUSIONS

In our modulo-3 shadow datapath work, we have designed and implemented a fully automated high-level synthesis process to create error detecting cores capable of detecting an average of 99.42% of unmasked errors for an assortment of three different kinds of fault models with negligible delay cost, 25.7% area cost, and a detection latency 4150x faster than an end result check. We have taken the first step towards the fully automated generation of low area cost, low development cost reliable hardware through high-level synthesis. We also explored a rollback recovery method for soft errors with an additional area cost of 28% through which we achieve up to a 175x increase in reliability against soft errors. Future directions related to this research include:

1. adding support for floating-point operations;
2. exploring other modular bases (5,7,9, etc.);
3. fixing timing errors through rollback combined with frequency-voltage scaling.

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 two orders of magnitude improvement in error detection latency, up to threefold improvement in coverage, less than 10% accelerator-level 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 ac-

celerators, 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


[1] K. Campbell, P. Vissa, D. Pan, and D. Chen, "High-level synthesis of error detecting cores through low-cost modulo-3 shadow datapaths," in *DAC*, 2015.

[2] K. Campbell, D. Lin, S. Mitra, and D. Chen, "Hybrid quick error detection (H-QED): Accelerator validation and debug using high-level synthesis principles," in *DAC*, 2015.

[3] J. Keane and C. H. Kim, "Transistor aging," *IEEE Spectrum*, Apr 2011.

[4] J. G. Tryon, "Quadded logic," in *Redundancy Techniques for Computing Systems*, R. H. Wilcox and W. C. Mann, Eds. Spartan Books, 1962.

[5] T. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *MICRO*, 1999, pp. 196–207.

[6] A. Antola, V. Piuri, and M. Sami, "High-level synthesis of data paths with concurrent error detection," in *IEEE Symp. Defect and Fault Tolerance in VLSI Systems*, Nov 1998, pp. 292–300.

[7] K. Wu and R. Karri, "Algorithm level recomputing with allocation diversity: a register transfer level time redundancy based concurrent error detection technique," in *ITC*, 2001, pp. 221–229.

[8] A. Meixner, M. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *MICRO*, Dec 2007, pp. 210–222.

[9] J. von Neumann, "Probabilistic logics and synthesis of reliable organisms from unreliable components," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, 1956, pp. 43–98.

[10] R. Karri, K. Hogstedt, and A. Orailoglu, "Computer-aided design of fault-tolerant VLSI systems," *IEEE Design and Test of Computers*, vol. 13, no. 3, pp. 88–96, Fall 1996.

[11] E. P. Kim, "Statistical error compensation for robust digital signal processing and machine learning," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2014.

[12] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *MICRO*, Dec. 2003, pp. 7–18.

[13] S. Tosun, O. Ozturk, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, and W.-L. Hung, "An ILP formulation for reliability-oriented high-level synthesis," in *ISQED*, March 2005, pp. 364–369.

[14] T. Hong, Y. Li, S.-B. Park, D. Muil, D. Lin, Z. A. Kaleql, N. Hakim, H. Naeimi, D. S. Gardner, and S. Mitra, "QED: Quick error detection tests for effective post-silicon validation," in *ITC*, 2010, pp. 1–10.

[15] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra, "Quick detection of difficult bugs for effective post-silicon validation," in *DAC*, 2012, pp. 561–566.

[16] D. Lin, T. Hong, Y. Li, Eswaran S., S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "Effective post-silicon of system-on-chips using quick error detection," *IEEE Trans. CAD*, vol. 33, no. 10, pp. 1573–1590, Oct. 2014.

[17] D. Lin, Eswaran S., S. Kumar, E. Rentschler, and S. Mitra, "Quick error detection tests with fast runtimes for effective post-silicon validation and debug," in *DATE*, 2015.

[18] A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv, "Threadmill: A post-silicon exerciser for multi-threaded processors," in *DAC*, 2011.

[19] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *ICCD*, 2008.

[20] M. Abramovici, "In-system silicon validation and debug," *IEEE Design and Test of Computers*, vol. 25, no. 3, pp. 216–223, May 2008.

[21] ARM, "CoreSight debug and trace." [Online]. Available: http://www.arm.com/products/system-ip/coresight

[22] S. B. Park, 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.

[23] S.-B. Park, A. Bracy, H. Wang, and S. Mitra, "BLoG: Post-silicon bug localization in processors using bug localization graph," in *DAC*, 2010, pp. 368–373.

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

[25] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors – a survey," *IEEE Trans. Computers*, vol. 37, no. 2, pp. 160–174, Feb. 1988.

[26] N. R. Saxena, S. Fernandez-Gomez, W. J. Huang, S. Mitra, S. Y. Yu, and E. J. McCluskey, "Online testing in adaptive and configurable systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 29–41, Jan.-Mar. 2000.

[27] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Fingerprinting: Bounding soft-error detection latency and bandwidth," in *ACM ASPLOS*, 2004, pp. 224–234.

[28] E. S. Sogomonyan, A. Morosov, M. Gössel, A. Singh, and J. Rzeha, "Early error detection in system-on-chip for fault-tolerance and at-speed debugging," in *IEEE VLSI Test Symp.*, 2001, pp. 184–189.

[29] R. Karri and A. Orailoglu, "High-level synthesis of fault-secure microarchitectures," in *DAC*, 1993, pp. 429–433.

[30] S. Mitra, N. R. Saxena, and E. J. McCluskey, "Fault escapes in duplex systems," in *IEEE VLSI Test Symp.*, 2000, pp. 453–458.

[31] N. R. Saxena and E. J. McCluskey, "Dependable adaptive computing systems," in *IEEE Systems, Man, and Cybernetics Conf.*, 1998, pp. 2172–2177.

[32] X. Feng and A. J. Hu, "Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification," in *DAC*, 2006, pp. 1063–1068.

[33] M. Fujita, "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.

[34] A. Mathur, M. Fujita, E. Clarke, and P. Urard, "Functional equivalence verification tools in high-level synthesis flows," in *IEEE Design and Test of Computers*, 2009, pp. 88–95.

[35] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. on Embedded Computing Systems*, vol. 13, no. 2, Sep. 2013.

[36] S. Piestrak, F. Pedron, and O. Senlieys, "VLSI implementation and complexity comparison of residue generators modulo 3," in *EUSIPCO*, 1998, pp. 511–514.

[37] M. Gao, P. Lisherness, and T. Cheng, "On error modeling of electrical bugs for post-silicon timing validation," in *ASPDAC*, 2012, pp. 701–706.

[38] L.-N. Pouchet and T. Yuki, "PolyBench/C 3.2." [Online]. Available: http://www.cse.ohio-state.edu/~pouchet/software/polybench/

[39] A. Bogorad, J. Likar, R. Lombardi, S. Stone, and R. Herschitz, "On-orbit error rates of RHBD SRAMs: Comparison of calculation techniques and space environmental models with observed performance," *IEEE Transactions on Nuclear Science*, vol. 58, no. 6, pp. 2804–2806, Dec 2011.

[40] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design and Test of Computers*, vol. 26, no. 4, 2009.

[41] K. Rupnow, Y. Liang, Y. Li, and D. Chen, "A study of high-level synthesis: Promises and challenges," in *IEEE Intl. Conf. on ASIC*, 2011, pp. 1102–1105.

[42] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, 2009.

[43] J. Gray, "Why do computers stop and what can be done about it?" *Tandem Computer, Tech. Report*, vol. 85.7, 1985.

[44] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, 2004, pp. 75–86.

[45] C. Lattner, "LLVM and Clang: Advancing compiler technology," keynote talk, *FOSDEM*, 2011.