ROBUST AND RELIABLE HARDWARE ACCELERATOR DESIGN THROUGH HIGH-LEVEL SYNTHESIS

BY

KEITH A. CAMPBELL

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Deming Chen, Chair
Professor Wen-Mei W. Hwu
Professor Martin D. F. Wong
Associate Professor Nam Sung Kim

ABSTRACT

System-on-chip design is becoming increasingly complex as technology scaling enables more and more functionality on a chip. This scaling-driven 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 dissertation shows that high-level synthesis also has the power to address validation and reliability challenges through three automated solutions targeting three key stages in the hardware design and use cycle: pre-silicon debugging, post-silicon validation, and post-deployment error detection.

Our solution for rapid pre-silicon debugging of accelerator designs is *hybrid tracing*: comparing a datapath-level trace of hardware execution with a reference software implementation at a fine temporal and spatial granularity to detect logic bugs. An integrated backtrace process delivers source-code meaning to the hardware designer, pinpointing the location of bug activation and providing a strong hint for potential bug fixes. Experimental results show that we are able to detect and aid in localization of logic bugs from both C/C++ specifications as well as the high-level synthesis engine itself.

A variation of this solution tailored for rapid post-silicon validation of accelerator designs is *hybrid hashing*: 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 hybrid hashing, 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. Hybrid hashing also uncovered previously unknown bugs in the CHStone benchmark suite, which is widely used by the HLS community. Hybrid hashing 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 hybrid hashing.

Finally, our solution for post-deployment error detection 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 (4150× 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 175× increase in reliability against soft errors.

While the area cost of our modulo shadow datapaths is much better than traditional modular redundancy approaches, we want to maximize the applicability of our approach. To this end, we take a dive into gate-level architectural design for modulo arithmetic functional units. We introduce new low-cost gate-level architectures for all four key functional units in a shadow datapath: (1) a modulo reduction algorithm that generates architectures consisting entirely of full-adder standard cells; (2) minimum-area modulo adder and subtractor architectures; (3) an array-based modulo multiplier design; and (4) a modulo equality comparator that handles the residue encoding produced by the above.

We compare our new functional units to the previous state-of-the-art

approach, observing a 12.5% reduction in area and a 47.1% reduction in delay for a 32-bit mod-3 reducer; that our reducer costs, which tend to dominate shadow datapath costs, do not increase with larger modulo bases; and that for modulo-15 and above, all of our modulo functional units have better area and delay then their previous counterparts. We also demonstrate the practicality of our approach by designing a custom shadow datapath for error detection of a multiply accumulate functional unit, which has an area overhead of only 12% for a 32-bit main datapath and 2-bit modulo-3 shadow datapath.

Taking our reliability solution further, we look at the bigger picture of modulo shadow datapaths combined with other solutions at different abstraction layers, looking to answer the following question: Given all of the existing reliability improvement techniques for application-specific hardware accelerators, what techniques or combinations of techniques are the most cost-effective? To answer this question, we consider a soft error fault model and empirically evaluate cross-layer combinations of ABFT, EDDI, and modulo shadow datapaths in the context of high-level synthesis; parity in logic synthesis; and flip-flop hardening techniques at the physical design level. We measure the reliability benefit and area, energy, and performance cost of each technique individually and for interesting technique combinations through FPGA emulated fault-injection and physical place-and-route. Our results show that a combination of parity and flip-flop hardening is the most cost-effective in general with an average 1.3% area cost and 5.7% energy cost for a 50× improvement in reliability. The addition of modulo-3 shadow datapaths to this combination provides some additional benefit for some applications, even without considering its combinational logic, stuck-at fault, and timing error protection benefits. We also observe new efficiency challenges for ABFT and EDDI when used for hardware accelerators.

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, Yao Chen for complimenting my ideas and treating me like a professional, Wei Zuo for her honest assessments of my work, and Anand Ramachandran for interesting conversations.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS ix

CHAPTER 1 INTRODUCTION 1
1.1 Root Causes for Hardware Failure 3
1.2 Root Cause Effects 8
1.3 Error Propagation 10

CHAPTER 2 BACKGROUND 12
2.1 Execution Signatures 12
2.2 Modulo Arithmetic 14
2.3 High-Level Synthesis 17

CHAPTER 3 RELATED WORK 19
3.1 Hybrid Quick Error Detection 19
3.2 Modulo Shadow Datapaths 21
3.3 Cross-Layer Reliability 23

CHAPTER 4 HYBRID QUICK ERROR DETECTION 25
4.1 Basic Principles 25
4.2 Hybrid Tracing vs. Hybrid Hashing 28
4.3 Effectiveness and Practicality 29

CHAPTER 5 PRE-SILICON DEBUG: HYBRID TRACING 31
5.1 Comparison to Software Debugging 32
5.2 Hybrid Tracing Framework 33
5.3 Simulation Breakpoint Trigger 42
5.4 Bug Example 42
5.5 Experimental Results 43

CHAPTER 6 POST-SILICON VALIDATION: HYBRID HASHING . 52
6.1 Hybrid Hashing Framework 54
6.2 Binding to Minimize Area 60
6.3 Integration into PSV Testing 60
6.4 Real-time Error Detection 61
6.5 Experimental Results 62

CHAPTER 7 POST-DEPLOYMENT RESILIENCE: MODULO-3 SHADOW DATAPATHS 66
7.1 Framework 66
7.2 Results and Analysis 77

CHAPTER 8 CHEAPER MODULO FUNCTIONAL UNITS 82
8.1 Modulo Functional Units Architecture 82
8.2 Quality of Results Comparisons 88

CHAPTER 9 CROSS-LAYER RESILIENCE SYNERGIES 92
9.1 Framework 93
9.2 Results and Analysis 101

CHAPTER 10 CONCLUSIONS 107

REFERENCES 110

LIST OF ABBREVIATIONS

ABFT	Algorithm Based Fault Tolerance
ACM	Association for Computing Machinery
ARM	Company that designs CPU cores, initially an acronym for Acorn RISC Machine
ASIC	Application Specific Integrated Circuit
AST	Abstract Syntax Tree
BTI	Bias Temperature Instability
CDFG	Control and DataFlow Graph
CED	Concurrent Error Detection
CHStone	C-based High-level Synthesis benchmark suite
CLEAR	Cross-Layer Exploration for Architecting Resilience
CPU	Central Processing Unit
DAC	Design Automation Conference
DICE	Dual Interlocked Storage Cell
DIVA	Dynamic Implementation Verification Architecture, a fault-tolerant CPU architecture
DMR	Double Modular Redundancy
DRAM	Dynamic Random Access Memory
ECO	Engineering Change Order
EDDI	Error Detection by Duplicated Instructions
EDL	Error Detection Latency

ERC	End Result Check
FA	Full Adder
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GCC	GNU Compiler Collection
GNU	GNU is Not Unix
GPU	Graphics Processing Unit
HA	Half Adder
HH	Hybrid Hashing
HLS	High-Level Synthesis, also known as behavioral synthesis
HT	Hybrid Tracing
HW	HardWare
IC	Integrated Circuit
ID	IDentifier
IEEE	Institute of Electrical and Electronics Engineers
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
LEAP	Layout design through Error-Aware transistor Positioning
LFSR	Linear Feedback Shift Register
LHL	Light-Hardened LEAP
LLVM	An open-source compiler development framework, initially an acronym for Low-Level Virtual Machine
LUT	LookUp Table
MAC	Multiply and ACcumulate

MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MSB	Most Significant Bit
MUX	MUltipleXer
P&R	Place and Route
PSV	Post-Silicon Validation
QA	Quarter Adder
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
SDC	Silent Data Corruption
SEC	Statistical Error Compensation
SEMU	Single Event Multiple Upsets
SER	Soft Error Rate
SEU	Single Event Upset
SoC	System on a Chip
SP	Service Pack, a minor software update
SP&R	Synthesis, Place, and Route
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

INTRODUCTION

Designing hardware is hard.[1] 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 microprocessor-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 paradigm to the hardware design process, enabling hardware designers to

[1] That is why it is called *hard*ware.

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, we discuss our research to leverage this power of HLS to address the aforementioned hardware validation and reliability problems through three automated solutions, targeting three key stages of the hardware design and use cycle.

In Chapter 5, we propose the insertion of non-synthesizable instrumentation into an HLS-generated hardware design to capture a trace of internal behavior at a fine spatial and temporal granularity in hardware (RTL) simulation. By comparing this trace with a software version generated to produce the same result, we show that logic bug detection is possible for both bugs in the hardware specification source and in the HLS engine. Furthermore, through the use of debugging metadata, we show that this technique can pinpoint the line where a source-code bug resides. This technique also leverages co-simulation to use high-level language simulation for parts of the design not being directly tested. HLS is critical here because it identifies key RTL variables that have source-code meaning, avoiding the deluge of data from an RTL-level value change dump.

In Chapter 6, we propose the insertion of signature generation logic into a fabricated hardware design to create a heavily compressed signature stream that captures the internal behavior of the design during post-silicon validation at a fine temporal and spatial granularity. By comparing the generated sequence of signatures to a reference set generated by high-level simulation, we 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.

In Chapter 7, we 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 8, we take a dive into gate-level optimization to further optimize these shadow datapaths, exploring new gate-level algorithms and architectural templates for modulo arithmetic functional units with the goal of automating the generation of these units. We show that the use of these new functional units reduces shadow datapath cost, and enables practical scaling to larger shadow datapath widths for improved error detection effectiveness.

In Chapter 9, we take shadow datapaths further by looking for cross-layer synergies with techniques for improving soft-error reliability ranging from the algorithm to the physical design level. By combining techniques, we can exploit the strength of each technique while compensating for weaknesses. As a side effect, this chapter explores the effectiveness of algorithm and instruction level techniques when applied in the context of high-level synthesis.

Before these main chapters, we will provide some background on the reliability and validation problems hardware designers face in the rest of this chapter, introduce important concepts used in our work in Chapter 2, and discuss related work in Chapter 3. Chapter 4 introduces the concept of hybrid error detection, which creates the foundation for Chapters 5 and 6. We end with concluding remarks in Chapter 10.

This thesis is based on our three publications in the IEEE/ACM Design Automation Conferences of 2015 and 2016: "High-Level Synthesis of Error Detecting Cores through Low-Cost Modulo-3 Shadow Datapaths" [1], "Hybrid Quick Error Detection (H-QED): Accelerator Validation and Debug Using High-Level Synthesis Principles" [2], and "Debugging and Verifying SoC Designs through Effective Cross-Layer Hardware-Software Co-Simulation" [3]. Chapter 9 is based on a publication to appear in TECHCON 2017: "Cost-Effective Cross-Layer Resilience for Hardware Accelerators" [4].

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.

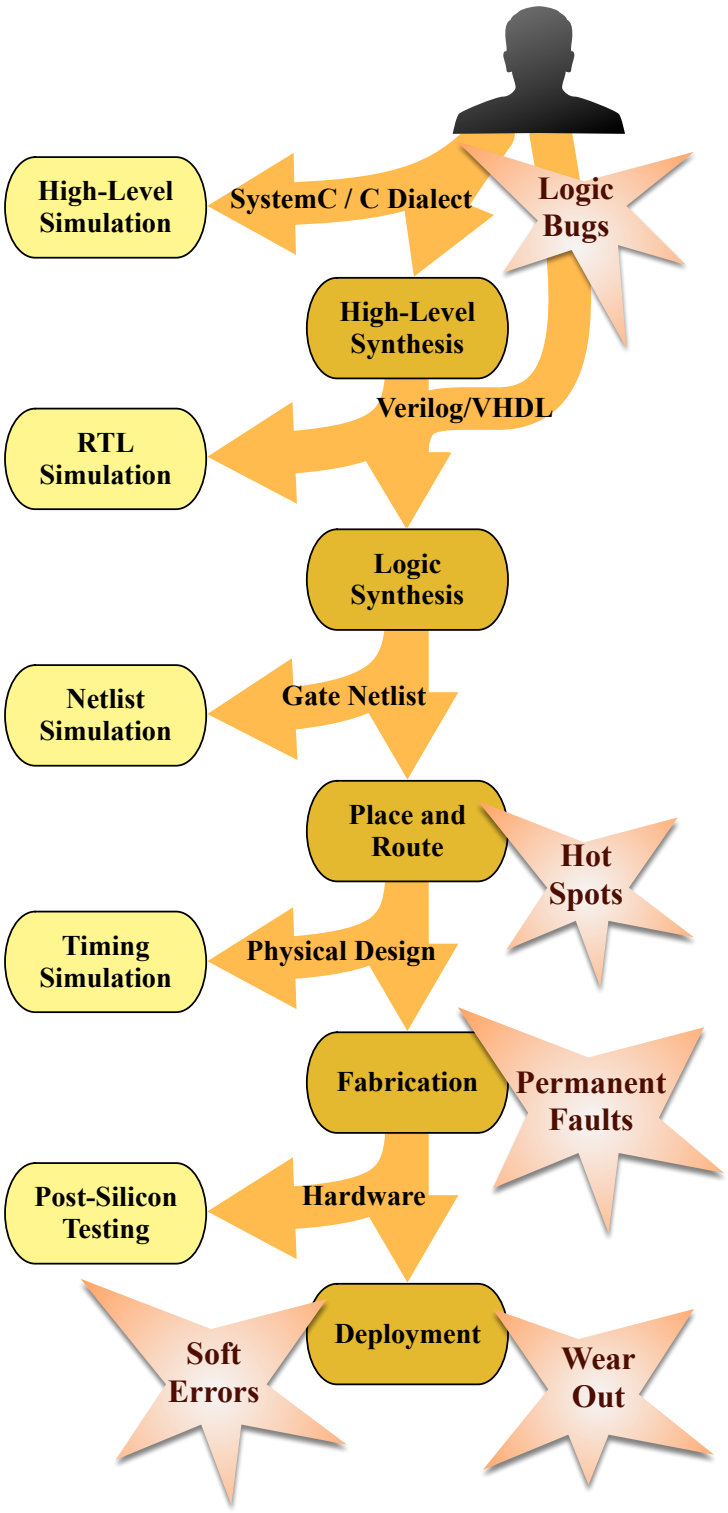


Figure 1.1: The hazards inherent in designing custom hardware.

For improved productivity, the designer may also elect to specify design blocks at the behavioral level in SystemC or the 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.

- **Non-deterministic 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, non-deterministic semantics may be specified as resulting in "undefined behavior." An example of a non-deterministic 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 [5]. 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 [5].

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 causes 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.

CHAPTER 2

BACKGROUND

Each of the sections in this chapter provides some useful background information for the convenience of the reader who may be unfamiliar with some of the concepts in the chapters that follow.

2.1 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 6, we 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.

2.1.1 Catching Logic Bugs

If a design contains a non-deterministic 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 comparison 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.

2.1.2 Hash Functions

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

$$H(x_1, x_2, ..., x_n) = x_1 \oplus x_2 \oplus ... \oplus x_n \tag{2.1}$$

$$S_n = \begin{cases} H_0 \oplus C & \text{if } n = 0 \\ H_n \oplus \text{rotate}(S_{n-1}, r) & \text{if } n > 0 \end{cases} \tag{2.2}$$

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 the 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 Chapter 6, we 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 (2.2) has the additional desirable property that S_n depends on the number of cycles that have passed, n, even if all $H_n = 0$.

2.2 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 2.1 shows the mapping from integer space to modulo-3 space and Table 2.2 provides the modulo-3 addition, subtraction, and multiplication tables.

2.2.1 Properties

Since equivalent lightweight computations can be performed in modulo space as in integer space, modulo 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 arithmetic. In other words, given integers $\{x, y, z\}$ and corresponding modulo variables $\{x', y', z'\} = \{x, y, z\} \bmod b$ we observe the following properties:

$$x + y = z \implies x' + y' = z' \pmod{b} \tag{2.3}$$

Table 2.1: 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 2.2: 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

$$x - y = z \implies x' - y' = z' \pmod{b} \tag{2.4}$$

$$xy = z \implies x'y' = z' \pmod{b} \tag{2.5}$$

where (mod b) next to an equation indicates that the arithmetic is performed in modulo-b space. Thus for Equations (2.3), (2.4), (2.5), z' can be independently computed two ways: by mapping z to modulo space or by mapping x' and y' to modulo space and performing the "shadow computation" in each equation.

Note that this "shadow computation" property holds for arbitrarily complex 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.

2.2.2 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 2.1) 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 2.1 and 2.2). Thus, in our application of modulo-3 arithmetic, we pay special attention to multiplication operations (see Section 7.1.2).

2.2.3 Modular Base

To use modulo-b arithmetic to detect errors effectively in binary logic, we 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). We also want each bit in z to have the ability to affect any bit in z' to reduce the 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 Chapters 7 and 9, we choose $b = 3$ to minimize the hardware cost, as only two bits are needed to represent the three possible modulo-3 values.

2.2.4 Mersenne Numbers

For positive integers n we define the Mersenne numbers by $M(n) = 2^n - 1$. The use of $M(n)$ as a modulo base has the following useful property for $n \geq 2$:

$$2^n = 1 \pmod{M(n)} \tag{2.6}$$

2.2.5 Binary Representations

Our encodings for modulo residues are based on the standard binary representation for integers, where bits have weights with successive powers of two. In other words the integer value of a particular sequence $\{b_{n-1}, b_{n-2}, ..., b_0\}$ of bits is defined as:

$$v = \sum_{i=0}^{n-1} 2^i b_i \tag{2.7}$$

A standard Mersenne number residue r with base $M(n)$ will be in the range $0 \leq r \leq M(n) - 1 = 2^n - 2$. Thus n bits are sufficient to encode a residue with base $M(n)$, and the most significant bit (MSB), b_{n-1}, will have weight 2^{n-1}. If a carry bit is generated from adding two MSB bits, it will have weight 2^n which is equivalent to 1 by application of Equation (2.6).

2.2.6 Normalization

There is one special encoding possible for an $M(n)$ residue encoded with n bits, the value where all bits $b_i = 1$. This encoding has the integer value $2^n - 1 = M(n)$ by Equation (2.7). Since this residue is the same as the modulo base, it is equivalent to zero. We call this special encoding for zero the *denormalized* encoding of zero, write it as -0, and call encodings that allow it *non-normalized* encodings.

2.3 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 3

RELATED WORK

3.1 Hybrid Quick Error Detection

The inspiration for H-QED is QED [6–9], which is a software technique for the validation of programmable microprocessors. In general, validation techniques that target processors (e.g., [10, 11] and others) are inadequate for bugs inside accelerators.

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 [12–14]. The goal is to detect bugs in the implementation that are caused by the HLS tool. However, formal equivalence checking techniques are limited in their capacity to handle HLS transformations and this limitation is further compounded by the large state space of HLS implementations. In contrast, H-QED is a dynamic technique that integrates into the HLS engine to follow instructions through HLS transformations and to generate the corresponding software reference implementation. H-QED can be run in pre-silicon simulation at RTL simulation speeds (with acceptable overhead) or during post-silicon validation at full hardware speed.

3.1.1 Hybrid Tracing

Prior works such as [15, 16] perform source-level transformations to create external ports for selected signals to improve observability. However, this approach requires manual source code instrumentation. Furthermore, source instrumentation interferes with compiler optimizations, creating intrusiveness. A hardware-software runtime trace comparison technique is proposed in [17,18] to provide automated HW/SW discrepancy detection. Both techniques use a

mapping between software variables and hardware components through LLVM variables to detect discrepancies and assist debug. Again, these techniques are intrusive as they insert additional error detection operations that change the schedule of the hardware design. In contrast, hybrid tracing instrumentation is integrated into HLS to eliminate intrusiveness, creating an RTL design with nonintrusive debugging annotations that can easily be removed before synthesis.

3.1.2 Hybrid Hashing

Although hybrid hashing may appear to be similar to tracing techniques used in PSV (e.g., using trace buffers or system memory [19–22]), there are important differences:

1. Hybrid hashing systematically collects signatures, unlike tracing techniques that are often ad-hoc or based on heuristics.

2. Hybrid hashing does not require extensive low-level (e.g., RTL) simulation.

3. Hybrid hashing does not require designer-crafted assertions.

4. Hybrid hashing enables very short error detection latencies and high bug coverage, unlike tracing techniques that become ineffective for difficult bugs with long error detection latencies.

Hybrid hashing is distinct from fault-tolerant computing techniques for processors (e.g., using watchdog processors, DIVA, multi-threading and signature techniques for duplex systems [23–28]). Many of these techniques only check the register values as defined by the Instruction Set Architecture (ISA). In contrast, hybrid hashing 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 [29–31], hybrid hashing 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.

3.2 Modulo Shadow Datapaths

3.2.1 Low-Level Fault Resilience

There are many existing approaches to fault resilience. The classical approach is modular redundancy [32, 33], duplicating the entire hardware module and comparing the outputs for discrepancies. Such an approach has $2\times$ – $3\times$ area cost, which is prohibitively expensive and negates the benefits of Moore's law scaling.

Razor logic [34, 35], an approach involving creating a shadow latch for each flip-flop in a design, has been proposed to address timing errors, but also imposes timing constraints on a design. Flip-flop hardening techniques [36, 37] have been proposed to address soft errors in flip-flops, but such techniques do not protect combinational logic. Logic parity [38] is another technique for protecting flip-flops by adding a parity flip-flop for flip-flop clusters with parity prediction and checking logic. Such parity techniques are practically limited to protecting only the flip-flops in a design using the aforementioned clustering technique [38] due to the high overheads (e.g., around 30% area overhead for a 32-bit adder [39]) associated with parity prediction across functional units.

While razor logic, flip-flop hardening, and parity are limited to certain kinds of faults and certain parts of a datapath, modulo shadow datapaths have none of these limitations. Modulo shadow datapaths holistically protect the entire datapath from input to output, including all of the combinational logic. Modulo shadow datapaths is a general purpose error detection technique with essentially no assumptions about fault behavior.

3.2.2 High-Level Error Resilience

There are also a number of high-level error resilience techniques, which are related to (or may be leveraged in) our high-level synthesis approaches in Chapters 7 and 9. DIVA [23] is a popular technique which uses an extra checker core to verify the correctness of a main core computation and commit only non-faulty results. Concurrent error detection (CED) [40] 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. In [41], Wu and Karri use 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 [42] 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.

In [43], Karri et al. 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 [44]. Tosun et al. [45] proposed a technique to recover from soft errors but do not perform any error injection experiments and has a passive approach to masking errors whereas we actively detect and correct errors.

Finally, Algorithm-Based Fault Tolerance (ABFT) [46, 47] is an algorithm-level technique for protecting linear vector and matrix computations by predicting and checking the sums of groups of output elements. ABFT can involve expensive extra memory accesses for checksum computation and storage and may require the duplication of vectors for certain computations. In Chapter 9, we empirically show that these costs make cost-effective application of ABFT to reliable accelerator designs difficult.

3.2.3 Modulo Arithmetic Functional Units

For small modulo bases, a lookup table based approach has been used for basic functional units [48] with explicit *don't cares* inserted to provide hints to the logic synthesis engine for inputs combinations that should never occur.

A reducer is built with a tree of such lookup-table based modulo adders [48]. Such an approach is impractical for larger bases due to exponential scaling.

Piestrak et al. propose a design for a modulo-3 reducer consisting of full-adder (FA) cells and interleaved inverters [48, 49] which exploits the fact that for a given bit $b \in \{0, 1\}$, $2b = -b = 3 - b = 2 + (1 - b) \pmod 3$. In other words, bits of weight 2 can be inverted and treated as a bit of weight 1 with a constant offset (which can be lumped together at the end) so that all bits have the same weight of 1 and can be passed through stages of FAs. While this design may appear superficially similar to our reducer design in Figure 8.1b on page 83, our design uses a more general strategy inspired by Wallace trees that does not require separate inverters. Furthermore our strategy generalizes to any Mersenne base while their design trick is limited to modulo-3 arithmetic.

For cryptography applications, there are also a number of hardware accelerator designs for accelerating modulo exponentiation of large (e.g. 256-bit) numbers which is performed with a series of modulo multiplies [50]. These designs use application-specific algorithms (e.g. Montgomery multiplication [51]) that make them very specialized for big-integer modulo exponentiation, and thus unsuitable for reliability applications.

3.3 Cross-Layer Reliability

The CLEAR study [38] was a cross-layer approach to finding the most cost-effective way to improve flip-flop soft error reliability in programmable microprocessors, considering both software and hardware transformations and their combinations to improve reliability. But, with Dennard scaling ending, the status quo of using programmable microprocessors for all computation is disrupted by hardware accelerators that proliferate on SoCs and embedded FPGAs due to their performance and energy benefits. Thus, a complete reliability solution for modern complex SoCs must consider both microprocessors and accelerators.

In Chapter 9, we take a cross-layer approach to the reliability problem for *application specific hardware accelerators*. While existing coding techniques address soft errors in memories and CLEAR [38] addressed soft errors in microprocessor flip-flops; cost-effective error resilience for flip-flops in hardware

accelerators remains a challenging problem.

Compared to the microprocessor reliability problem, the hardware accelerator reliability problem poses some unique opportunities and challenges. As shown in [38], when limiting processor cores to running specific applications, combinations incorporating algorithm-level techniques like algorithm-based fault tolerance (ABFT) correction can further reduce energy overheads. In general-purpose processor cores, imposing such limitations are not always possible; however, the application-specific nature of accelerators allows us to fully explore these opportunities.

Additionally, in application-specific hardware accelerators, most of the software parts of the stack are removed since the algorithm is hard-wired into the hardware logic. Thus, algorithm and instruction overhead (which manifested as execution time overhead on microprocessors) is translated into hardware overhead. On the other hand, this translation into hardware results allows for optimizations of the software techniques at the hardware level by leveraging hardware customization not previously possible in general-purpose processors. Finally, given the interest of implementing application-specific accelerators not only on custom application-specific integrated circuits (ASICs) but also in an agile manner by utilizing reconfigurable field-programmable gate arrays (FPGAs), it is necessary to explore and understand the implications for reliability when changing the underlying hardware assumptions.

CHAPTER 4

HYBRID QUICK ERROR DETECTION

In this chapter, we introduce the basic concepts for the Hybrid Quick Error Detection (H-QED) technique to overcome validation and debugging challenges for non-programmable hardware accelerators on 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 [6–9]. 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) [52, 53] 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. H-QED can be applied to both pre-silicon and post-silicon validation and debugging scenarios and provides effective source-code bug localization.

4.1 Basic Principles

The basic principles of H-QED are illustrated in Figure 4.1, which involves three key components:

- fine-grained debugging instrumentation
- simulation/execution with two diverse toolchains
- comparison of the instrument outputs

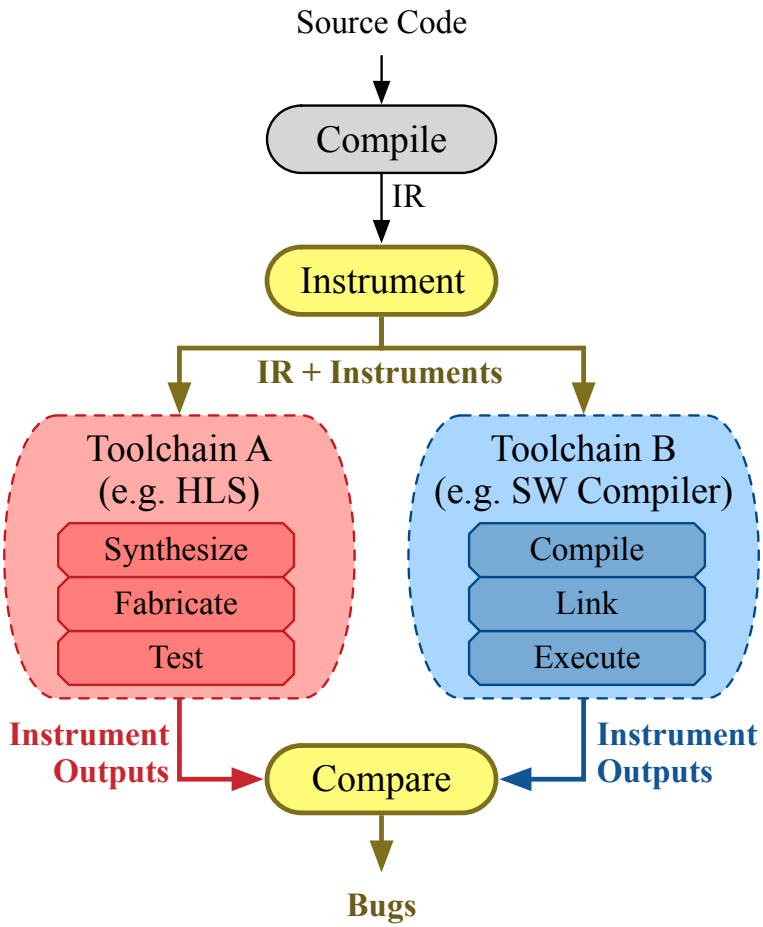


Figure 4.1: Basic working principles of H-QED. A toolchain can be any process that executes or simulates the design. IR = Intermediate Representation.

The instrument pass adds debugging logic or instructions that cause the design to generate output which is a function of the design's internal state. This instrumentation can be translated to unsynthesizable code for pre-silicon validation or signature generation logic for post-silicon validation. The two toolchains are abstract processes that perform transformations on the design leading to some execution or simulation that produces the runtime output of the instruments. When a design contains a non-deterministic bug, it is unlikely that two diverse toolchains will generate models and/or physical designs with identical external *and internal* behavior. Fine-grained instruments will capture this behavioral discrepancy resulting in two different instrument outputs and the bug will be caught. Because the instruments are fine-grained, the error detection latency will be low and many masked bugs will also be caught. Thus H-QED should quickly detect any bug in the source code that results in non-deterministic behavior. It is also clear from Figure 4.1 that H-QED can detect toolchain bugs as a bug in one toolchain that affects the instrument outputs in that branch will result in a mismatch with the outputs produced by the other toolchain. When one "toolchain" involves a full IC fabrication and testing process, this capability of H-QED is particularly important for detecting electrical bugs.

While different variations of this process are possible (in particular integrating the instrumentation pass into parts of the two toolchains is beneficial as we will see), the key invariant is that the instruments must produce the same output in both toolchains if the design is free of bugs. In order to achieve this, the instruments must generate output that is a function of deterministic variables, e.g. output cannot depend on signals in unknown states. Furthermore, the order of the instrument output must be preserved in both toolchains. Finally, an important constraint from a practical point of view is to insert instruments so as to avoid *intrusiveness*, meaning that the external *and internal* behavior of a design should not change when instruments are added. Instruments that change the behavior of a design can also change the behavior of bugs in that design, undermining its bug detection and localization benefit.

4.2 Hybrid Tracing vs. Hybrid Hashing

Since pre-silicon validation and post-silicon validation have different constraints for inserting instruments into a design, we created different variations of H-QED for each scenario. In this thesis, we present both adaptations of H-QED: a pre-silicon tailored adaptation called *hybrid tracing* which we discuss in Chapter 5 and a post-silicon tailored adaptation called *hybrid hashing* which we discuss in Chapter 6. We now provide a brief comparative overview of each adaptation.

4.2.1 Hybrid Tracing

In pre-silicon validation, simulation time is the primary challenge. As mentioned before, simulation speeds are many orders of magnitude slower than real-time, limiting testing coverage. To address this problem, hybrid tracing leverages HLS to select RTL signals for only CDFG variables, minimizing overhead. Hybrid tracing also leverages co-simulation to use high-level language simulation for parts of the design not being directly tested. As mentioned before, high-level language simulation is 1000× faster than RTL simulation. Compared to hybrid hashing, hybrid tracing has the following unique advantages:

1. Hybrid tracing instrument output produces full variable values with unsynthesizable constructs, making the design state fully visible.

2. Hybrid tracing leverages this information to pinpoint the source code location for bug activation.

3. Hybrid tracing instruments are easily removed or ignored from the generated RTL when the design passes validation and is ready for synthesis.

4.2.2 Hybrid Hashing

In post-silicon validation, minimizing area and instrument output bandwidth costs are the primary challenges. To meet bandwidth constraints, hybrid hashing probes internal CDFG variable values and reduces those values with

a running hash function, reducing output bandwidth to a single bit for each multi-cycle interval. The hash function logic has significant area cost, so it is shared as much as possible with multiplexers by scheduling probes for a "non-temporary" variable subset (variables with long lifetimes and thus also scheduling flexibility). Compared to hybrid tracing, hybrid hashing has the following unique advantages:

1. Hybrid hashing instruments are designed to be synthesized as lightweight hardware integrated into an accelerator, enabling bugs to be caught (electrical bugs in particular) that could not be caught in pre-silicon validation due to model limits.

2. Hybrid hashing has essentially no performance impact and allows a manufactured IC to run at full speed.

4.3 Effectiveness and Practicality

As a preview for our experimental results when applying H-QED, we observe the following, demonstrating the effectiveness and practicality of the technique:

1. H-QED enables 2–3 orders of magnitude improvement in error detection latencies for both electrical bugs and logic bugs vs. validation techniques using end-result-checks that compare accelerator outputs against known correct outputs.

2. H-QED uncovered two previously unknown logic bugs in the widely used CHStone HLS benchmark suite [54].

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

4. H-QED allows accelerators to operate in "native" mode (similar to normal system operation) and has a minimal intrusiveness impact. (Incorporation of H-QED continues to detect bugs that are detected by traditional validation techniques.)

5. Hybrid tracing detects all non-deterministic logic bugs in CHStone within one cycle.

6. Hybrid tracing pinpoints where in the source code the bug activates and provides a strong hint for possible bug fixes.

7. Hybrid hashing improves electrical bug (timing error) coverage by up to 3× compared to PSV techniques using end-result-checks.

8. Hybrid hashing incurs an 8% accelerator area overhead and negligible performance costs.

Chapters 5 and 6 discuss our pre-silicon and post-silicon H-QED implementations in detail. Chapter 5 discusses the hybrid tracing technique and Chapter 6 discusses the hybrid hashing technique.

CHAPTER 5

PRE-SILICON DEBUG: HYBRID TRACING

We call our pre-silicon variation of H-QED *hybrid tracing* since we use uncompressed traces of variable values for the instrument outputs in Figure 4.1 on page 26. Hybrid tracing can be used for both module-level pre-silicon verification of HLS-produced RTL as well as pre-silicon integration testing — verification of multiple RTL modules and software on a CPU into a system. Although module-level testing is important, integration testing invariably detects additional bugs that went undetected due to insufficient module-level test vectors or bugs that relate to integration (e.g. a module that works perfectly with the expected number of input data items, but another module sends the wrong amount of data). In both cases, the goal of hybrid tracing is to detect logic bugs as RTL-level simulation models only logic, not electrical behavior.

As illustrated in Figure 5.1, hybrid tracing enables hardware designers to isolate logic bugs by swapping between C/C++ reference implementations and RTL implementations with HLS. Thus, designers can validate complex designs piecemeal, selecting one module at a time to integrate with the rest of the system for verification. Note that the designer of a target module only needs high-level C/C++ models of the system it interfaces with, which need *not* be synthesizable, enabling early stage integration testing for parts that *are* synthesizable. Our framework compares the series of module output values for discrepancies between the software model and the RTL implementation. When validation reveals a problem due to non-deterministic behavior, our code instrumentation, trace comparison and back-tracing steps (discussed in Section 5.2) provide the hardware designer with C/C++ locations where the discrepancies occur.

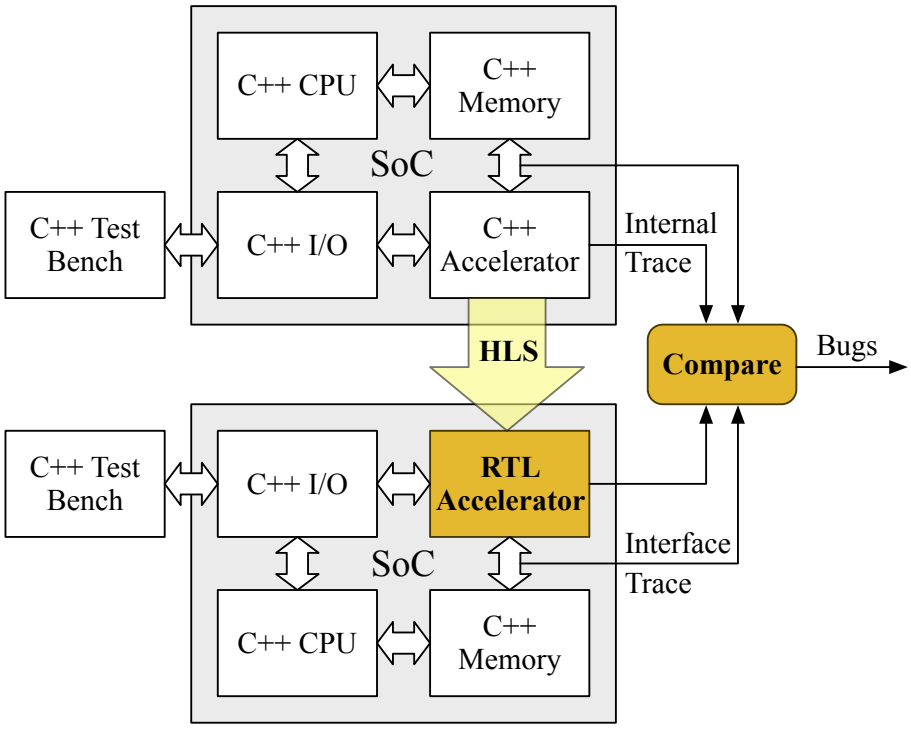


Figure 5.1: Using hybrid tracing for early pre-silicon integration testing.

Table 5.1: Methods for Catching Different Kinds of Logic Bugs

	unactivated	**masked**	**unmasked**
deterministic	coverage	unit testing	debug tools
non-deterministic	analysis	**hybrid tracing**	

5.1 Comparison to Software Debugging

As mentioned in Chapter 1, logic bugs have many ways to elude detection. Fortunately, hybrid tracing leverages HLS, which brings in a variety of software debugging tools to bear on the problem. Table 5.1 broadly classifies logic bugs by their behavior in three categories: unactivated, masked, and unmasked. Each of these bug classes can be further divided into deterministic and non-deterministic subcategories.

Unactivated bugs are caused by gaps in coverage (e.g. the buggy line of code was never executed or a condition was never met) which are best addressed by software coverage analysis tools. Such tools will point out these gaps, allowing the hardware designer to modify the design or test vectors to eliminate the coverage gaps and activate bugs that may be hiding in those gaps. A deterministic, activated bug is reliably reproducible by definition. Existing

software debugging tools are good at helping a user to isolate a deterministic bug. While software debugging tools can help with deterministic, masked bugs as well by increasing observability, software practices also encourage unit testing to help detect such masked bugs in the first place.

Software debugging techniques are much less useful for non-deterministic, activated bugs. By definition, such bugs are likely to behave differently in a software testing environment when compared to an RTL simulation environment. For example, the bug may cause a failure in RTL simulation, but the high-level simulation produces correct output, rendering software debugging techniques by themselves unhelpful. Without any aid to track this bug down, the hardware designer has little choice but to attempt to find the bug in the RTL waveform by tracing backward in execution from the observed failure to the root cause. After this painstaking process, the designer has another difficult problem to solve: determining the source-code level meaning for the buggy RTL variable he identified. This can be very non-trivial with the complex software and HLS transformations involved in translating a high-level language to RTL. Hybrid tracing is designed to address both of these difficult problems, making isolation for the most difficult bugs automated and fast.

5.2 Hybrid Tracing Framework

Our hybrid tracing implementation is illustrated in Figure 5.2. The input to the framework is a C++ module targeted for debugging and written with a synthesizable subset of C++ supported by the HLS tool. Additional non-synthesizable modules (not shown to simplify the illustration) representing the system environment such as those in Figure 5.1 can be integrated into the hardware simulation through co-simulation and into the software simulation environment through linking.

As one would expect from an H-QED variant, there are two branches of the framework, a hardware RTL-level simulation branch and a "reference" software branch. Both branches have integrated instrumentation passes to enable greater observability of internal source-level variables. In the hardware branch, the instrumentation is integrated into the HLS engine after scheduling to minimize intrusiveness. The HLS engine produces SystemVerilog as output, which is then translated to a cycle-accurate SystemC module

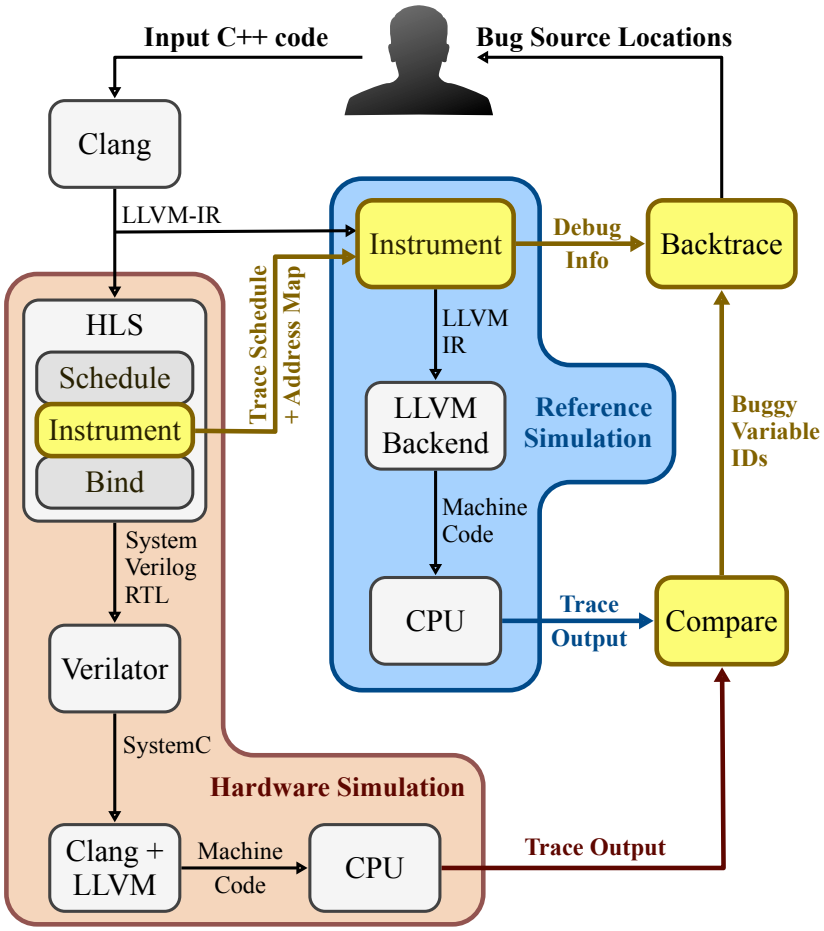


Figure 5.2: Our hybrid tracing framework.

using Verilator [55, 56]. The software branch performs software compilation using the LLVM framework [57] and contains a custom instrumentation pass designed to reproduce the trace output produced by the hardware simulation given some scheduling and address mapping information from the HLS instrumentation pass. The output of the two branches is variable trace sequences for discrepancy analysis; when mismatches are found, information on which variable(s) caused the discrepancy can be used to identify the C/C++ source code involved with the bug. We now discuss each component of our framework in detail in the following subsections.

5.2.1 Hardware Simulation

The hardware simulation is a cycle-accurate RTL simulation of the hardware module in a test environment that can include high-level implementations of modules it interfaces with. This process starts with the high-level synthesis of the LLVM Intermediate Representation (LLVM-IR) for the hardware module. We use an in-house high-level synthesis engine that is based on LegUp [58]. We insert our hardware instrumentation pass after scheduling and optimization, but before binding. The pass takes an optimized, scheduled CDFG as input and adds trace annotations on all variables that have a software counterpart.

In our previous prototype implementation in [3], the instrumentation pass was inserted pre-scheduling. The problem with this approach is that the trace calls need to be scheduled and their dependencies considered. This results in some cases in deferred scheduling of trace calls to maintain ordering (i.e. the scheduler would prefer to reorder the trace calls to match the scheduling of the operations traced, but is not allowed to) which can increase register pressure artificially, change the synthesis result, and result in multicycle error detection latencies. Furthermore, trace calls create false dependencies that can potentially block or complicate HLS optimizations. To improve the hybrid tracing implementation in [3], we split the instrumentation pass into complementary hardware and software passes as shown in Figure 5.2 and integrated the hardware instrumentation pass into our HLS engine.

Adding the trace calls pre-binding makes them mere debugging annotations on signals that the HLS engine has decided are "real" signals (i.e. not redundant operations or dead operations) that must be bound to a physical

Listing 5.1: Input C++ Code (foo.cpp)

```
1 int bar[4];
2 int foo (int x, int index) {
3     int y = bar[index];
4     bar[index] = x + y;
5     return x * y;
6 }
```

Listing 5.2: LLVM-IR (Simplified for Clarity)

```
global [4 x i32] bar

i32 foo (i32 x, i32 index) {
    i32 * addr = getelementptr(bar[index])
    i32 y = load addr
    i32 tmp1 = add x, y
    store tmp1 → addr
    i32 tmp2 = mul x, y
    ret tmp2
}
```

resource. The trace annotations simply follow their operations and variables to the physical functional units and registers that they are bound to, producing the appropriate output in the state the variable is generated. During binding, the annotations are handled separately from the binding of "real" hardware. In other words, the addition of these debugging annotations is nonintrusive as they do not affect the synthesizable binding solution generated by the HLS engine. Furthermore, the annotations can easily be removed or ignored for the purpose of synthesis.

We illustrate our hardware instrumentation pass with an example shown

Listing 5.3: Scheduled Operations (Custom IR)

```
<0x1000> global [4 x i32] bar

i32 foo (i32 x, i32 index) {
 [0]     i32 addr = add index, 0x1000
[0-1]    i32 y = load addr
 [1]     i32 tmp1 = add x, y
[1-2]    store tmp1 → addr
[1-2]    i32 tmp2 = mul x, y
 [2]     ret tmp2
}
```

Listing 5.4: Hardware Trace Operations Inserted

```
i32 foo (i32 x, i32 index) {
 ...
 [0]  trace(0, x)
 [0]  trace(1, index)
 [0]  trace(2, addr)
 [1]  trace(3, y)
 [1]  trace(4, tmp1)
 [2]  trace(5, tmp2)
 [2]  ret tmp2
}
```

Listing 5.5: Software Trace Operations Inserted

```
i32 foo (i32 x, i32 index) {
    ...
    trace(0, x)
    trace(1, index)
    trace(2, addr_convert(addr))
    trace(3, y)
    trace(4, tmp1)
    trace(5, tmp2)
    ret tmp2
}
```

Table 5.2: Hardware Address Map

Memory	**Address**	**Depth**	**Width**
bar	0x1000	4	32

Table 5.3: Trace Schedule

Func.	**Block**	**Traced Variables**
foo	entry	x:0, index:1, addr:2, y:3, tmp1:4, tmp2:5

Table 5.4: Debugging Information

id	**func:var**	**file:line:col**
0	foo:x	foo.cpp:2:14
1	foo:index	foo.cpp:2:21
2	foo:addr	foo.cpp:3:16
3	foo:y	foo.cpp:3:9
4	foo:tmp1	foo.cpp:4:20
5	foo:tmp2	foo.cpp:5:14

Table 5.5: Address Translation Table

Variable	SW addr	HW addr
bar[0]	0xa7010	0x1000
bar[1]	0xa7014	0x1001
bar[2]	0xa7018	0x1002
bar[3]	0xa701c	0x1003

in Listings 5.1–5.5 and Tables 5.2–5.5. Listing 5.1 shows an example input C program with a global variable to be mapped to a memory and a function which becomes a hardware module. Listing 5.2 shows the same program lowered to LLVM-IR and Listing 5.3 shows the scheduled, optimized hardware IR (internal scheduled CDFG and memory address space map representation) right before binding. The memory is annotated with its base address on the left, and each instruction is annotated on the left with scheduling information indicating the cycles the instruction is scheduled for execution in. An instruction result becomes available on the final cycle it executes.

Listing 5.4 shows the additional trace instructions inserted by our hardware instrumentation pass as well as their scheduled states. Table 5.2 shows the *hardware address map* determined by our HLS engine and passed to the software instrumentation pass (Section 5.2.2). Each row of the address map indicates an LLVM variable, its base hardware address and the corresponding memory block depth and width. Table 5.3 is the *trace schedule* passed from our hardware instrumentation pass to the software instrumentation pass. The trace schedule has a row for each basic block in each function and indicates the LLVM-IR variables traced in that basic block, in the order they are traced, as well as a unique integer identifier for each variable.

The challenge in creating this trace schedule is mapping hardware IR variables to LLVM-IR variables. Not all LLVM-IR variables have a corresponding hardware IR counterpart as some CDFG nodes may be optimized away by HLS transformations (e.g. a global array reference becomes a constant address in hardware IR after the HLS engine defines a static address space mapping which can lead to further constant propagation optimizations). Similarly not all hardware IR variables have an LLVM-IR counterpart. An abstract LLVM-IR operation such as the memory address computing *getelementptr* operation can involve a number of additions and multiplications, generating multiple hardware IR variables that represent intermediate computations and

do not correspond to the final *getelementptr* result.

Our solution to this problem is to propagate debugging annotations in our hardware IR. During the initial lowering of LLVM-IR to our hardware IR, we annotate hardware IR CDFG nodes with references to the corresponding LLVM-IR variables they are equivalent to. We then preserve these LLVM-IR variable references across hardware IR transforms such as scheduling and optimization where feasible. Even if a variable is lowered to a constant, we propagate the variable annotation to a constant as this enables the compile-time computation of the constant value to be checked. Our hardware instrumentation pass can then scan the hardware IR to find all nodes with debugging annotations, generate trace instructions for them, and use the annotations to produce the corresponding LLVM-IR instruction in the trace schedule.

Once the trace annotations are inserted, our HLS engine performs binding and finishes with RTL generation, during which our HLS engine lowers each "trace" instruction instance to a SystemVerilog "$fwrite" call that prints the corresponding variable ID and value to a file.[1] The hardware simulation process then proceeds with the following steps:

Verilator Send the resulting SystemVerilog RTL code through Verilator [55, 56]. Verilator translates the RTL code to an equivalent cycle-accurate SystemC representation that a standard C++ compiler can compile and run against the SystemC libraries.[2]

Clang+LLVM Using the Clang and LLVM compiler toolchain, compile the cycle-accurate SystemC version of the hardware module. The hardware module can optionally be linked with untimed high-level C/C++ versions of software modules it interfaces with some additional glue code to connect the SystemC interfaces to the untimed C/C++ function calls.

CPU Execution Run the resulting machine code on a CPU. The binary will use the CPU for untimed execution of software portions of the design, and will perform cycle-accurate RTL simulation of the hardware module. The

[1] These calls can easily be ignored or removed for the purpose of synthesis after debugging is complete.

[2] An alternative to SystemC translation is RTL/C++ co-simulation. We find that SystemC RTL simulation is faster [3].

instrumented hardware module will dump RTL execution traces to a file to be sent to the comparison step (Section 5.2.3) together with the software-only reference trace.

5.2.2 Reference Simulation

The purpose of the reference simulation is to produce a "gold" reference trace that the hardware simulation should reproduce exactly under bug-free conditions. This process starts with the LLVM-IR software instrumentation pass that inserts instrumentation that generates this trace. In order for the software instrumentation pass to generate code that matches the hardware trace, it needs two pieces of information from the hardware instrumentation pass (Section 5.2.1): a *trace schedule* and a *hardware address map*.

The trace schedule provides the ordering of the trace calls as determined by the decisions made by the HLS scheduler, enabling the software pass to generate code that produces the trace output in the same order. The hardware address map enables the software pass to generate code that reproduces hardware address values by translating software addresses to hardware addresses. The hardware instrumentation pass also generates a debugging information table that provides a source location for each LLVM-IR variable that is traced, which enables the backtracing process (Section 5.2.3) to provide source-level meaning to the hardware designer. This debugging information is generated from debugging metadata provided by the LLVM compiler infrastructure.

We continue our example to illustrate this process. Listing 5.5 shows the software trace operations the software instrumentation pass adds to the LLVM-IR in Listing 5.2 and Table 5.4 shows the outputted debugging information file. The pass adds calls to two library functions that we implement and link the LLVM module against: "trace" and "addr_convert". The trace function is semantically equivalent to the hardware variant, taking a variable ID and value and writing that variable ID and value pair to a file with an "fprintf" call. The addr_convert call is inserted for all address variables traced and takes a software address as input, translates it to a hardware address, and outputs that hardware address. This translation process is based on a static address translation table generated at runtime, which we will discuss shortly.

We then run the instrumented LLVM-IR through the LLVM backend to

generate machine code that runs on the host CPU, linking the code with our library that implements the "trace" and "addr_convert" functions. Since the trace function generates output, the software compiler cannot change the relative ordering of the trace calls, ensuring that the software trace order will match the hardware trace order under bug-free conditions.

The resulting machine code is then run on the CPU, which executes the software model of the module and generates a trace through the inserted "trace" calls. To enable the "addr_convert" calls, we first generate a translation table for all variable elements mapped to hardware memory blocks at the start of execution using the hardware address map as well as dynamic software addresses of variable elements in that address map. Table 5.5 shows the address translation table for our example, generated with the help of the hardware address map (Table 5.2). The software addresses for these variables are fixed at runtime and thus can be computed at runtime initialization because we ensure that such variables are mapped to static memory. Once the table is initialized, each software to hardware address conversion is a simple lookup in this table.[3]

5.2.3 Trace Comparison and Debugging

Once the reference simulation (Section 5.2.2) and hardware simulation (Section 5.2.1) are complete, we compare the resulting trace files for both simulations. Under bug-free conditions, the traces will be identical since we ensure that the ordering of the trace calls produced by the reference simulation matches the hardware schedule and we perform software to hardware address translation to produce hardware address values in the reference simulation. Thus any discrepancy indicates a bug. (See Section 5.4 for an example bug and how our process detects it.) For discrepancies observed, we look up the variable IDs with mismatched values in the debug info file generated by the software instrumentation pass (Section 5.2.2, Table 5.4) to identify the variable name and source locations for the mismatched variables. For the first variable ID with a discrepancy, we report the variable name, source location, and the pair of mismatched values observed to the hardware designer.

[3]To translate legitimate pointers to the "end" of an array (i.e. one past the last element, used as an upper pointer bound in a loop), we first pass the LLVM-IR module through a transformation that adds an extra dummy element to each array.

Table 5.6: Bug Detection Example

Source Code	Reference Trace	Hardware Trace

`int x;`
`if (cond) {`
`  x = 1;`
`}`
`z = x + y;`
	`cond : 0`
	`-- skip body --`
	`x : 7461`
	`y : 7`
	`x + y : 7468`
		`cond : 0`
		`-- skip body --`
		`x : -24905`
		`y : 7`
		`x + y : -24898`

5.3 Simulation Breakpoint Trigger

A variation of hybrid tracing can be used as a hardware simulation breakpoint trigger. This can be useful if a bug is only activated in the generated hardware. In this variation, the reference simulation trace must be generated first. In the hardware simulation, the trace function is then implemented as a function that reads one variable ID and value pair from the reference simulation trace and checks if it matches the variable ID and value parameters the function receives. If there is a mismatch, the function calls the Verilog "$stop" function (or similar) to suspend the simulation, put it in interactive mode, and enable the test engineer to examine the simulated hardware state right at the point the bug first activates.

5.4 Bug Example

How does hybrid tracing detect bugs? We have found that a reader's intuition often leads one to the conclusion that source code bugs cannot be caught because the same buggy code is fed to both the hardware and software simulations, and thus the two simulations will produce identical traces. While this intuition is correct for *deterministic* bugs that always behave the same way, this intuition fails for *non-deterministic* bugs that have hard-to-predict behavior that depends on many confounding factors. (The reader may want to refer back to Section 1.3 for insights about different kinds of bug behavior and to Section 5.1 for how hybrid tracing fits in with other debugging techniques.)

To drive this point home, we provide a simple example of a non-deterministic source-code bug in Table 5.6. The bug in the source code is that "x" is used

uninitialized, and thus its value is non-deterministic, i.e. it is toolchain and environment dependent. The reference simulation will likely use some garbage value from the stack, previously used for some other variable. The hardware simulation could use the register initial power-on state. It is unlikely that the hardware and software values for "x" are identical, and thus hybrid tracing pinpoints the location where the bug activates.

Note that while this bug is a simple example for explanatory purposes, initialization bugs can have much more complex activation conditions (e.g. involving large buffers that are partially initialized). More importantly, there are many other types of non-deterministic bugs that cause different hardware and software behavior such as undefined memory accesses, timing dependent bugs, and hardware-specific protocol violations. In our experiments with all of the known bugs in the CHStone high-level synthesis benchmark suite [54] in Section 5.5.3 we find that hybrid tracing is able to detect many different kinds of logic bugs, including some previously unknown bugs as well as bugs that a suite of existing static software analysis and dynamic software bug detection techniques are unable to detect.

5.5 Experimental Results

To demonstrate the effectiveness and practicality of hybrid tracing we ran a series of simulation experiments to collect data for cycle, flip-flop, and simulation overhead as well as error detection latencies and coverage estimates for logic bugs. We used all 12 benchmarks from CHStone [54] and 15 benchmarks from the PolyBench [54] benchmark suites.

5.5.1 Intrusiveness

To determine the intrusiveness of hybrid tracing, we performed HLS with and without hybrid tracing and measured the number of flip-flops and benchmark execution cycles for each benchmark. The hybrid tracing values normalized to the baseline values are shown Table 5.7. For reference we also performed the same experiment with the LLVM-level instrumentation insertion approach of [3]. We observe significant overheads for that approach as high as a 247% cycle overhead and a 2.5% flip-flop overhead.

Table 5.7: Overhead Due to Intrusiveness (flip-flops / cycles, %)

Bench	[3]	HT
adpcm	24.91 / 0.00	0 / 0
aes	246.88 / 0.02	0 / 0
atax	0.00 / 0.00	0 / 0
bicg	0.00 / 0.00	0 / 0
blowfish	29.16 / 0.00	0 / 0
dfadd	11.05 / 2.51	0 / 0
dfdiv	0.00 / 0.00	0 / 0
dfmul	5.65 / 0.00	0 / 0
dfsin	7.55 / 1.47	0 / 0
doitgen	0.00 / 0.00	0 / 0
floyd-warsh	17.87 / 0.00	0 / 0
gemm	0.00 / 0.00	0 / 0
gemver	5.80 / 0.00	0 / 0
gesummv	0.00 / 0.00	0 / 0

Bench	[3]	HT
gsm	0.31 / 0.00	0 / 0
jpeg	25.63 / 0.00	0 / 0
matrix	0.00 / 0.00	0 / 0
matrix-tiled	28.20 / 0.00	0 / 0
mips	0.00 / 0.00	0 / 0
motion	0.00 / 0.00	0 / 0
mvt	26.48 / 0.00	0 / 0
reg-detect	22.96 / 0.08	0 / 0
sha	18.28 / 0.00	0 / 0
symm	0.00 / 0.00	0 / 0
syr2k	0.00 / 0.00	0 / 0
syrk	0.00 / 0.00	0 / 0
trmm	0.00 / 0.00	0 / 0

The LLVM-level instrumentation of [3] creates significant cycle overhead because it imposes an additional constraint on the scheduler to maintain trace call order. The flip-flop overhead is the result of variable lifetime extensions needed for trace calls scheduled one or more cycles after what would normally be the end of a variable's lifetime. This delayed trace call scheduling is the result of the aforementioned scheduling constraints and causes increased register pressure, and thus may increase required register allocation. Our hybrid tracing approach, in contrast, has no intrusiveness because the instrumentation is inserted after scheduling, so the instrumentation cannot interfere with the schedule by design. Furthermore, we show that QoR is unaffected by instrumentation, and thus the instrumentation can be ignored or removed for logic synthesis input equivalent to the same design without instrumentation.

5.5.2 Simulation Time Costs

To determine the simulation performance impact of hybrid tracing, we measured the SystemC RTL simulation time of untraced RTL code for our 27 benchmarks and compared it with the combined time of software simulation and SystemC RTL simulation with tracing enabled. The results are shown

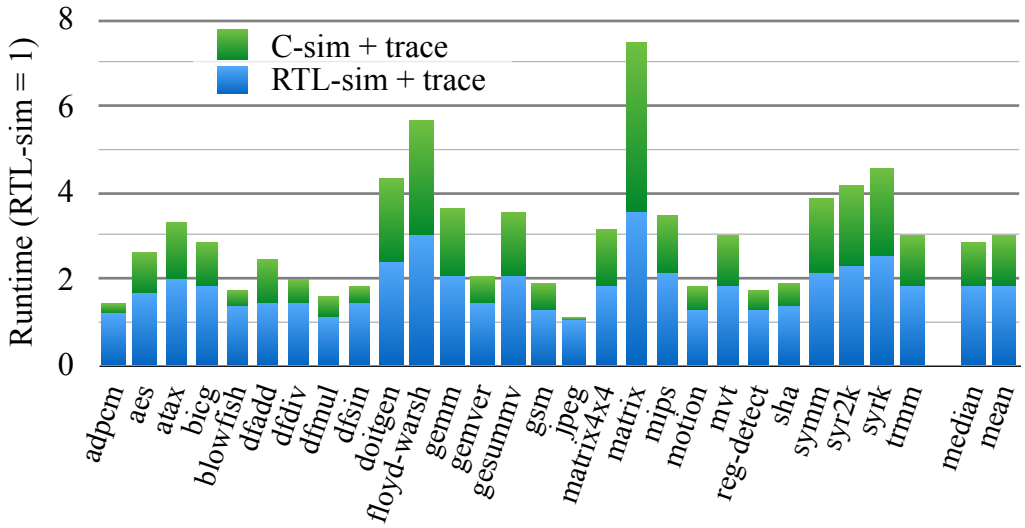


Figure 5.3: Hybrid tracing instrumented RTL simulation and C++ simulation time normalized to uninstrumented RTL simulation.

in Figure 5.3. We observe a mean RTL simulation time of 1.79× and an additional reference simulation time of 1.18× compared to untraced RTL simulation for a total mean overhead of 2.97×. Reduction of overheads may be possible by engineering faster variations of the trace functions, in particular by using binary mode I/O operations for writing and comparing traces (i.e. to avoid formatting overhead for human readability of trace files).

5.5.3 Logic Bug Effectiveness

To evaluate the effectiveness of hybrid tracing in detecting logic bugs, we considered all 21 known real bugs in the current and past versions of CHStone [54], 7 synthetic bugs injected into CHStone benchmarks, 2 bugs in previous versions of our HLS engine itself, and 3 bugs in a synthesizable C implementation of a matrix multiply kernel generated by FCUDA [59]. We attempted to detect these bugs with both hybrid tracing and hybrid hashing. Hybrid hashing is designed primarily to detect electrical bugs, but we provide logic bug results for hybrid hashing here for the sake of completeness. (We discuss hybrid hashing in detail in Chapter 6.) We also use an end result check that compares the output of the benchmark with a known correct output and several software-based static and dynamic bug detection tools as references for comparison. In hardware simulation, we initialized registers and memories

to random values, a common technique for enhancing bug detection. Results of our experiments are enumerated in Table 5.8.

Real bugs were identified by exhaustively exploring the version changes of CHStone for bug fixes. H-QED also uncovered some previously unknown bugs in the then-current version of CHStone, 1.10, prompting the release of 1.11, in which H-QED found yet another bug that was introduced. Previously unknown bugs are highlighted in bold in Table 5.8. 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. All real CHStone bugs were confirmed with the CHStone authors.

We also created synthetic bugs in some CHStone benchmarks by violating interface assumptions made about benchmark inputs (e.g. that the number of inputs is even). HLS engine bugs were reproduced by modifying our HLS engine to emulate the original buggy behavior. The FCUDA output bugs were handled in the same way as the real CHStone bugs: each bug is isolated into a bug benchmark.

The columns in Table 5.8 are as follows:

- **Benchmark: Versions(s)** indicates the CHStone benchmark and version the bug benchmark was based on. For the real CHStone bugs, the version indicated is the last version of the benchmark containing the bug.

- **Bug Patch Line(s)** indicates what line(s) of code were modified to fix the bug. Canonical bug fixes from version history are used for this column where applicable. For the synthetic bugs, this column indicates the line modified to inject the bug.

- **Bug Type** indicates the root cause classification for the bug. The type acronyms in this column are defined in Table 5.9.

- **Nondet.?** indicates whether the bug is non-deterministic, meaning that the bug behavior is not well defined by standard C [60] semantics (or only affects the hardware in the case of the two HLS engine bugs). A "C only" value means that the bug is non-deterministic, but becomes deterministic after compiler optimizations.

- **Act.?** indicates if the bug activates during benchmark execution.

Table 5.8: Evaluation of H-QED and Software Tools against Logic Bugs

	Benchmark: Version(s)	Bug Patch Line(s)	Bug Type	Non-det.?	Act.?	Clang Cov.	Nondet. Act. Line	Cpp-check	Val-grind	Clang San.	HT Line	Com. Vars	HT Lat.	HH Lat.	ER Lat.
Real CHStone Bugs	adpcm:1.8	adpcm.c:689	MLU	no	yes	yes	N/A	-	-	-	N/A	N/A	N/A	N/A	yes
	gsm:1.4	lpc.c:87	OOB	C only	yes	yes	lpc.c:88	-	88	88	-	N/A	-	-	-
		lpc.c:150	OOB	C only	yes	yes	lpc.c:151	-	151	151	-	N/A	-	-	-
		lpc.c:157	OOB	yes	yes	yes	lpc.c:158	-	158	158	158	no	1	77	-
	jpeg:1.9	decode.c:204	*++	no	no	no	N/A	204	N/A	N/A	N/A	N/A	N/A	N/A	N/A
		decode.c:205	*++	no	no	no	N/A	205	N/A	N/A	N/A	N/A	N/A	N/A	N/A
		decode.c:209	*++	no	no	no	N/A	209	N/A	N/A	N/A	N/A	N/A	N/A	N/A
		marker.c:0[a]	INIT	yes	yes	yes	marker.c:385	-	-	-	400	N/A	1	-	-
	mips:1.6	mips.c:172-173	USE	no	no	no	N/A	-	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	mips:1.9	mips.c:102	INIT	yes	yes	no	mips.c:179	-	-	-	179	yes	1	-	-
		mips.c:103	INIT	yes	yes	no	mips.c:182	-	-	-	182	yes	1	-	-
	mips:1.10	**mips.c:105**	**INIT**	yes	yes	yes	mips.c:255	-	-	-	255	yes	1	22	-
	mips:1.11	**mips.c:91**	**OOB**	yes	yes	yes	mips.c:134	134	-	134	134	yes	1	9	-
	motion:1.2	mpeg2.c:225	OOB	yes	yes	yes	mpeg2.c:226	-	226	-	226	yes	1	8	91
	motion:1.4	getbits.c:113	SHFT	C only	yes	yes	getbits.c:113	-	-	113	-	N/A	-	-	yes
		motion.c:155	SHFT	C only	yes	yes	motions.c:155	-	-	155	-	N/A	-	-	-
		motion.c:160	SHFT	yes	yes	yes	motion.c:160	-	-	160	mpeg2.c: 388	no	1	15	15
		motion.c:166	SHFT	yes	no	no	motion.c:166	-	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	motion:1.10	**getbits.c:134**	**SHFT**	yes	yes	yes	getbits.c:134	-	-	134	134	yes	0	102	-
		getbits.c:144	**SHFT**	yes	yes	yes	getbits.c:144	-	-	144	144	yes	0	-	-
		getbits.c:155	SHFT	yes	no	yes	getbits.c:155	-	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Synthetic Bugs	adpcm:1.11	adpcm.c:778	OOB	yes	yes	yes	adpcm.c:848	848	854	848	848	yes	1	46	-
	aes:1.11	aes.c:83	OOB	yes	yes	yes	aes_func.c:159	-	140[b]	159	159	yes	-7	362	903
	blowfish:1.11	bf_pi.h:77	OOB	yes	yes	yes	bf_enc.c:105	105	-	-	105	no	1	180	643
	gsm:1.11	gsm.c:32	OOB	yes	yes	yes	lpc.c:59	-	59[c]	59	59	no	1	179	999
	jpeg:1.11	huffman.c:86	ZERO	no	yes	yes	N/A	-	-	-	N/A	N/A	N/A	N/A	yes
	motion:1.11	mpeg2.c:354	INIT	yes	yes	yes	motion.c:68	mpeg2.c: 377	68	68[b]	68	yes	1	10	168
	sha:1.11	sha.h:52	OOB	yes	yes	yes	sha.c:84	-	-	84	84	no	1	11	857
HLS Bugs	jpeg:1.4	HLS Engine	INIT	yes	yes	yes	jfif_read.c:69	N/A	N/A	N/A	69	yes	1	122	hang
	jpeg:1.11	HLS Engine	ZERO	yes	yes	yes	huffman.c:118	N/A	N/A	N/A	118	yes	1	70	811k
	matrix-mul: FCUDA generated	mm.c:52, 54	BUF	no	yes	yes	N/A	-	-	-	N/A	N/A	N/A	N/A	yes
		mm.c:157, 161	INF	no	yes	yes	N/A	157, 161	-	-	N/A	N/A	N/A	N/A	hang
		mm.c:165-172	OOB	yes	yes	yes	memcpy.h:5	-	-	5	5	no	-1	30	292k

[a] Bug is the absence of a header file "#include" directive.
[b] Execution was terminated with a segmentation fault reported at this line.
[c] add.c:68 is at the top of the reported trace, followed by lpc.c:59.

Table 5.9: Bug Types

Type	Description
MLU	Manual loop unrolling omits one iteration
OOB	Out-of-bounds array access
*++	Wrongly assuming dereference (*) has higher precedence than postincrement (++)
INIT	Read of uninitialized variable
USE	Unintended sign extension
SHFT	Bit shift by out-of-bounds amount
ZERO	Variable initialized to zero instead of nonzero initializer
BUF	Copying from the wrong half of a split buffer
INF	Infinite loop due to erroneous loop termination condition

- **Clang Cov.** The Clang coverage analysis tool [61] provides source-based dynamic code coverage through code instrumentation that tracks the execution count for AST nodes. We used the coverage tool in Clang 3.9 with the "Source-based Code Coverage" reference methodology to determine if code lines relevant to each bug are executed. The "Clang cov." column indicates if the non-deterministic activation line is covered. If the bug is deterministic, this column indicates if the bug patch line(s) are covered.

- **Nondet. Activation Line** indicates the line of code, determined by inspection, where non-deterministic behavior first occurs (e.g. uninitialized variable access, out-of-bounds memory access, etc.).

- **Cppcheck** [62] is a static analysis tool that detects many kinds of C/C++ bugs through a battery of static checking heuristics. We ran Cppcheck 1.76.1 with additional "warning" checks enabled. We report the first line flagged as a warning or an error by this tool.[4] An entry of "-" means that a technique failed to detect the bug.

- **Valgrind** [63] is a dynamic binary instrumentation framework with instrumentation modes that can detect memory errors as well as stack

[4] We ignored warnings in the "gsm" benchmark about a function parameter assignment having no effect outside of the function as inspection of the code showed this kind of coding style to be intentional. Cppcheck also flags this in the current version (free of all known bugs) of gsm at lpc.c:308. We also ignored warnings in the FCUDA benchmarks about an uninitialized struct member as we determined through exhaustive search that this member was never accessed.

and global array overruns. We ran two passes of Valgrind 3.11.0 using the reference methodology provided in the "Valgrind Quick Start Guide" using GCC 4.9.2 as the source compiler.[5] The first pass invoked the "Memcheck" memory error detection tool and the second invoked the "SGCheck" experimental stack and global array overrun detector tool. When Valgrind detects an error, it will indicate the line where the error occurred with a function call stack below. We report the first line indicated, except where otherwise noted. Dynamic techniques are unable to detect unactivated bugs, so we indicate "N/A" for those cases.

- **Clang San.** Clang also comes with a set of code "sanitizer" tools which add instrumentation to the code at compile time to detect errors. The instrumentation then performs checks at runtime to identify various kinds of memory access and undefined behavior problems. We ran three passes of Clang with the three relevant sanitizer tools: the AddressSanitizer which detects memory errors, the MemorySanitizer which detects uninitialized reads, and the UndefinedBehaviorSanitizer which detects undefined behavior. When the sanitizer instrumentation identifies a problem, it dumps a stack trace similar to Valgrind; we report the first line indicated.

- **HT Line** indicates the buggy line as reported by our hybrid tracing framework. Hybrid tracing is also not applicable for deterministic bugs, so we indicate "N/A" for those cases.

- **Com. Vars** indicates if there are variables in common between the HT Line and the Bug Patch Line(s), indicating a strong hint for a potential bug fix.

- **HT Lat., HH Lat., ER Lat.** indicate the error detection latency for respectively hybrid tracing, hybrid hashing, and the end result check measured in cycles from non-deterministic bug activation to detection. For deterministic bugs, hybrid tracing and hybrid hashing are not applicable, but the end result check still is, although there is no well-defined non-deterministic bug activation cycle for measuring

[5] We found that debugging information generated by GCC was compatible with Valgrind, while debugging information generated by Clang was not, hence the decision to use GCC with Valgrind.

error detection latency, so we indicate with a "yes" entry if the ERC is able to detect such bugs. An ERC entry of "hang" means that the benchmark execution fails to terminate.

We also evaluated our bug benchmarks with the Clang static analyzer [64], which is another source-code analysis tool for finding bugs in C, C++, and Objective-C programs. We ran the built-in static analyzer in Clang 3.9 using the "scan-build" wrapper tool. The Clang static analyzer failed to identify any of our bugs.[6]

We make the following observations in these results:

1. Of the 21 real CHStone bugs, 16 are non-deterministic, providing evidence that the "difficult" bugs that escape into releases/production tend to be non-deterministic.

2. Six of these bugs are not activated, and the Clang coverage analysis tool detects 5 of those cases.[7]

3. No tool dominates the others in bug detection and each has unique strengths. Using a combination of tools is the best way to detect/localize bugs.

4. Compiler optimizations can complicate bug detection by making non-deterministic bugs deterministic (e.g. by statically evaluating undefined behavior and eliminating it).

5. In most cases, the different tools agree on the bug location although in a few cases compiler optimizations complicate bug localization by

[6] The static analyzer did report a number of warnings about values stored in a variable that were never read. While this is arguably poor coding style, manual inspection of each flagged line finds no evidence of the dead store being intended to have some effect. Furthermore, the current version (free of all known bugs) of each benchmark generates the same warnings.

The static analyzer also complained about several uninitialized variables being read in the FCUDA matrix multiply benchmark. These same warnings also showed up in the reference (free of all known bugs) version. We found these claims to be vague and difficult to investigate by code inspection and thus resorted to empirical methods. We were able to eliminate these warnings by initializing several variables at the point of allocation. We then attempted to verify the claims by initializing those same variables with random data and observing if any failure occurs (incorrect output, or otherwise). No failure was observed with random initialization.

[7] At getbits.c:155 a variable range condition that is never met is required for bug activation.

making it difficult to map instructions back to source code locations, resulting in some localization accuracy loss.

6. In 12 out of 19 cases where hybrid tracing reported a buggy line, the line had at least one variable in common with the bug patch line (or *was* the patch line), indicating a strong hint for a fix.

7. Hybrid tracing error detection latency is 1 cycle or less, average hybrid hashing error detection latency is 83 cycles (all bug benchmarks have a signature output interval $n = 100$), and end result check latency can be thousands of cycles.

8. We observe *negative* hybrid tracing error detection latency for two OOB bugs, meaning that hybrid tracing detects activation conditions the reported number of cycles *before* bug activation. In both of these cases, the hardware version of the benchmark computed an out-of-bounds address one or more cycles before issuing a load for that address. In the software version, the corresponding address overflowed beyond the translation table for the variable it was intended to point to, resulting in the software-to-hardware translation producing a mismatch before the undefined memory access even occurred.

CHAPTER 6

POST-SILICON VALIDATION: HYBRID HASHING

As mentioned in Chapter 4, area and bandwidth costs are the primary constraints when inserting instrumentation for post-silicon validation of accelerators. We call our post-silicon variation of H-QED *hybrid hashing* since we reduce the traces of variable values to a running hash value that is used to generate a low-bandwidth trickle of "signature" bits. The primary goal of hybrid hashing is to detect electrical bugs. Hybrid hashing can also detect most of the logic bugs that hybrid tracing can, but we expect hybrid tracing to catch most (if not all) non-deterministic logic bugs pre-silicon. While both of our pre-silicon and post-silicon H-QED solutions can be integrated into an SoC design, we pay special attention to integration post-silicon because of the limited testing flexibility of physical hardware compared to pre-silicon testing.

Figure 6.1 shows an SoC-level view of our hybrid hashing enabled accelerators. The SoC typically consists of processor core(s), accelerator(s) (with hybrid hashing instrumentation in our case), and uncore components. The inputs and outputs of the accelerators are supplied by the processor cores inside the SoC. During PSV, the accelerators generate hardware signatures that are saved in dedicated on-chip memories (Figure 6.1a). These signatures are then later compared to a reference set of signatures to detect bugs using a similar software instrumentation pass as hybrid tracing, but modified to reproduce the running hashing function and signature generation functionality of the hardware. Section 6.1 discusses the hybrid hashing process in detail, and Section 6.3 details a proposed methodology for integrating hybrid hashing into a post-silicon validation run.

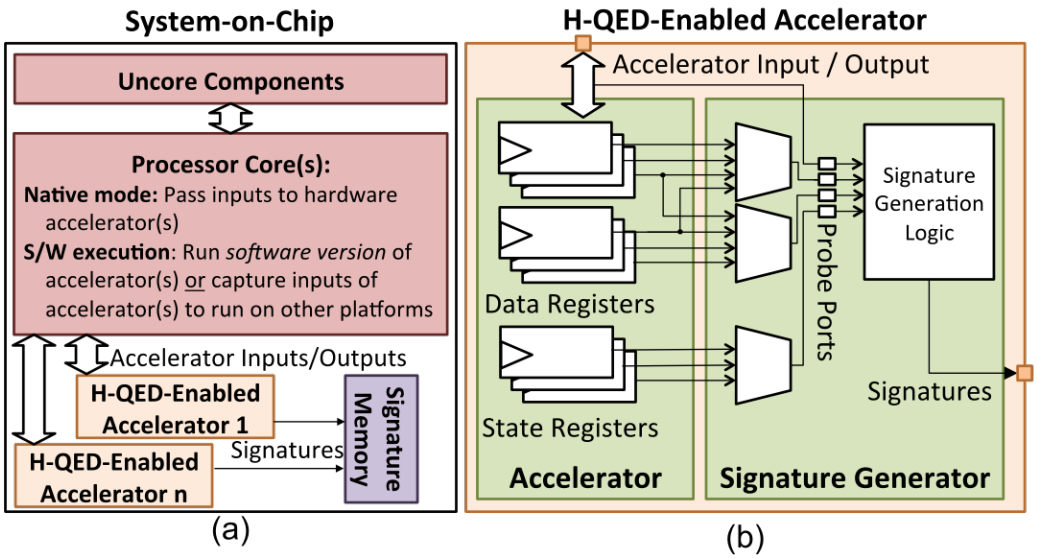


Figure 6.1: Hybrid hashing instrumented accelerators inside an SoC. (a) SoC-level view, and (b) block diagram of an instrumented accelerator showing the accelerator and the signature generator.

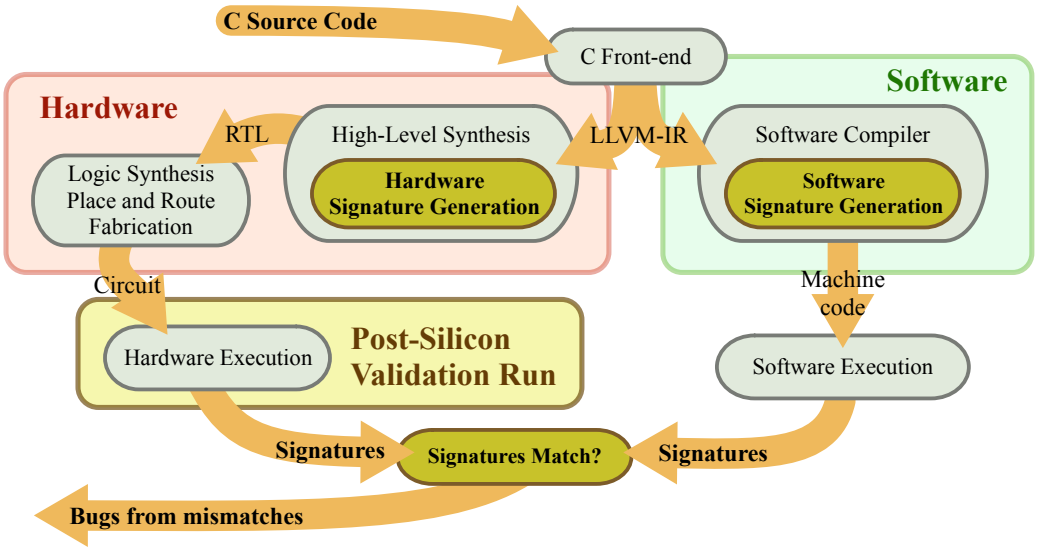


Figure 6.2: Our hybrid hashing framework.

6.1 Hybrid Hashing Framework

Our hybrid hashing implementation is illustrated in Figure 6.2. The input to the framework is a high-level design of a hardware accelerator. Again, as one would expect from an H-QED variant, there are two branches of the framework, a hardware branch and a software branch. Like the hybrid tracing process, the hardware branch has an instrumentation pass integrated into the HLS engine after scheduling. The software branch also involves a complementary instrumentation pass that takes as input a *probe schedule* and a *hardware address map* from our hardware instrumentation pass to ensure that it will produce the same signature stream as the hardware under bug-free conditions. We use the term "probe" instead of "trace" in the context of post-silicon validation to avoid confusing our high-level hybrid hashing technique with the many trace-buffer based approaches that perform cycle-granularity recording of RTL-level signals. (See Section 3.1.2 for a discussion contrasting hybrid hashing with trace-buffer techniques.)

Unlike hybrid tracing, our hybrid hashing framework is area cost and bandwidth constrained, and thus adds some twists to reduce these costs. Our cost reduction strategies are fourfold:

1. Reduce the initial raw signal probing bandwidth by only tracing key "non-temporary" variables.

2. Use a hybrid multiplexor and XOR tree reduction logic to drastically reduce the number of probe bits to a small number with minimum area cost.

3. Use an LFSR to compute a running hash of this reduced signature.

4. Output a single bit checksum computed from the LFSR state every n cycles (configurable).

As Figure 6.1b shows, our hybrid hashing framework produces an RTL implementation with integrated hashing instrumentation. This instrumentation generates a sequence of signature bits during a PSV run. Care must be taken to ensure that the instrumentation does 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,

we store hardware signatures in a dedicated on-chip memory with dedicated communication channels, as shown in Figure 6.1a.[1] The costs associated with this storage are reported in Section 9.2.

As mentioned earlier, the primary goal of hybrid hashing is to detect electrical bugs as hybrid tracing will detect most logic bugs while electrical bugs are likely to escape pre-silicon validation due to the difficulty involved in accurately modeling and predicting the electrical level behavior of a complex design. The main problem with electrical bug modeling is that accurate modeling is too slow to be useful. Indeed even the relatively fast RTL-level simulation process, which does not model electrical bugs, can have simulation speeds that are several orders of magnitude slower than real-time for complex designs. With limited modeling capabilities to detect electrical bugs pre-silicon, post-silicon validation becomes the last line of defense against electrical bugs escaping to end-user deployment.

While we find that hybrid tracing of hardware signals that are equivalent to LLVM-IR variables is sufficient to detect any non-deterministic logic bug that activates in pre-silicon validation with essentially no error detection latency (see Section 5.5.3 for a demonstration of this with our bug benchmarks), electrical bugs activated in post-silicon validation can affect almost any hardware structure in the hardware accelerator, including the state register. To ensure that electrical bugs are caught quickly and do not make it past the outputs of the accelerator undetected, we add additional instrumentation to the state register as well as the accelerator's input and output ports. The intuition here is that if we check all accelerator outputs and periodically check all "non-temporary" bits of the accelerator's state, then electrical bugs will have no place to hide.

We now discuss these additions in detail.

6.1.1 Hardware Execution

The hardware execution is an in-situ test of the fabricated accelerator with embedded hybrid hashing instrumentation through an existing post-silicon validation testing harness. Similar to the hybrid tracing process, we start

[1] It may be possible to minimize signature storage costs (while controlling intrusiveness) by streaming hardware signatures to off-chip memory using existing debugging ports, such as JTAG.

with an LLVM-IR implementation for the hardware accelerator and perform instrumentation after scheduling and optimization, but before binding on an internal hardware IR representation. To reduce the initial raw probing bandwidth, we only probe variables that are *non-temporary*. Looking at the states in the FSM that each variable is live, we define a non-temporary variable as one that crosses more than one state transition, at least one of which is a basic block boundary (i.e. the variable is live in more than one basic block).

Our scheduler prefers to schedule each probe for a variable in its last use state (last state where it is accessed). The intuition here is to observe the variable at the last cycle in its lifetime to catch all potential electrical-bug induced value mutations that could have occurred in earlier cycles before the value goes into a functional unit where the mutation could be masked. Note that this contrasts with hybrid tracing instrumentation which targets logic bugs by observing variable values at the start of their lifetime, right after the value is generated by some operation, since logic bugs causing variable value mutations are unlikely. Another goal, however, is to minimize the number of probe ports carrying these probe signals coming out of the accelerator through multiplexing. To allocate a minimum number of register probe ports, we use an algorithm that attempts to create a feasible probe schedule using a single register probe. We attempt to reschedule probes for variables with the same use state to predecessor states (where the variable is still live) to produce a feasible schedule. If scheduling fails, we attempt to schedule again with an additional probe port and repeat until scheduling succeeds. This algorithm is similar to our algorithm for scheduling shadow datapath checkpoints in Chapter 7 (Algorithm 1 on page 74).

After instrumentation, the resulting hardware IR is run through the binding process to produce a set of shared probe ports for the accelerator's CDFG variables. During the RTL generation process of our HLS engine, appropriate multiplexors are produced for those probe ports. In addition, we add dedicated probe ports for each accelerator input and output and state machine. As discussed earlier, these additional ports enable electrical bug detection. Each probe port outputs a probed value in the states the signal it is probing is live (i.e. driven to a value that is accessed), zero otherwise to avoid contaminating the generated signatures with garbage values.

Our HLS engine generates additional RTL for signature generation logic.

Listing 6.1: Scheduled Operations (custom IR)

```
<0x1000> global [100 x i32] Z
<0x2000> global [100 x i32] B

void bar(i32 z_ptr, i32 b_ptr) {
[0-1]    i32 z = load z_ptr
 [1]     i32 a = add x, y
[1-2]    i32 b = mul a, z
[2-3]    store b → b_ptr
}
```

As shown in Figure 6.3, signature generation involves a bitwise XOR reduction of the probe port signals to reduce the number of bits to a small number with minimum area cost.[2] This XOR reducer output is then fed to an LFSR, which computes a running hash of the reducer output, ensuring that all probed signal history is captured in this hash, including the cycle timing of those signals. Periodically, every n cycles (configurable), we output a one-bit signature from the LFSR.

Assuming that the LFSR has a sufficient number of state bits for the probability of aliasing inside the LFSR to be negligible, we can compute the expected time from an error being captured (meaning becoming different from the error-free value) in the LFSR state to being captured in a signature bit as follows: After an error is captured in the LFSR state, the average time until the next signature bit is outputted is $n/2$. The probability of aliasing in each signature bit is $1/2$ and each alias occurrence costs n cycles of latency. Thus the expected cycle delay from LFSR error capture to the first signature bit error capture is

$$E[\text{sig EDL}] = \frac{1}{2}n + \sum_{i=0}^{\infty} \frac{i}{2^{i+1}}n = \frac{3}{2}n \tag{6.1}$$

Note that the delay distribution decays exponentially, so delays several times this average are unlikely.

[2]The probe port multiplexors and XOR reducers can be pipelined as needed with some number p of additional pipeline registers to meet timing, resulting in p cycles of additional real-time error detection latency which will add a p cycle delay to a real-time error trigger. If the signature generation logic is run p cycles behind as well to match the change in the XOR reducer output timing, this delay will not affect the signatures generated. Thus offline comparison with reference signatures will have the same result, resulting in an *effective* error detection latency overhead of zero.

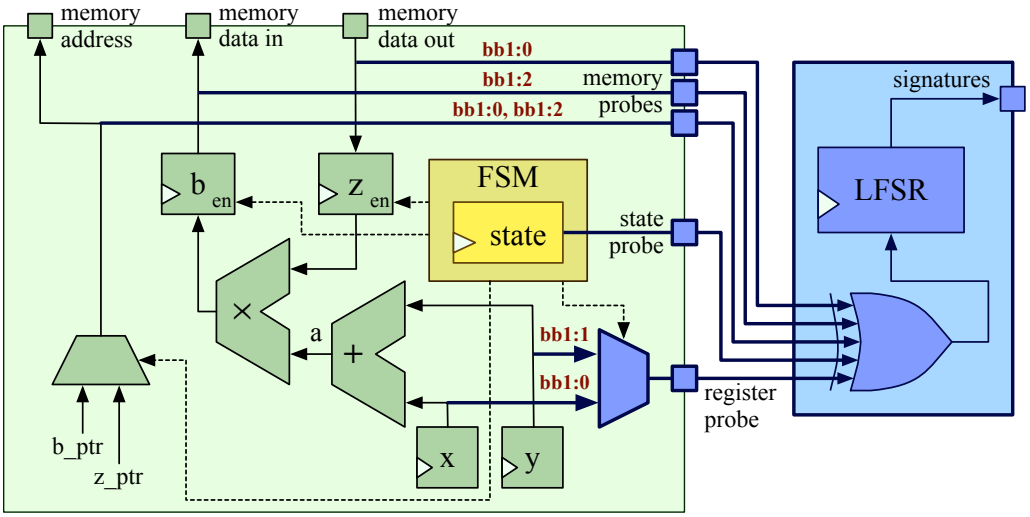


Figure 6.3: Example hybrid hashing instrumented accelerator with hardware signature generation. Probe wires are labeled in red with the basic block and cycle(s) in which they are probed.

Listing 6.2: Software Trace Operations Added

```
void bar(i32 z_ptr, i32 b_ptr) {
    ...
    software_lfsr(1 ⊕ addr_convert(z_ptr) ⊕ z ⊕ x)
    software_lfsr(2 ⊕ y)
    software_lfsr(3 ⊕ addr_convert(b_ptr) ⊕ b)
}
```

Table 6.1: Probe Schedule

Func.	Block	Cycle	Const	Probed vars
bar	bb1	0	1	z_ptr, z, x
		1	2	y
		2	3	b_ptr, b

We illustrate our hardware instrumentation and PSV execution with an example in Figure 6.3, Listings 6.1–6.2, and Table 6.1. Listing 6.1 provides an example scheduled hardware IR similar to Listing 5.3 for hybrid tracing. Table 6.1 is the probe schedule. Note that with one probe port allocated, both "x" and "y" could not be scheduled in their last use (accessed) cycle, cycle 1. Thus we rescheduled the probe of "x" for cycle 0. ("z_ptr," "z," "b_ptr," and "b" are probed through dedicated memory port probes.)

The probe schedule is similar to the pre-silicon trace schedule in Table 5.3 on page 37, but there are a number of differences. One is that each basic block in the schedule is broken into cycles which correspond to a state in the FSM that controls the accelerator. This level of granularity is needed because the downstream LFSR is sensitive to the cycle the probe values are provided. Another difference is that there is a fixed constant provided in addition to the probed variables. This fixed constant is a lumped XOR sum of all of the values that are fixed in that cycle; in this example, the only constant is the FSM's state encoding for that cycle. Finally, no variable IDs are tracked as variable-value associations are lost in the hashing of all of the probed values.

Figure 6.3 shows the resulting hardware generated by our HLS engine. Each probe port has a multiplexer associated with it that drives the port to logic 0 when it is not probed. The select signals of the multiplexer are derived from the corresponding states annotated in Figure 6.3.

6.1.2 Reference Simulation

As with hybrid tracing, the purpose of reference simulation is to reproduce the signatures produced by the instrumented hardware under bug-free conditions. This process is similar to the hybrid tracing variation, with some changes to the software instrumentation pass.

As with hybrid tracing, the software instrumentation pass takes a hardware address map and probe schedule as input. Instead of reproducing a trace, the software instrumentation pass is now tasked with reproducing the hardware's signature sequence. In order to do this, the instrumentation pass must implement the XOR reduction and LFSR in software. We design our hardware to be software implementation friendly, so the XOR reduction is simply a software XOR of all of the variables and the constant for a particular state

while the LFSR is a small series of bit shifts and XOR operations. The software instrumentation for our example is shown in Listing 6.2. The LFSR function also mimics exactly the signature output interval of the hardware LFSR, enabling the software to generate signatures that match the hardware.

6.2 Binding to Minimize Area

Efficient operator and data register sharing are crucial for minimizing hybrid hashing 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 multiplexers. 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.

6.3 Integration into PSV Testing

As mentioned at the start of this chapter, the limited testing flexibility of physical hardware compared to the pre-silicon testing of simulated hardware necessitates a careful consideration of how an accelerator will be tested as an integral part of an SoC during a PSV run. To demonstrate the practicality of our approach, we describe a proposed testing procedure as follows.

During PSV, a sequence of hardware signatures is generated and stored in on-chip memory. The signatures are then collected at the end of the PSV run. Note that during the PSV run, the hardware accelerator (and the overall SoC) operates in its native mode. Bugs inside the accelerator are thus expected to be activated during the PSV run. Next, the software version is executed on a processor; strategies to provide the same inputs to the software version as the

hardware accelerator are discussed later in this section. The software version generates a sequence of software signatures during its execution. Bugs may or may not be activated during the execution of the software version. Hence, the execution of the software version can be totally decoupled from the PSV run. For example, the user may choose to execute the software version on a different hardware platform vs. the PSV run. The sequence of hardware signatures obtained from the PSV run is compared with the sequence of software signatures obtained from the execution of the software version; any mismatch indicates bug detection. Since the execution of the software version and the subsequent signature comparisons are totally decoupled from the PSV run, we minimize possible intrusiveness introduced by hybrid hashing. 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.

6.4 Real-time Error Detection

As we discussed in Section 5.3, hybrid tracing can be used as a simulation breakpoint trigger. In a similar vein, hybrid hashing can be used as a

trigger to stop hardware execution during a post-silicon validation run when a bug is detected. This variation of hybrid hashing involves first generating the reference signature sequence for an accelerator and storing it in the accelerator's on-chip signature memory. Instead of writing to the signature memory, the signature generation logic is configured to read from the signature memory and perform a realtime signature comparison of the reference and generated signatures. If a mismatch is found, the signature comparator asserts a stop trigger, which stops all hardware execution on the SoC and enables the validation engineer to examine the chip's state (e.g. by reading out scan chains). Trace buffers can also be used in conjunction with such a trigger to provide information about past state (up to and including bug activation if the error detection latency does not exceed trace buffer capacity).

6.5 Experimental Results

To demonstrate the effectiveness and practicality of hybrid hashing we ran a series of simulation and FPGA-based emulation experiments to collect data for area and clock period overheads as well as error detection latencies and coverage estimates for electrical bugs. We used all 12 benchmarks from CHStone [54] and 15 benchmarks from the PolyBench [54] benchmark suites.

We used a 16-bit LFSR and outputted a single bit hash of the LFSR state at a regular interval. We fixed the signature output interval of each benchmark at 100 cycles or the interval that would result in a 5% signature storage area cost, whichever interval is larger. At the end of benchmark execution, we dump the full contents of both the hardware and software LFSRs into the signature stream to ensure that any late LFSR mismatches are detected.

6.5.1 Area and Delay Costs

To determine the area and delay costs of adding hybrid hashing instrumentation to an accelerator, we performed HLS with and without hybrid hashing. We then performed logic synthesis using Synopsys Design Compiler 2013-12.sp1, mapping to a 45 nm ARM standard cell library, and targeting maximum clock frequency. The area and clock period overheads for each accelerator core are shown in Figure 6.4. Results show a mean accelerator-level

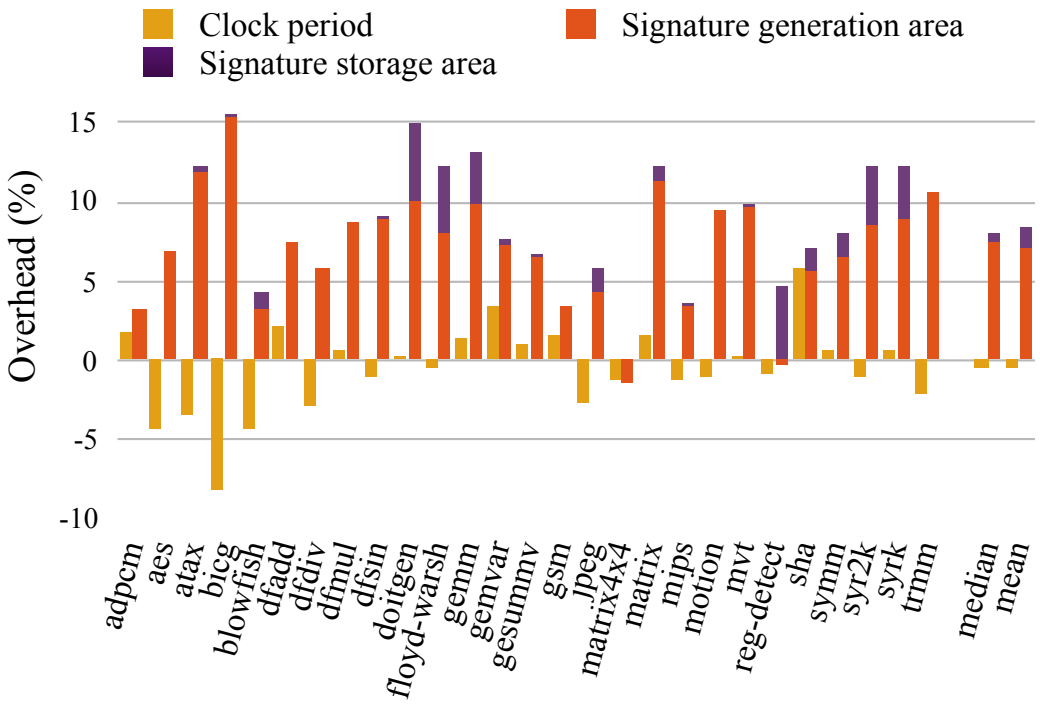


Figure 6.4: Hybrid hashing area and performance overheads.

area cost of 8.3%. We observe no clock period overhead on average.

6.5.2 Electrical Bug Effectiveness

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 hybrid hashing, 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 [65]: 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

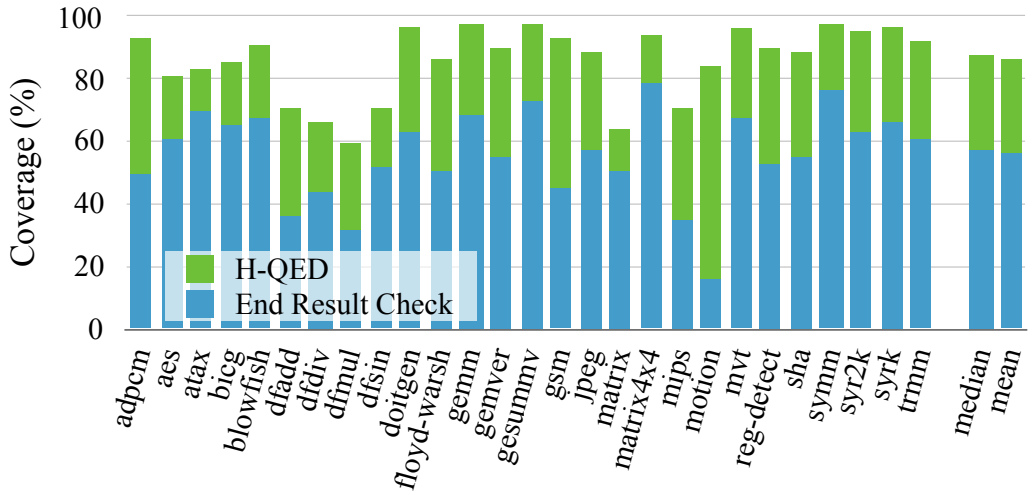


Figure 6.5: Timing error detection coverage.

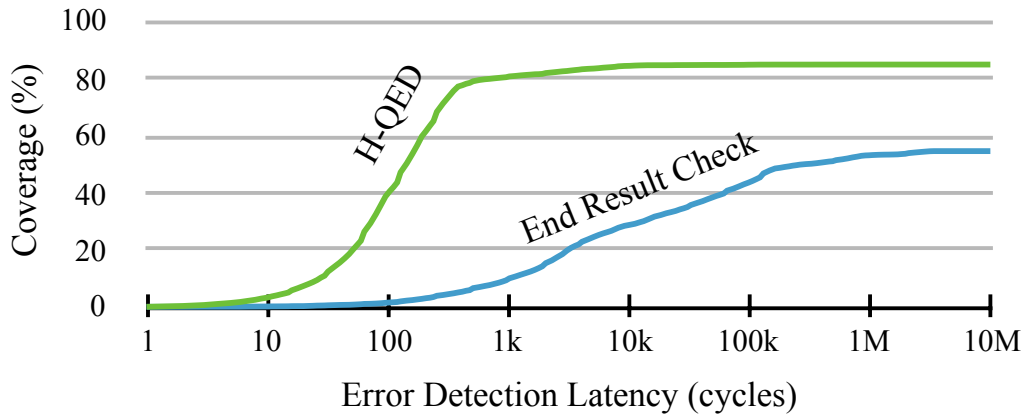


Figure 6.6: Overall timing error coverage as a function of error detection latency.

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 (the number of errors detected divided by the number of errors injected) is presented in Figure 6.5, including both masked (errors

that do not propagate to accelerator outputs so they are invisible externally) and unmasked errors (errors that propagate to the primary outputs and affect accelerator results). Note that the unmasked timing error detection coverage is 100% with hybrid hashing (i.e., we detect all unmasked errors). The overall error detection latency distribution is shown in Figure 6.6. We observed mean timing error detection coverage for hybrid hashing of 85.8% compared to 55.8% for the end result check, resulting in a 3.1× improvement (i.e., reduction) in undetected timing errors. We also observed a mean error detection latency of 705 cycles for hybrid hashing, compared to 124,490 cycles for end result check, resulting in a 176× improvement (i.e., reduction) in error detection latency.

CHAPTER 7

POST-DEPLOYMENT RESILIENCE: MODULO-3 SHADOW DATAPATHS

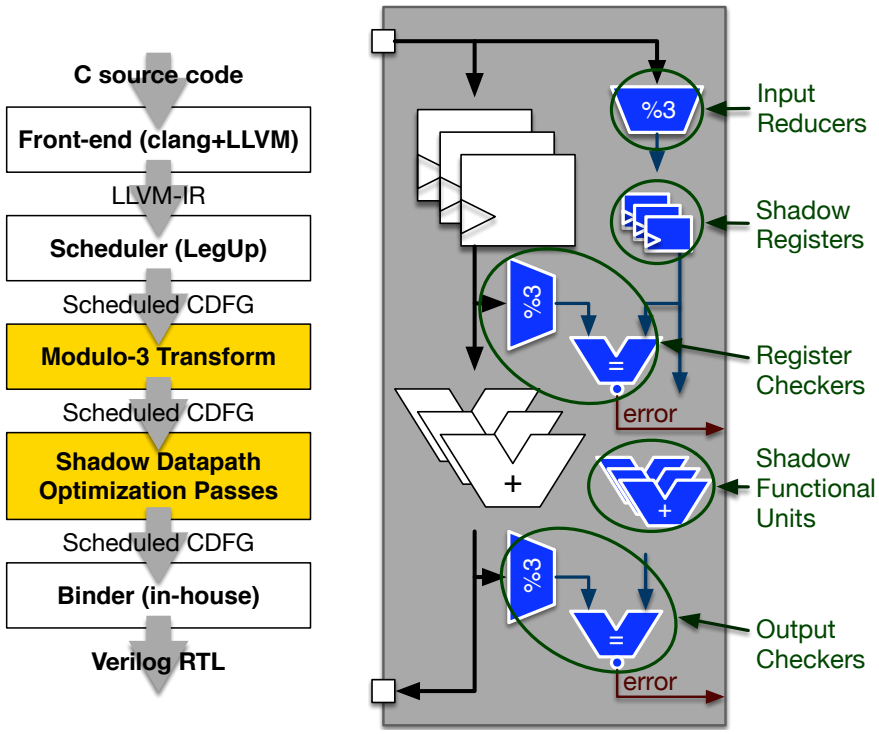
In this chapter, we propose creating a redundant, but smaller "shadow" datapath based on modulo arithmetic to detect reliability problems in an HLS design's main datapath. We automate the creation of this "shadow" datapath through a series of modulo-3 shadow datapath HLS transformations. Our main innovations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

- Intelligent scheduling of intermediate register consistency checks for maximum coverage with minimum checker allocation
- Support for mixed arithmetic/non-arithmetic data paths
- A register-duplication based checkpointing technique to demonstrate the error correction potential of our approach
- An FPGA accelerated, fully automated error injection framework using a gate-netlist transformation to enable accelerated injection for three fault models
- Error detection latencies three orders of magnitude faster than an end result check
- 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 7.1 explains the method we use to perform our error detection and correction transformations and Section 7.2 discusses our experimental setup and results.

7.1 Framework

Our approach to protecting a hardware design is a series of modulo-3 shadow datapath HLS transformations. An overview of how these transformations



(a) HLS Overview (b) Modulo-3 Transform

Figure 7.1: 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.

fit into the HLS process is illustrated in Figure 7.1a. We use the LegUp HLS scheduling engine [58] to schedule the original datapath, and perform binding with our in-house binding engine. Our transformations involve some additional scheduling steps (see Section 7.1.2). We perform our error detection transformations after scheduling but before binding to ensure that the latency of the hardware function does not increase.

Figure 7.1b 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

Table 7.1: 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 7.2: 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

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.

7.1.1 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*, meaning that there are no constraints on the output given a **U** input.

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 7.1 for the mod-3 adder, there are nine fixed output cases and seven *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

Table 7.3: 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

solution through an exhaustive search of all 4^7 possible *don't care* assignments (i.e. to check for better solutions involving compound gates). Table 7.2 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 45 nm 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 7.3 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 ×1 have no area cost since they lower to the identity function and ×0 lowers to the constant zero for multiplication. As discussed in Section 7.1.2, 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 [48]. An example reducer for $n = 16$ is

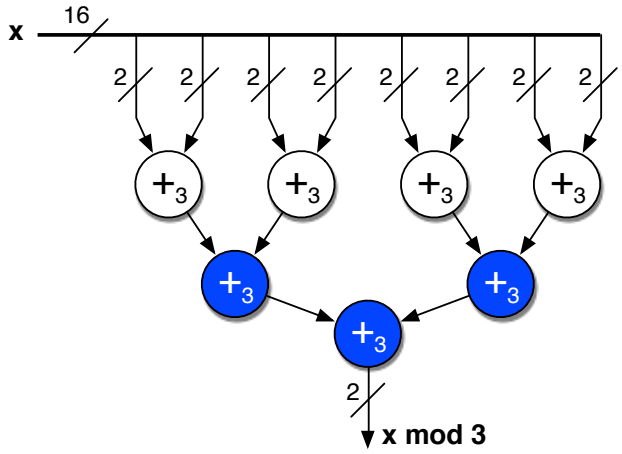


Figure 7.2: Optimized mod-3 reducer topology for a 16-bit unsigned reducer. Optimized mod-3 adders are colored blue.

Table 7.4: Optimization Results for 32-bit Mod-3 Reducer

Reducer Type	[48]		**ours**	
	area	delay	area	delay
Unsigned	263	0.62	203	0.46
Signed	267	0.66	207	0.51

illustrated in Figure 7.2. 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.

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 [48].

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 7.4 shows the small cost of this extra half-adder.

7.1.2 High-Level Synthesis Transformations

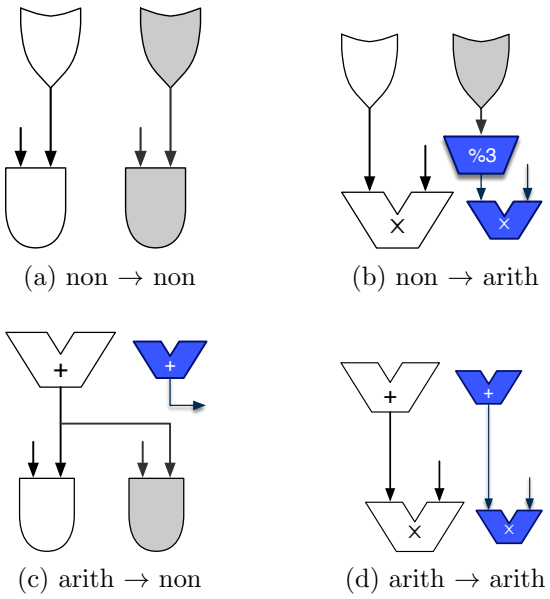
Our HLS transformations, as illustrated in Figure 7.1 on page 67, consist of a core mod-3 transform that generates the shadow datapath, as well as some dataflow-level optimization passes on the generated mod-3 logic. Our transformations operate on a scheduled control/data flow graph.

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 Table 7.5 on page 79).

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 7.3. Connections between two duplicate components and between two mod-3 components are straightforward: just make connections corresponding



(a) non → non (b) non → arith

(c) arith → non (d) arith → arith

Figure 7.3: 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 gray. Arithmetic components ("arith") are mod-3 shadowed with the shadows in blue. The unit labeled "%3" is a mod-3 reducer.

to those in the original datapath (Figures 7.3a and 7.3d). We can connect a duplicate component output (full bit width) to a mod-3 component input (2 bit) through a mod-3 reducer (Figure 7.3b). 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 7.3c).

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 7.3c 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 ensure 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 2.2.2). 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 2.2.2) 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 7.2 and 7.4). 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 7.1b). 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 scheduling succeeds.

Pipelining for Deferred Shadow Datapath Scheduling

While our mod-3 shadow functional units have low latency (Tables 7.2 and 7.3), our mod-3 reducers have high latency (Table 7.4). In addition, the insertion

[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
```

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 1× transistors are replaced with 4× and 8× transistors to meet timing requirements. Ideally, we want all of the mod-3 components to be mapped to 1× 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 7.1a 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 $\times 6$ 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 $\times 8$ and a $\times 11$ multiplier both result in the generation of a $\times 2$ mod-3 component. If both multipliers are connected to the same input, the second $\times 2$ mod-3 component is redundant and can be removed.

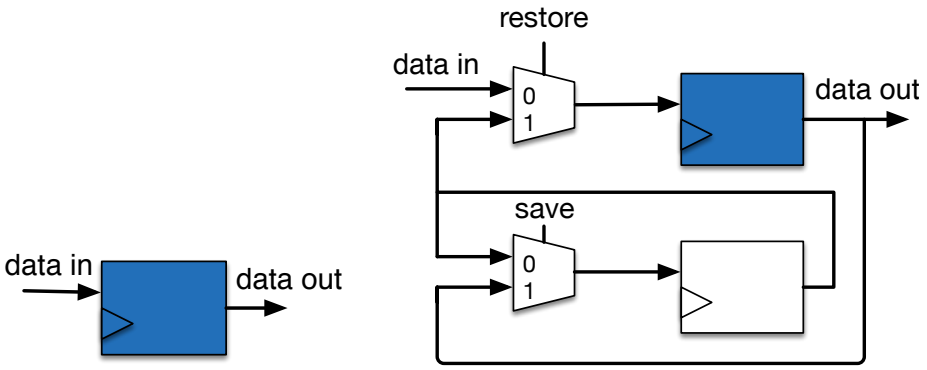
Diverse Binding

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.

7.1.3 Recovery

To enable error recovery for soft errors, we use a checkpoint and recovery register transformation, illustrated in Figure 7.4. 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.

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



(a) Original flop (b) Transformed flop

Figure 7.4: Flip-flop transformation for soft error recovery.

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 (7.1), 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 (7.2). In this equation, E is the error removal rate; M is the *error masking rate* (defined as the 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 (7.3), 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 (7.4)). 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

(7.5)).

$$P_{\mathrm{CC}} = \sum_{l} P_l \frac{\min(l, \mathrm{CI})}{\mathrm{CI}} \leq \frac{l_{\mathrm{avg}}}{\mathrm{CI}} \tag{7.1}$$

$$\mathrm{E} = \mathrm{M} + \mathrm{U}(1 - P_{\mathrm{CC}}) \tag{7.2}$$

$$\mathrm{ED} = P_{error} \times \mathrm{det.} \tag{7.3}$$

$$\text{Avg. Rollback} = \sum_{r=1}^{\mathrm{CI}} \frac{r}{\mathrm{CI}} = \frac{\mathrm{CI}+1}{2} \tag{7.4}$$

$$\text{Cycle Overhead} = \mathrm{ED} \times \text{Avg. Rollback} \tag{7.5}$$

7.2 Results and Analysis

7.2.1 Setup

Our experimental setup is illustrated in Figure 7.5. We performed logic synthesis with Synopsys Design Compiler 2013-12.sp1 with an ARM 45 nm 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 transformation 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 [65]. Then we pass these flop-cycle pairs as a subset of transient errors to our error injection enabling netlist transformation.

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

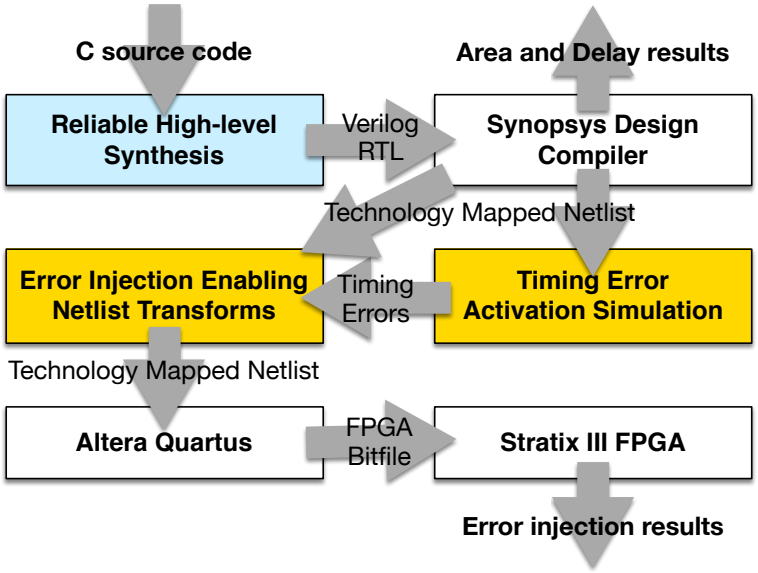


Figure 7.5: Our error detection coverage evaluation framework. Our "reliability-centric" high-level synthesis process is elaborated in Figure 7.1a. Our customized steps are highlighted in yellow.

are activated for one cycle.

7.2.2 Results

We used benchmarks from the PolyBench/C 3.2 benchmark suite [66] 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 4×4" is a tiled version of the matrix multiply benchmark that completely unrolls 4×4 tiles to explore performance/area tradeoff. We synthesized our benchmarks using our method (Section 7.1.2) and used our experimental setup (Section 7.2.1).

To determine the area cost of our error detection approach, we compare 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 7.5 shows the area and clock period overhead for both the detection logic and estimated overhead (through characterization of the hardware in Figure 7.4) for the total logic which includes both detection and recovery. We observe on average an area cost

Table 7.5: Area and Clock Period Overhead Results

Benchmark	Baseline		Detection		Total	
	area (μm^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

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 4 × 4" 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 7.6.

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 [42].

It is difficult to make a direct comparison with previous HLS work since high-level synthesis benchmarks with experimental error injection and area cost are quite limited. For reference, Concurrent Error Detection [40] 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 7.6 shows the estimated soft error removal rate and rollback cycle

Table 7.6: 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 4×4	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

overhead for our error recovery method with checkpoint intervals ranging from 10 to 100 k cycles calculated through Equations (7.1)-(7.5).

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 four parallel lines represent

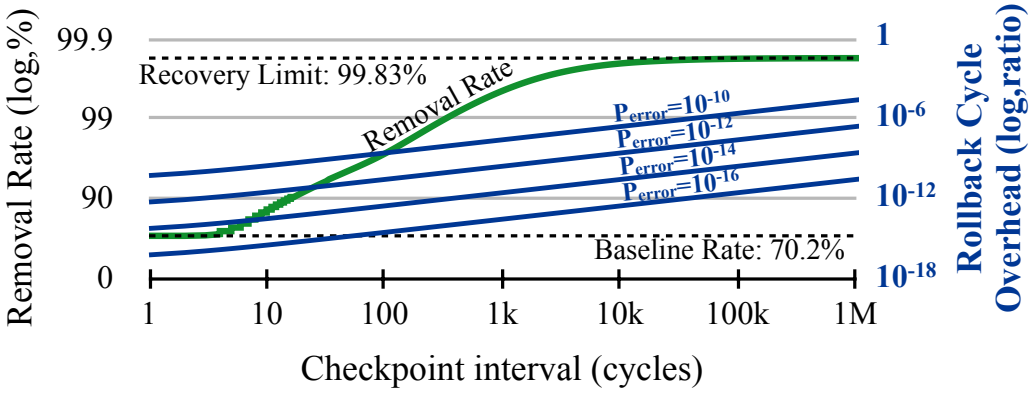


Figure 7.6: Error removal rate and rollback cycle overhead.

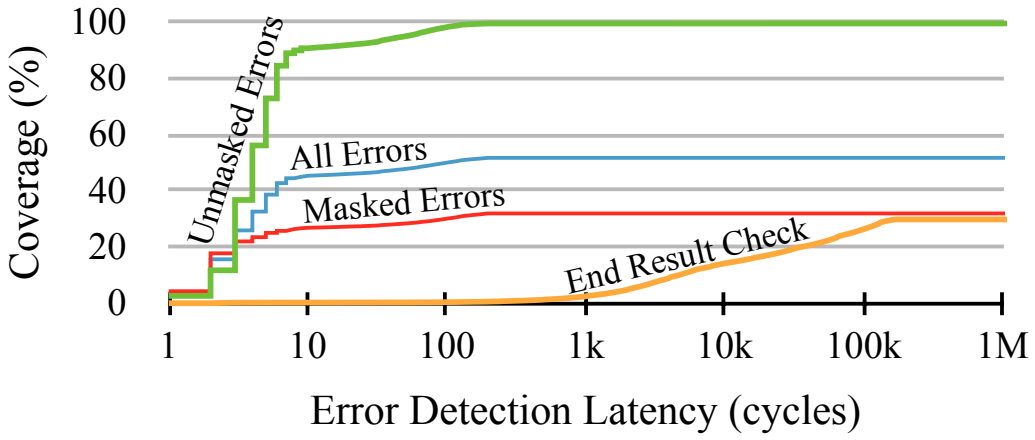


Figure 7.7: Soft error detection latency distribution.

rollback cycle overheads for different soft error rates. For reference, [67] reports a worst-case error rate of around 10^{-16} errors / cycle for a space environment assuming a clock frequency of 1 GHz.

What is interesting to observe is the tradeoff between the error removal rate and rollback cycle overhead. Larger checkpoint intervals reduce the 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 7.7 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 2 k cycles for unmasked, masked, both and ERC respectively, for an error detection latency improvement of 4150× over the ERC.

CHAPTER 8

CHEAPER MODULO FUNCTIONAL UNITS

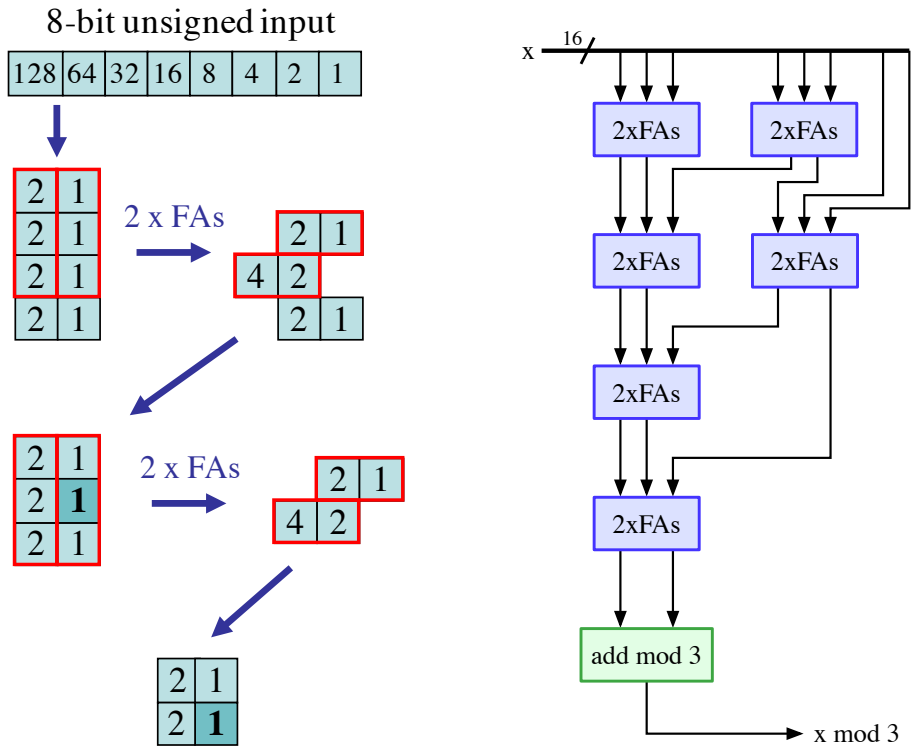
While the area cost of our modulo shadow datapaths in Chapter 7 is much better than traditional modular redundancy approaches, we want to maximize the applicability of our approach. To this end, we take a dive into gate-level architectural design for modulo arithmetic functional units. In this chapter, we create new cost-effective gate-level designs for Mersenne ($M(n)$) modulo functional units (see Section 2.2.4 for definition), and show that we cannot only decrease the cost for modulo-3 shadow datapaths, but also enable practical scaling to larger shadow datapath widths. To demonstrate the applicability of our approach for reliability, we use these building blocks to create a self-checking multiply-accumulate datapath.

8.1 Modulo Functional Units Architecture

The following subsections discuss our gate-level architectures for our modulo $M(n)$ integer reducers, adders, multipliers, negators, and zero comparator functional units. All of these functional units work with *non-normalized* n-bit encodings (see Section 2.2.6 for definition) for residues modulo $M(n)$. Furthermore, we provide illustrated examples with specific values of n for explanatory purposes, but these architectures generalize in a straightforward manner (except where noted otherwise) to any $n \geq 2$ and any input bitwidth $w \geq 2n$.

8.1.1 Reducer

Our reducer functional units compute $y = a \bmod M(n)$, where a is a w-bit wide datapath value, and y is an n-bit residue value.



(a) 8-bit mod-3 reduction strategy

(b) 16-bit mod-3 reducer

Figure 8.1: Wallace-tree like reduction strategy and 16 bit modulo-3 reducer. In (a), each square represents a bit, and the number in the square is the weight of that bit. In (b), each "2 × FA" box represents a pair of full adders, one taking three bits of weight 1 as input and one taking three bits of weight 2. Each wire (except the top input bundle) bundles two bits of weights 1 and 2.

Reduction Strategy

To perform this reduction to a residue, our unique approach is a Wallace-tree like reduction strategy shown in Figure 8.1a. Our reducer starts with a standard bit sequence representing an integer with bits having weights with successive powers of two. Using the standard homomorphism from integer arithmetic to modulo arithmetic (see Section 2.2.1), we can reduce the weights

Algorithm 2 Partial Modulo Reduction

```
procedure REDUCE(A)                          ▷ A is a m × n matrix of bits
    while |A| ≥ 3 do                         ▷ |A| is the number of rows in A
        G ← row triplets selected from A.
        L ← leftover rows, |L| < 3.
        G′ ← G passed through n × FA blocks.
        A ← {G′, L}.
    end while
    return A                                 ▷ The result is a 2 × n matrix of bits
end procedure
```

in Equation (2.7) to residues as follows:

$$y = \left(\sum_{i=0}^{w-1} 2^i a_i \right) \bmod M(n) \tag{8.1}$$

$$= \left(\sum_{i=0}^{w-1} (2^i \bmod M(n)) a_n \right) \bmod M(n) \tag{8.2}$$

$$= \left(\sum_{i=0}^{w-1} (2^{i \bmod n}) a_n \right) \bmod M(n) \tag{8.3}$$

where the last equivalence follows from Equation (2.6).

In other words, the weights on the input bits shown in Figure 8.1a are reduced to a repeating cycle of successive powers of two, drawn as a 4×2 matrix in Figure 8.1a for $n = 2$ and $M(n) = 3$. We now feed these bits to full adder (FA) gates. A full adder takes three bits of weight w as input and produces two bits as output: one of weight w and another of weight $2w$. A FA is a transistor-level optimized cell in a standard cell library that reduces the number of bits by 1 (3 inputs less 2 outputs), and as we will see shortly, performs arithmetic amenable to a modulo context. Thus FAs are ideal technology mapping targets for cost-effective modulo arithmetic.

In the 4×2 matrix in Figure 8.1a, we can select two groups of 3 bits with the same weight (highlighted in red) and pass them through full adders. The result is two bits of weight 2, one bit of weight 1, and one bit of weight 4. But $4 = 1 \pmod 3$ (an example of Equation (2.6)), so the output of the FAs is equivalent to two bits of weight 1 as well as 2. Since we also have two bits left over from the input, we now have a 3×2 matrix of bits (lower left corner of Figure 8.1a). We repeat this process, selecting groups of 3 bits and

putting them through FAs until no groups of 3 bits remain. This process is formalized in Algorithm 2.

Intuitively, it is desirable to perform reductions with entirely full adders since each FA gate is doing useful work reducing the number of bits by 1. Half adders (HA) take two bits of the same weight, w, as input and produce two bits of weights w and $2w$ and thus do not by themselves reduce the number of bits.

Architecture

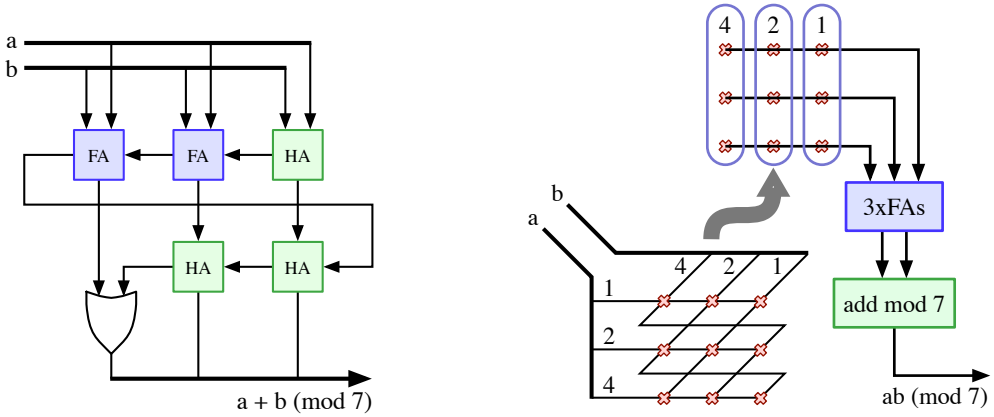
Figure 8.1b provides a block diagram for our Mersenne modulo reducer gate-level architecture for $n = 2 \implies M(n) = 3$ and input width $w = 16$. Each wire (except the top input bundle) corresponds to a bundle for a bit matrix row in Figure 8.1a if it were to be expanded from an 8-bit input to a 16-bit input. Three wires representing three rows are connected to corresponding bundles of FAs (the "$2 \times$ FA" blocks) which generate two rows (wires) of output. Note that, perhaps counterintuitively, the two output wires of each "$2\times$FA" block in Figure 8.1b represent bundles with all of the different possible bit weights (i.e. a row of a bit matrix in Figure 8.1a), **not** a bundle of sum bits and a bundle of carry bits.

Using the reduction strategy in Algorithm 2, we iteratively process all of the groups of three wires from the previous stage in parallel by passing them through "$2 \times$ FA" blocks until only two wires remain. Note that while Figure 8.1b provides an example for $n = 2 \implies M(n) = 3$ and input width $w = 16$, along with our other functional units in this section, this example generalizes to any $n \geq 2$ and any $w \geq 2n$.[1] For the final reduction stage, we use a binary modulo adder, which we discuss next.

8.1.2 Adder

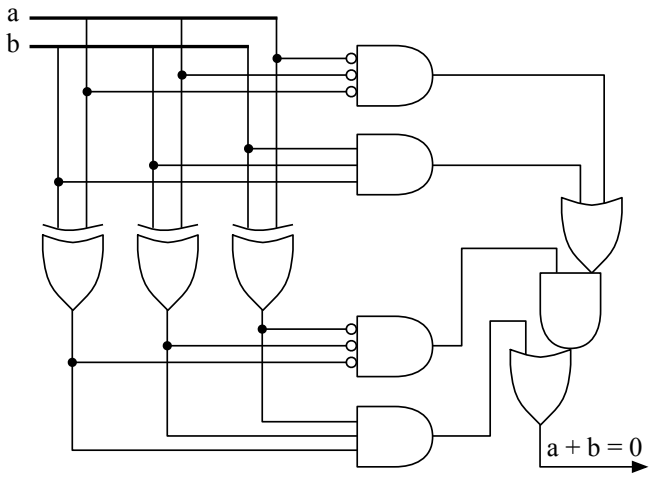
Our gate-level modulo binary adder architecture is shown in Figure 8.2a for $n = 3 \implies M = 7$. The first stage is a standard ripple-carry adder.

[1]For w values that are not a multiple of n, we can pad the input with conceptual constant zero bits until $w \bmod n = 0$. Each of those zero bits will be connected to a different full-adder, so we can recover from most of the padding overhead by optimizing those full adders with one constant zero bit to half adders.



(a) adder

(b) multiplier



(c) zero comparator

Figure 8.2: Modulo-7 adder, multiplier, and zero comparator. In (b) bits are annotated with their weights. Each X represents a 2-input AND gate with inputs connect on the left, and outputs connected on the right.

The final carry produced by the first stage has weight $2^n = 1 \pmod{M}$ by Equation (2.6), so it wraps around as a carry-in to the second stage. We guarantee under all possible adder input combinations that this carry circulation will stop before or at the most significant bit in the second stage. In other words, at most one of the input bits to the MSB adder gate in the second stage is a 1. We call an adder gate with this input constraint a *quarter adder* (QA) and implement it with a 2-input OR gate.

Theorem 1. *At most one of the inputs to the quarter adder gate in our modulo binary adder is* 1.

Proof. We prove this guarantee by contradiction. Suppose both inputs to the QA are 1. Then both inputs to each half adder in the second stage must be 1. Then all inputs to each full adder in the first stage must be 1. Then both outputs of the half adder in the first stage must be 1, which is impossible. □

8.1.3 Multiplier

Refer to Figure 8.2b for our modulo binary multiplier architecture for $n = 3 \implies M = 7$. The multiplier is like an array multiplier with a twist: each combination of input bits is combined with a 2-input AND gate, but bit weights wrap around modulo M, resulting in a $n \times n$ matrix of partial product bits as shown in the upper part of Figure 8.2b, which also corresponds to the lower-left corner. For example, the product of the two bits of weight $2^2 = 4$ will have weight $2^4 = 16 = 2^1 = 2 \pmod 7$ by Equation (2.6).

Since the output is a square matrix of bits, we can then apply our reduction techniques from Section 8.1.1 to reduce them. Figure 8.2b elaborates on the reduction for a 3×3 matrix of 9 bits.

8.1.4 Negation and Subtraction

Negation with our encodings of Mersenne modulo numbers is quite simple: just pass each bit through a NOT gate. Mathematically, this works because

$$-a = M - a = (2^n - 1) - \sum_{i=0}^{n-1} 2^i a_i \tag{8.4}$$

$$= \sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} 2^i a_i = \sum_{i=0}^{n-1} 2^i (1 - a_i) \pmod M \tag{8.5}$$

These NOT gates can be integrated into gates in upstream or downstream functional units to effectively eliminate their overhead (e.g., flip-flops with inverted outputs or NAND gates instead of AND gates in a multiplier array). Subtraction is implemented as a composition of negation and addition, i.e. $a - b = a + (-b)$.

8.1.5 Zero Comparator

We created a custom architecture for a zero comparator which takes $2n$ bits as input and compares the sum of the $2 \times n$ matrix of bits with zero, illustrated in Figure 8.2c for $n = 3 \implies M = 7$. We take $2n$ bits as input due to the extra cost of reducing $2n$ bits to n bits (see Section 8.2.2).

Theorem 2. *Our zero comparator architecture illustrated in Figure 8.2c is correct.*

Proof. We start with some special cases: the inputs $(-0 + -0)$ (all ones) and $(0 + 0)$ (all zeroes) produce the correct output by inspection. For the remaining cases the only way to get a sum of zero is if a and b are bitwise complements of each other. Again, we see by inspection that the logic will output a 1 for this case. If a and b are not bitwise complements, the only way for the logic to output a 1 is if $a = b = \pm 0$, the special cases we already discussed. □

8.2 Quality of Results Comparisons

To evaluate the area and delay of our approach, we implemented our gate-level designs with a 45 nm ARM standard cell library. Our focus is on minimum area to minimize cost, so we selected the smallest ($1\times$) standard cell for each gate type for the modulo functional units. The longer delay of a modulo shadow datapath simply increases error detection latency by a few cycles, so this tradeoff is acceptable in return for reduced area cost. We compare with other techniques compiled with the logic synthesis tool Synopsys Design Compiler 2016.03-SP5-5 and also map modulo functional units from those designs to $1\times$ standard cells to enable meaningful comparisons.

8.2.1 Modulo Functional Units

Our first set of comparisons looks at the functional-unit level and compares our designs for modulo adders, subtractors, reducers, and multipliers to equivalent designs from Chapter 7. We implement a subtractor with a negation of one input followed by an adder. In Chapter 7, the adder, subtractor, and multiplier

Table 8.1: Functional-Unit Level Results

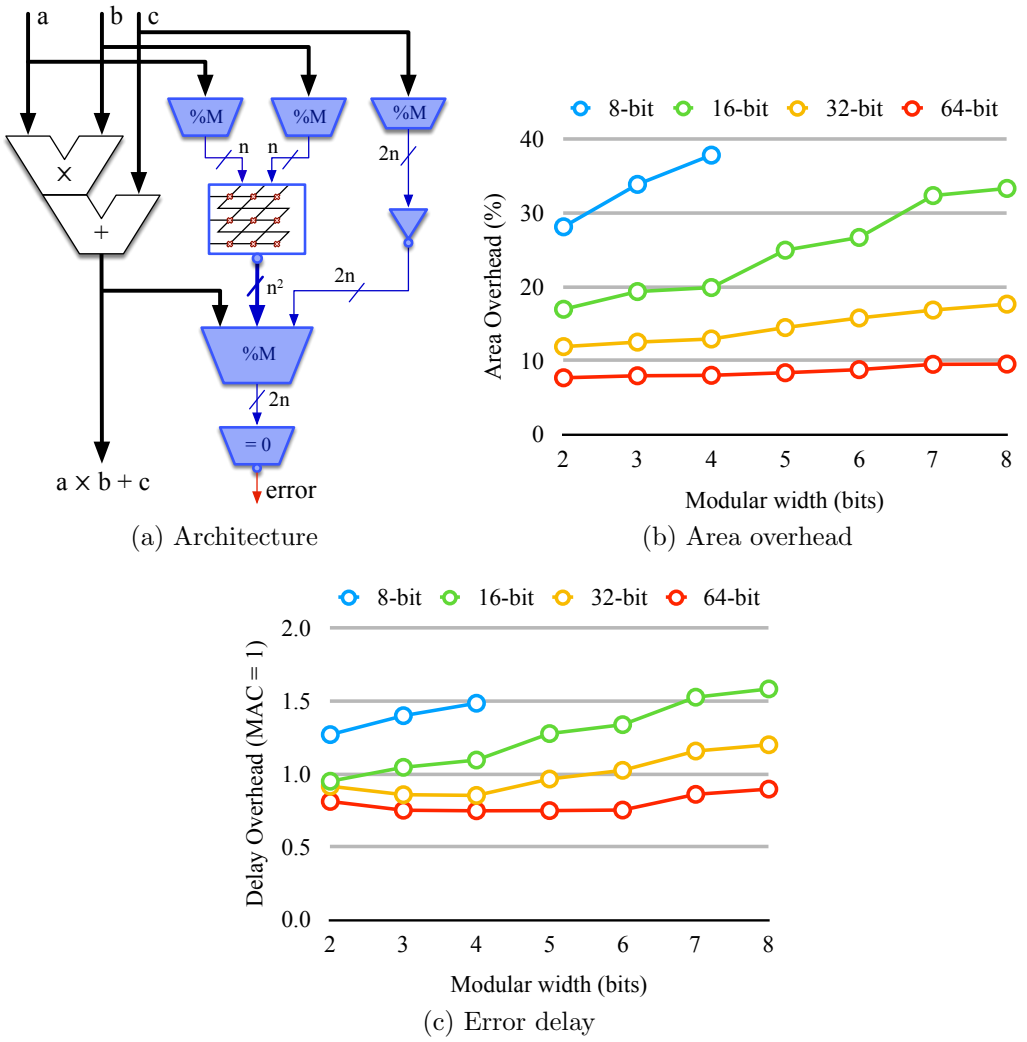
Functional Unit		Chapter 7		New Design		Difference	
		Area	Delay	Area	Delay	Area	Delay
mod-3	adder	8.3	0.09	12.8	0.13	53.7%	42.9%
	subtractor	8.3	0.09	14.0	0.15	68.9%	60.4%
	multiplier	4.5	0.04	17.8	0.16	299.3%	292.7%
	32-bit reducer	177.8	0.73	155.6	0.39	−12.5%	−47.1%
mod-7	adder	55.9	0.32	21.1	0.21	−62.3%	−35.8%
	subtractor	59.7	0.33	23.0	0.22	−61.6%	−32.7%
	multiplier	30.0	0.21	47.8	0.30	59.2%	42.3%
	32-bit reducer	493.2	1.27	153.6	0.61	−68.8%	−52.0%
mod-15	adder	188.0	0.46	29.3	0.27	−84.4%	−41.6%
	subtractor	192.8	0.53	31.9	0.29	−83.5%	−46.0%
	multiplier	133.4	0.51	90.4	0.42	−32.2%	−16.8%
	32-bit reducer	687.6	1.55	151.7	0.53	−77.9%	−66.1%

are implemented with lookup tables, while a reducer is implemented as a tree of modulo adders.

Table 8.1 shows the results of our comparisons. Area is measured in μm^2 while delay is measured in ns. We observe that our reducer designs, which tend to be the dominant part of shadow datapath costs, provide lower area and delay than those of Chapter 7. Even for the simplest modulo-3 reducer, we achieve a 12.5% reduction in area and a 47.1% reduction in delay. Furthermore, this reducer cost is essentially fixed as the modulo base scales because the number of full adders required is the same as the number of bits reduced $(w - n)$. Longer delays also increase the need for pipeline flip-flops which in turn impacts area cost. Our other observation is that as we scale to larger Mersenne bases, even the adders and subtractors and eventually the multiplier become less costly than Chapter 7. This is expected due to the exponential scaling nature of lookup tables in Chapter 7.

8.2.2 Self-Checking Multiply Accumulator

In this section, we demonstrate the applicability of our approach to a self-checking multiply-accumulator (MAC) illustrated in Figure 8.3a. The shadow datapath is built from components introduced in Section 8.1: a full reducer to n bits (Figure 8.1b), a partial reducer to $2n$ bits (omitting the final binary adder in Figure 8.1b), a modulo multiplier matrix from Figure 8.2b (with



(a) Architecture (b) Area overhead

(c) Error delay

Figure 8.3: Self-checking multiply accumulator architecture and overhead evaluation. $M = 2^n - 1$.

NAND gates to negate the output), a negation inverter (Section 8.1.4), and a zero comparator (Figure 8.2c). Note that the reducers are summation reducers, so they function as adders.

Under error free conditions, the shadow datapath will compute $-(a \bmod M)(b \bmod M)$ and $-(c \bmod M)$, add it to $ab + c$ from the output, reduce the result modulo M, and get a result of 0. Computation errors in either the main or shadow datapath will generate a nonzero result (provided aliasing does not occur, which in our experience is unlikely for single bit errors).

A key strategy in this design is the avoidance of reduction beyond $2n$ bits (except for multiplier inputs) as reduction beyond $2n$ bits involves the use

of half adders which do not directly provide bit reduction while the main reduction process is mapped entirely to full adders. This strategy is similar to the carry save technique used in standard binary integer arithmetic design.

We evaluated our MAC architecture by synthesizing the multiply accumulate main datapath with Design Compiler targeting minimum delay while generating 1× gate-level designs for the shadow datapath with gate-level architectural templates and Algorithm 2. QoR results for different width datapaths and modulo widths (n) are shown in Figures 8.3b and 8.3c. We observe 12–18% area overhead for a 32-bit self-checking MAC. We observe error signal delays of about 2× the delay of the main datapath. As mentioned at the start of this section, the longer delay of a modulo shadow datapath simply increases error detection latency by a few cycles, and does not affect the performance of the main datapath, so this tradeoff is acceptable in return for reduced area cost.[2]

[2]For example, in Chapter 7, we used a pipelining strategy to run the shadow datapath two cycles behind the main datapath without affecting performance.

CHAPTER 9

CROSS-LAYER RESILIENCE SYNERGIES

In this chapter, we take shadow datapaths further by looking for cross-layer synergies with other techniques for increasing reliability against soft errors. We consider five different existing reliability improvement techniques: algorithm based fault tolerance (ABFT) [46,47], error detection by duplicated instructions (EDDI) [68], modulo-3 shadow datapaths (Chapter 7), parity checkers at the logic synthesis level [38], and hardened flip-flop standard cells [36,37]. ABFT, parity, and flip-flop hardening are the techniques that demonstrated benefit in the CLEAR study [38]. EDDI protects all instructions and variables through full, fine-grained duplication, and thus provides a benchmark for maximum coverage through algorithm- or instruction-level transformation. As discussed in Chapter 7, modulo-3 shadow datapaths is a high-level synthesis technique that has demonstrated a 175× reliability improvement at a low cost. We also consider useful combinations of these techniques with a systematic feed-forward approach: applying higher-level techniques first and then using lower-level techniques to fill in any reliability gaps that remain.

To systematically evaluate these representative techniques and their combinations, we developed an automated framework, shown in Figure 9.1, involving high-level synthesis and full place-and-route physical design with our five reliability transformations applied in their respective layers. We used an error injection enabling netlist transformation and FPGA emulation to perform a grand total of over 400,000 emulated flip-flop error injections across our 12 accelerator designs, one injection per accelerator execution. We also evaluated runtime and energy overhead through the use of commercial simulation and power estimation tools.

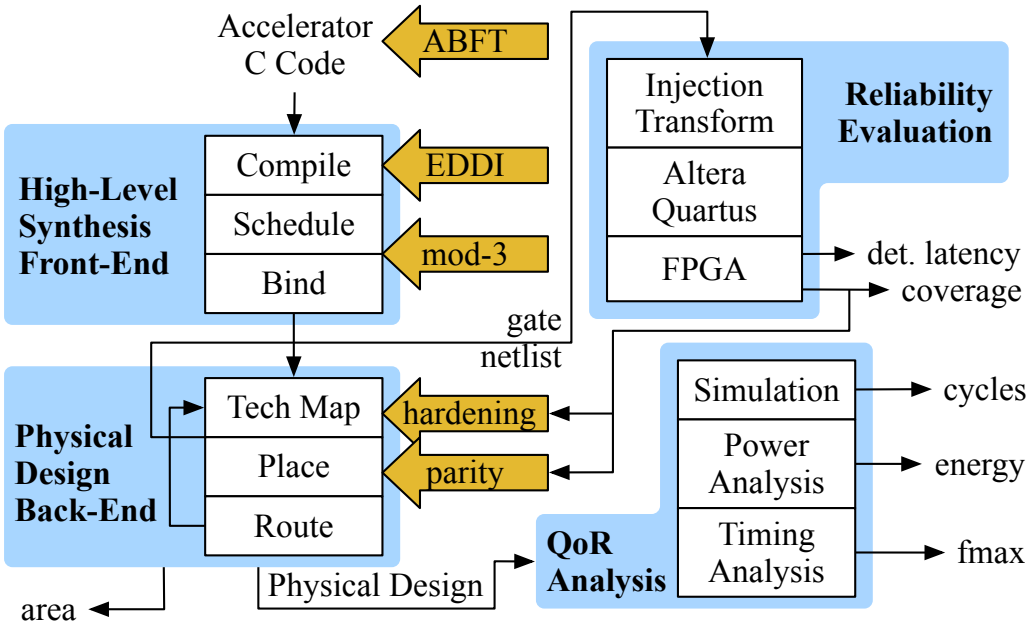


Figure 9.1: Our cross-layer reliability framework.

9.1 Framework

Figure 9.1 provides an overview of our experimental framework which evaluates the reliability and quality of results (QoR) of each of our reliability techniques and combinations for each experimental accelerator design relative to an unprotected baseline. The hardening and parity techniques are guided by reliability evaluation for each individual flip-flop after application of the higher-level techniques; they prioritize protecting the most vulnerable flip-flops first. There is a feedback loop in the back end since the hardening and parity techniques modify the physical design through an engineering change order (ECO) process. We discuss reliability evaluation in Section 9.1.1, our high-level synthesis "front-end" in Section 9.1.2, and our physical design "back-end" and QoR analysis in Section 9.1.3. We introduce our reliability techniques in Section 9.1.4 and discuss techniques for accelerator error recovery in Section 9.1.5.

9.1.1 Reliability Evaluation

To evaluate the reliability of each of the 12 accelerator designs against flip-flop soft errors, we start with a gate netlist as shown in Figure 9.1. We then

Figure 9.2: Error injection enabling transform.

run the netlist through an *error injection enabling* transformation which inserts XOR gates at the "D" input to each flip-flop as shown in Figure 9.2. One input of each XOR gate is connected to the corresponding original "D" driving wire, while the other input is connected to an error injection controller, which enables a bit flip to be injected into a specific flip-flop at a specific cycle. We map this transformed netlist to an FPGA (Altera Stratix III) and generate 10,000 random (cycle, flip-flop) pairs for each accelerator design. Such random error injections at the flip-flop-level have been experimentally shown to accurately model the behavior of soft errors in actual systems [69]. We then execute the accelerator on the FPGA with the same input 10,000 times, injecting one error from the list each time and recording the results for all 10,000 runs.

FPGA emulated error injection is critical in order to evaluate large injected error sample sizes, which in turn is important for accurately estimating the reliability of a given design against *all* possible single flip-flop bit flips. In our experiments, we find that FPGA emulated error injection is on the order of 1,000-10,000× faster than RTL-level simulation. This speed enables us to perform all of our experiments (a total of over 400,000 error injection runs for the baseline and resilient designs) with approximately 40 FPGA-hours of computation time. Running RTL simulations with the same sample size would require a large CPU computation cluster.

There are three possible outcomes for each execution: correct, wrong, and hang. "Correct" means that the accelerator produced the correct output with the correct timing (output exactly matches a "gold" error-free run). In other words, the error is masked. "Wrong" means that the accelerator output does not match the error-free run, typically referred to as Silent Data Corruption (SDC). "Hang" means that the accelerator was given twice the number of cycles as the error-free run to finish, but failed to complete execution in that time. Such hangs can be detected with existing watchdog techniques. We will

focus on the "Wrong" outcomes or SDCs as these problems will go undetected in an unprotected design and are thus the most pernicious effect of soft errors.

We used this reliability evaluation for unprotected baseline designs as well as experimental designs with various reliability transforms applied. The experimental accelerator designs may have an additional error output which indicates that the accelerator has detected the error. If the error is detected, we consider the accelerator to be protected against the corresponding injection since it can respond by restarting its execution (see Section 9.1.5). Recording the cycle count when the error signal is asserted allows us to also measure the *error detection latency* for each error that is detected, calculated as the number of cycles from when the error is injected to when the error signal is raised.

Some reliability transforms may increase the number of flip-flops or the accelerator runtime, which proportionally increases the soft error rate per accelerator execution. Thus we use the following equations for reliability improvement to model this effect:

$$\text{SDC impr.} = \frac{\text{Runtime}_{\text{base}} \times \text{Flip-flops}_{\text{base}}}{\text{Runtime} \times \text{Flip-flops}} \times \frac{\text{SDC}_{\text{base}}}{\text{SDC}} \tag{9.1}$$

$$\text{SDC} = \frac{\text{Wrong, undetected outcomes}}{\text{Total Errors Injected}} \tag{9.2}$$

where Runtime = Cycles×Frequency. Note that our error injection evaluation framework cannot emulate hardened flip-flops on an FPGA. Instead we model such hardening by scaling the wrong, undetected outcome count for just the hardened flip-flops by dividing by the known soft error rate improvement of the hardened flip-flop (see Section 9.1.4).

9.1.2 High-Level Synthesis

High-level synthesis enables us to take reliability methods previously applied to software and retarget them for nonprogrammable custom hardware. Thus, we start with software specifications for each of our accelerators and our hardware synthesis process begins with high-level synthesis (HLS) [70], a process for compiling a software design specification into custom hardware specialized to execute exactly that software functionality and only that software functionality.

To the best of our knowledge, this is the first comprehensive study evaluating effective software reliability methods as applied to hardware.

For our experiments, we used an in-house HLS engine leveraging the LegUp [58] compilation process and scheduler, but with a custom binding and RTL generation engine. As shown in Figure 9.1 the ABFT and EDDI transforms are performed before HLS; the modulo-3 transform is integrated into HLS; and the parity and hardening techniques are applied after HLS. (Section 9.1.4 discusses these transforms individually.)

9.1.3 Physical Design

In order to accurately evaluate the physical design properties (i.e., area, energy, and clock frequency) of each accelerator, *synthesis and place-and-route (SP&R)* is run for each accelerator configuration (both before and after adding resilience). For the ASIC design flow, accelerators are mapped to using a commercial 28 nm technology library and SRAM compiler (the latter is used to generate 2-port SRAM blocks for accelerator output). Synopsys design tools (Design Compiler, IC Compiler, and Primetime) are used to perform synthesis, place-and-route, and power analysis. It is crucial to evaluate area, power, and timing impact post-layout in order to fully capture the impact of physical design (i.e., impact of wire routing and timing constraints). Energy analysis is performed by running VCS simulation (to generate accurate application traces and switching activities for each accelerator) combined with timing and power information obtained from IC Compiler and Primetime. For the FPGA design flow, accelerators are mapped to and analyzed using the same FPGA (Altera Stratix III) platform used for reliability evaluation.

9.1.4 Resilience Techniques

In the following subsections, we elaborate on our experimental resilience techniques, which span the circuit-level to the algorithm-level. Of particular interest are the traditionally software- and algorithm-level techniques applied at the higher levels, since the software transforms in these techniques now become architecture-level hardware transforms through the use of high-level synthesis.

Table 9.1: Hardened Flip-Flops

Type	**Soft Error Rate**	**Area**	**Delay**	**Energy**
Baseline	1	1	1	1
Light-Hardened LEAP (LHL)	2.5×10^{-1}	1.2	1.2	1.3
LEAP-DICE	2×10^{-4}	2	1	1.8

Flip-Flop Hardening

Hardened flip-flops are flip-flops designed to tolerate radiation induced soft errors [36, 37]. Modifications to the flip-flop circuit and layout can reduce the probability (by up to three orders of magnitude [37]) that a particle hit will change the stored flip-flop state. These hardened flip-flops are incorporated into the standard cell library, such that during synthesis and place-and-route, existing (unhardened) flip-flops can be remapped and substituted on a one-to-one basis with their resilient counterpart. It is important to note that the hardened flip-flop design considered in this thesis (LEAP-DICE) tolerates both *single-event upsets (SEUs)* and *single-event multiple upsets (SEMUs)*, which is not the case for all hardened designs. For instance, in DICE (a traditional and well-known hardened flip-flop design) [71], a single particle strike can cause multiple nodes within the DICE cell to flip (i.e., SEMU), resulting in a state corruption. Applying LEAP layout modifications to DICE enables the creation of a new hardened flip-flop (LEAP-DICE), the latter which is tolerant to both SEUs and SEMUs. Compared to a baseline, unprotected, flip-flop, LEAP-DICE provides a 5,000× reduction in Soft Error Rate (SER) at 2× area, no delay, and 1.8× energy cost (as demonstrated with radiation beam experiments conducted by [37]).

Flip-Flop Group Parity Checking

Flip-flop group parity checking is a logic layer technique implemented by comparing the inputs and outputs for groups of flip-flops [72]. For radiation-induced soft errors in flip-flops, it is sufficient to implement this checking by utilizing an XOR-tree to calculate and compare the even-parity of the inputs (predictor tree) to that of the outputs (checker tree), as shown in Figure 9.3. In order to maintain the clock period, it may be necessary to add additional flip-flops to pipeline the parity calculation in the predictor tree

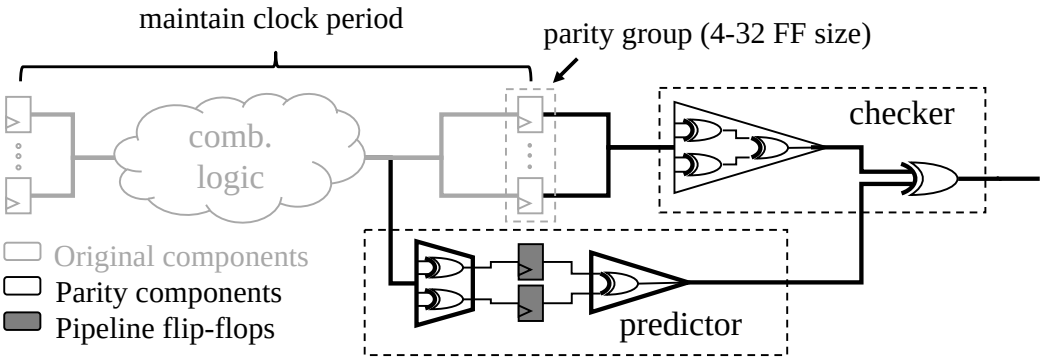


Figure 9.3: Flip-flop group parity checking (no timing impact).

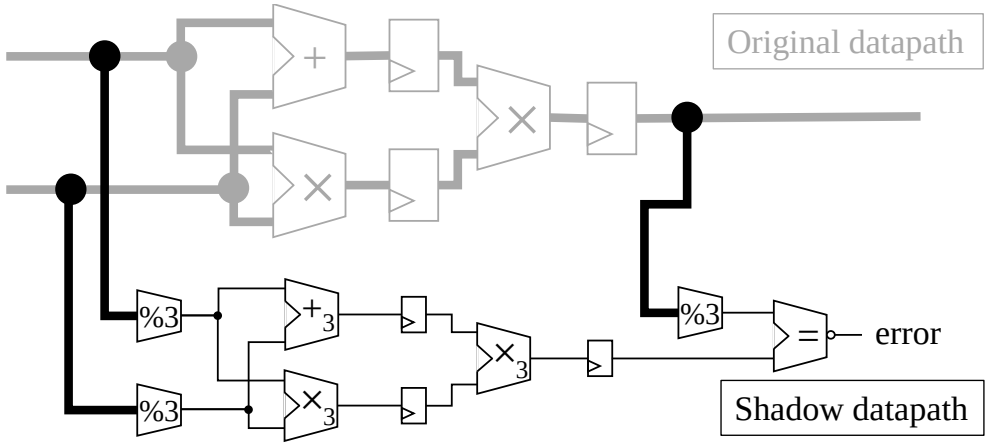


Figure 9.4: Example modulo-3 shadow datapath. Units labeled "%3" are mod-3 residue generators.

(to prevent disturbing critical paths). Implementation of the predictor and checker trees is performed via automatic netlist modifications during synthesis and place-and-route. The same design heuristics described in [38] are used to enable cost-effective implementations of parity checking that ensure no clock speed impact and mitigate the impact of SEMUs.

Modulo-3 Shadow Datapaths

Modulo-3 shadow datapath checking, as discussed in detail in Chapter 7, is a high-level synthesis technique for checking the computation of an arithmetic datapath involving multiple inputs, outputs, and operations. The technique works by creating a "shadow datapath" that performs the same computation as the main datapath, but with modulo-3 residues as illustrated in Figure 9.4. As there are only three unique values in modulo-3 space: 0, 1, and 2, the shadow

datapath is a lightweight version of the main datapath with 2-bit registers and 2-bit operations (such as mod-3 addition and multiplication labeled $+_3$ and $\times_3$ in Figure 9.4). These 2-bit operations are a key advantage of modulo-3 checksums over parity checksums: the ability to perform lightweight checksum *prediction* through an entire datapath. In particular, checksum predictions for addition, subtraction, and multiplication are cheap 2-bit operations for modulo-3 checksums but are expensive for parity checksums.

Thus, while the parity technique puts each flip-flop needing protection into a parity group to protect them *explicity*; modulo-3 residue generators and checkers are only needed at inputs and outputs of an arithmetic core, respectively, to *implicitly* protect all of the flip-flops in that core. We also use the high-level synthesis optimizations discussed in Chapter 7 that further reduce cost through intelligent checkpoint scheduling and binding to share a minimum allocation of modulo-3 residue generators.

In some parts of a datapath, non-arithmetic components such as bitwise logic and shifts may be present. We leave flip-flops in these non-arithmetic parts of the datapath unprotected instead of duplicating them as in Chapter 7. Similarly, we leave the state machine unprotected. The intuition here is that downstream parity and flip-flop hardening techniques provide a fine brush to cover these coverage gaps in a more cost-effective manner.

Instruction Duplication

Error Detection by Duplicated Instructions (EDDI) is a software technique that detects errors through comparison of redundant execution of instructions using separate register and memory partitions [8, 68]. Instruction duplication is implemented using LLVM compiler modifications before high-level synthesis to automatically transform applications to partition the memory space and add the duplicated instructions. The transform also inserts consistency checks between variables and their duplicates, which we pass to a custom check function. We modify our HLS engine to translate these check function calls to logic that asserts an output error signal whenever a consistency check fails. By using a custom check function that generates datapath hardware in our HLS tailored version of EDDI, we avoid the need for conditional branches or other control flow instructions to check for errors found in the software reference transforms in [8, 68].

Algorithm Based Fault Tolerance

Algorithm Based Fault Tolerance (ABFT) is an algorithm-layer technique that can only be applied to specific operations and algorithms to either detect or detect and correct errors [46, 47, 73, 74]. Fortunately, many accelerators are particularly amenable to ABFT modifications due to their prevalent use of matrix operations and other linear computations. For example, a matrix multiplication algorithm can be modified to add an additional column/row checksum to each input matrix, which allows for the resulting output checksums to be used to detect or correct an error in the output matrix. Note that, typically, ABFT manifests as software-only modifications (e.g., no hardware overhead) without additional performance impact in the common case to implement correction (for algorithms in which correction is possible). However, since ABFT is subsequently mapped into hardware checkers for the purposes of generating resilient accelerators, accelerator algorithms were modified to only provide detection capability; thus, saving on additional hardware overhead that would be required in order to provide correction capability. This tradeoff is acceptable for hardware accelerators since they can be restarted when a soft error occurs to effectively correct the error (see Section 9.1.5).

9.1.5 Recovery

To have an end-to-end resilient accelerator, errors need not only be detected but must also be corrected (i.e., recovered). Since the majority of resilience techniques (aside from hardened flip-flops, which perform in-place correction by mitigating the effect of soft errors) have been implemented using detection-only mechanics, an additional recovery mechanism is required. However, a beneficial property of accelerators is that they perform fixed-function computation without external interference (other than to receive input data prior to computation and to transmit output data at completion). In other words, the accelerator inputs essentially provide a snapshot of the accelerator state at the start of execution. As long as the accelerator input memories are protected (we assume memories are protected by existing coding techniques), this snapshot will not be corrupted and a "restore" of this snapshot can be achieved by resetting the accelerator (using existing reset signal(s)) and triggering another accelerator start (using existing control signal(s)). Thus, recovery

Table 9.2: Average Cost and Benefit Across All 12 Accelerators for Individual Techniques Relative to Baseline

Technique	Overhead				SDC Improv.	Avg. Det. Latency (cycles)
	Area	Energy	Freq.	Runtime		
LEAP-DICE	0-2.2%	0-8.8%	0%	0%	1-500×	n/a
Parity	0-3.8%	0-10.6%	0%	0%	1-500×	2
Mod-3	1.7%	3.5%	0.3%	0%	4.3×	732
EDDI	27.6%	33%	2.8%	42.7%	57.4×	7,399
ABFT	11.9%	23.8%	1.4%	8.5%	22.2×	265,980

is simply a matter of restarting computation once an error is detected (with only minimal error signal routing cost and negligible performance impact of re-execution given the rarity of a soft error event). This unique structure of application-specific accelerators serves as a partition between the accelerator and the external environment (input, even streaming input, can be stored until consumed and output can be held until validated), thus making the recovery mechanism described feasible, reasonable, and sufficient.

9.2 Results and Analysis

We explore and architect resilience for 12 accelerator designs: `atax`, `bicg`, `floyd-warshall`,[1] `gemm`, `gemver`, `gesummv`, `matrix`, `matrix-tiled`, `mvt`, `symm`, `syr2k`, and `syrk`. These accelerators are derived from software kernels in the PolyBench benchmark suite [66] involving linear algebra and dynamic programming. Since these kernels involve heavily nested loops that are computationally intensive, they are good candidates for offloading to accelerators. Our focus is on the linear algebra kernels since those applications are amenable to ABFT techniques involving matrix row and column checksums [47]. We also wrote a simple version of the `gemm` generalized matrix multiply kernel that performs only a matrix multiply that we call `matrix` and implemented a tiled version called `matrix-tiled` that performs the computation in 4×4 tile chunks to improve performance at the cost of area.

[1] floyd-warshall does not have a corresponding ABFT transform.

9.2.1 Individual Techniques

It is important to first understand how individual resilience techniques perform standalone (the hardware costs, properties, and resilience improvement afforded by each technique in isolation). Table 9.2 provides an overview of the average QoR cost *overhead* and resilience benefit of each individual technique when applied to each of the 12 accelerators mapped to the ASIC platform. Costs and improvement values were generated experimentally using the physical design and reliability evaluation components of our framework as described in Section 9.1. Since LEAP-DICE and parity checking can be selectively applied to flip-flops to achieve tunable resilience improvement, we report the average cost range and corresponding improvement range. From Table 9.2, it can be seen that software- and algorithm-level techniques (i.e., EDDI and ABFT) do not translate into cost-effective hardware checkers. In general, although these two techniques provide a high degree of SDC improvement, the area and energy costs for achieving this improvement is greater than the cost of protecting every single flip-flop using LEAP-DICE, for example.

9.2.2 Combined Techniques

With a greater understanding of the properties of individual resilience techniques, we can explore the cross-layer design space and analyze the benefits attained through interesting combinations of resilience techniques. Tables 9.3–9.6 present our cross-layer cost-effectiveness results for both the ASIC and FPGA platforms. Combinations of *multiple* resilience techniques are compared against the tunable, single-layer solutions (e.g., LEAP-DICE hardening and parity checking). Single-layer solutions are applied in a selective manner and guided using cross-layer analysis (error masking and propagation through the system stack guide implementation decisions).

Similar to [38], cross-layer combinations are created in a top-down fashion where higher-level techniques (e.g., ABFT, EDDI, and modulo-3) are applied first before subsequently augmenting resilience with lower-level techniques (e.g., parity and LEAP-DICE), as needed in order to achieve the desired SDC improvement. Additionally, selective insertion of parity and LEAP-DICE utilizes the heuristics found in [38] (e.g., critical path, flip-flop location, and

Table 9.3: Average ASIC Cost (area/energy) vs. SDC Improvement for Various Combinations Across 12 Accelerators

Type	**SDC Improvement**				
	2×	**5×**	**10×**	**50×**	**500×**
LEAP-DICE	0.9% / 3.3%	1.2% / 5.0%	1.4% / 5.9%	1.7% / 7.0%	2.2% / 8.8%
Parity checking	1.4% / 4.4%	2.2% / 6.4%	2.6% / 7.3%	3.1% / 8.7%	3.4% / 10.6%
Parity + LEAP-DICE	0.6% / 2.7%	1.0% / 3.9%	1.1% / 5.0%	1.3% / 5.7%	1.7% / 7.4%
Mod-3 + parity + LEAP-DICE	0.7% / 3.6%	2.3% / 4.7%	2.6% / 5.7%	2.9% / 6.5%	3.3% / 8.1%
EDDI + parity + LEAP-DICE	27.6% / 33.0%	27.6% / 33.2%	27.6% / 33.2%	27.6% / 33.4%	28.3% / 34.0%
ABFT + parity + LEAP-DICE	11.9% / 23.8%	12.2% / 24.1%	12.2% / 24.1%	12.3% / 24.2%	12.3% / 24.8%

Table 9.4: ASIC Cost (area/energy) for a 50× SDC Improvement Using Various Combinations Across 12 Accelerators

Benchmark	**LEAP-DICE**	**Parity**	**P+L**	**Mod3+P+L**	**EDDI+P+L**	**ABFT+P+L**
atax	3.1% / 10.7%	8.3% / 16.2%	2.8% / 10.0%	8.4% / 14.7%	33.0% / 37.5%	27.3% / 85.2%
bicg	4.2% / 15.1%	7.5% / 15.7%	3.5% / 10.9%	9.6% / 14.5%	46.7% / 88.9%	11.7% / 19.4%
floyd-warsh	1.0% / 3.7%	1.2% / 3.7%	0.7% / 2.5%	1.5% / 6.1%	18.2% / 54.8%	– / –
gemm	0.3% / 3.1%	0.4% / 2.1%	0.3% / 1.6%	0.3% / 1.7%	7.1% / 9.5%	3.7% / 21.2%
gemver	0.2% / 1.5%	0.4% / 1.9%	0.2% / 1.1%	0.4% / 0.7%	27.6% / 15.0%	10.0% / 5.2%
gesummv	4.2% / 10.1%	4.8% / 14.9%	2.6% / 8.9%	2.5% / 9.0%	61.0% / 32.7%	11.5% / 14.1%
matrix	3.3% / 12.2%	6.6% / 17.4%	2.6% / 10.8%	5.4% / 11.1%	27.3% / 42.4%	11.6% / 43.4%
matrix-tiled	0.3% / 1.4%	0.4% / 1.6%	0.2% / 0.9%	0.3% / 1.1%	20.4% / 35.8%	8.2% / 30.1%
mvt	1.1% / 11.5%	2.6% / 12.7%	1.1% / 8.3%	2.5% / 6.5%	56.9% / 37.0%	37.8% / 37.8%
symm	1.1% / 6.6%	2.5% / 8.8%	0.9% / 6.0%	1.6% / 6.2%	14.7% / 24.6%	2.3% / 18.6%
syr2k	0.9% / 3.8%	1.5% / 4.5%	0.9% / 3.8%	1.4% / 3.9%	4.3% / 34.1%	1.3% / 24.8%
syrk	0.5% / 3.9%	1.2% / 4.8%	0.4% / 3.1%	0.8% / 3.6%	15.5% / 32.2%	12.3% / 44.3%

Table 9.5: Average FPGA Cost (LUT area/energy) vs. SDC Improvement for Various Combinations Across 12 Accelerators

Type	**SDC Improvement**				
	2×	**5×**	**10×**	**50×**	**500×**
Parity checking	9.2% / 4.1%	16.6% / 8.5%	20.4% / 12.4%	25.6% / 22.3%	27.8% / 20.5%
Modulo-3 + parity	26.7% / 19.2%	31.8% / 21.1%	34.1% / 26.6%	39.3% / 35.2%	43.1% / 38.5%
EDDI + parity	107.3% / 174.7%	107.3% / 174.7%	107.3% / 174.7%	107.5% / 174.3%	107.7% / 167.4%
ABFT + parity	95.0% / 87.7%	99.0% / 94.2%	101.9% / 95.6%	104.7% / 93.7%	106.9% / 97.0%

Table 9.6: FPGA Cost (LUT area/energy) for a 50× SDC Improvement Using Various Combinations Across 12 Accelerators

Benchmark	**Parity**	**Mod3+Parity**	**EDDI+Parity**	**ABFT+Parity**
atax	20.4% / 22.8%	31.6% / 32.0%	120.9% / 143.2%	125.5% / 136.4%
bicg	27.3% / 36.5%	26.3% / 29.2%	85.5% / 162.3%	56.6% / 39.5%
floyd-warsh	25.5% / 29.9%	48.5% / 72.0%	91.9% / 154.2%	– / –
gemm	26.4% / 29.4%	28.8% / 35.9%	88.9% / 239.4%	106.3% / 95.3%
gemver	13.9% / 21.5%	41.9% / 38.7%	177.2% / 247.7%	122.2% / 92.1%
gesummv	24.9% / 18.9%	33.4% / 18.3%	81.0% / 147.6%	97.2% / 57.6%
matrix	26.9% / 12.6%	34.6% / 27.2%	48.5% / 82.0%	84.8% / 69.9%
matrix-tiled	30.0% / 19.6%	46.2% / 30.2%	169.4% / 240.3%	23.9% / 12.6%
mvt	20.2% / 17.4%	91.1% / 60.1%	77.1% / 95.3%	220.6% / 189.7%
symm	25.6% / 18.1%	28.2% / 44.5%	115.4% / 255.8%	71.6% / 55.5%
syr2k	34.4% / 20.2%	22.5% / 18.5%	123.4% / 161.3%	104.6% / 100.0%
syrk	31.5% / 20.6%	38.1% / 16.3%	110.4% / 163.1%	138.6% / 181.9%

parity group size aware optimization) to ensure the lowest cost technique is chosen while maintaining design frequency.

Since our cross-layer methodology and framework allows designers to tune resilience improvement targets that may vary depending on the intended domain, Tables 9.3 and 9.5 present the average costs for achieving specific resilience improvements using various combinations when averaged over all 12 accelerators mapped to the ASIC and FPGA platforms respectively. This average provides an overall picture of the cost-effectiveness of various combinations. Note that flip-flop hardening is not applicable to FPGA programming as FPGAs are programmable at the logic level and above, not the physical level.

From Table 9.3 it is clear that, generally, a combination of parity checking and LEAP-DICE is the most cost-effective for the ASIC platform. This efficiency is due to the fact that logic parity and LEAP-DICE selectively protect individual flip-flops, resulting in fine-grained protection of the exact flip-flops that are most vulnerable (determined via accurate flip-flop-level error injection described in Section 9.1.1). Additionally, this property of selective protection also helps explain why protection using LEAP-DICE-only yields resilient accelerators at roughly 1.5% additional area and energy cost compared to the cost-effective solution of LEAP-DICE and parity checking. For the FPGA platform, Table 9.5 shows that parity is the most cost-effective in general.

It is interesting to note that while techniques like EDDI and ABFT do provide high degrees of SDC improvement (Table 9.2), even cross-layer combinations involving these techniques do not yield cost-effective solutions. This is due to the fact that the costs for implementing the necessary hardware checkers for these techniques dominate. In fact, the area and energy costs for the checkers alone are more than the cost of implementing an over-designed resilient accelerator that protects every flip-flop using LEAP-DICE.

One interesting exception is the matrix-tiled benchmark mapped to the FPGA, where the ABFT + parity dominates the other techniques in area and energy cost. This benchmark is a matrix-matrix multiply, the canonical computation for ABFT application, in parallelized 4×4 tile chunks. We find that the high functional-unit resource usage of the benchmark makes the accelerator large, effectively compensating for the ABFT overheads.

In general, it is very difficult to find a more cost-effective resilience solution

than using a combination of parity and LEAP-DICE (or either parity or LEAP-DICE alone). However, for two specific ASIC hardware accelerators (`gemver` and `mvt`), a combination of modulo-3, parity, and LEAP-DICE was able to yield marginal energy savings for all resilience improvements when compared to a combination of parity and LEAP-DICE. For the FPGA platform, we find four accelerators where mod3+parity has an advantage over parity alone: `bicg`, `gesummv`, `syr2k`, and `syrk`. In general, the modulo-3 technique works well for arithmetic-oriented datapaths with sufficient arithmetic complexity to compensate for the cost of modulo-3 residue generators on the inputs and outputs (a 32-bit residue generator occupies about the same area as a 32-bit adder).

Tables 9.4 and 9.6 provide a detailed expansion for the costs to achieve a 50× SDC improvement with various combinations of techniques for all 12 accelerators studied for the ASIC and FPGA platforms respectively.

CHAPTER 10

CONCLUSIONS

In Chapters 4, 5, and 6 we introduced the H-QED technique and its hybrid tracing and hybrid hashing variations which utilize HLS principles for quickly detecting bugs inside hardware accelerators in SoCs in both pre-silicon debugging and post-silicon validation scenarios. Our results demonstrate the effectiveness and practicality of H-QED: up to two orders of magnitude improvement in error detection latency, up to a threefold improvement in coverage, less than 10% accelerator-level overhead, and negligible performance overhead. In our pre-silicon hybrid tracing variation, we demonstrate that the technique can pinpoint the source-code location of logic bug activation and provide a strong hint for potential bug fixes to the hardware designer. Furthermore, these techniques also discovered previously unknown bugs in the widely used CHStone HLS benchmark suite. Through hybrid hardware/software traces and signatures, our techniques minimize intrusiveness during validation. Thus, the combination of QED and hybrid tracing/hashing provides a systematic approach to validation of complex SoCs consisting of processor cores, uncore components, programmable accelerators, and hardware accelerators. Future directions related to H-QED include:

- Use of H-QED for a wide variety of high-level descriptions beyond C and C++ (e.g., various domain-specific languages)
- Use of H-QED for programmable accelerators

In our modulo-3 shadow datapath work in Chapter 7 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 4150× faster than an end result check. We have taken the first step toward 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 175× increase in reliability against soft errors. Future directions related to this research include:

- Adding support for floating-point operations
- Fixing timing errors through rollback combined with frequency-voltage scaling

In Chapter 8, we took a dive into the gate-level design of modulo functional units with the goal of reducing their cost with gate-level architectural optimizations. We introduced new gate-level architectures for Mersenne modulo functional units targeting shadow datapaths for reliability, including a modulo reduction algorithm that maps entirely to full adders and new adder and multiplier designs based on integer counterparts with a wraparound twist. We compared our functional units to the previous state-of-the-art approach in Chapter 7, observing a 12.5% reduction in area and a 47.1% reduction in delay for a 32-bit mod-3 reducer; that our reducer costs, which tend to dominate shadow datapath costs, do not increase with larger modulo bases; and that for modulo-15 and above, all of our modulo functional units have better area and delay then their previous counterparts. We also demonstrated the practicality of our approach with a self-checking multiply accumulate design, which has an overhead of only 12% for a 32-bit main datapath and 2-bit modulo-3 shadow datapath. Future directions for this research include:

- Extending support for modulo bases beyond Mersenne numbers
- Support for fixed-point arithmetic
- Gate-level automation through integration into a logic synthesis engine
- Integration into the high-level synthesis approach of Chapter 7

In Chapter 9, we took a step back and looked at the reliability problem from a cross-layer perspective. We built a first-of-its-kind, comprehensive framework to explore the problem of designing application specific hardware accelerators resilient against radiation-induced flip-flop soft errors on both

the ASIC and FPGA platforms, considering combinations of five existing techniques at different levels of abstraction ranging from the circuit-level to the algorithm-level including modulo-3 shadow datapaths. We applied algorithm- and instruction-level techniques, which are traditionally applied to software, to hardware through high-level synthesis. We found that, in general, a combination of parity checking and LEAP-DICE hardened flip-flops are the most cost-effective. For some arithmetic-oriented accelerators, adding modulo-3 shadow datapaths to this combination results in some additional benefit, even without considering its combinational logic, stuck-at fault, and timing error protection benefits. We also found that ABFT in the context of high-level synthesis incurs significant costs due to additional memory bits and memory accesses required for storing checksum data, which is not a significant problem in a software context.

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 *IEEE/ACM Design Automation Conference*, 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 *IEEE/ACM Design Automation Conference*, 2015.

[3] K. Campbell, L. He, L. Yang, S. Gurumani, K. Rupnow, and D. Chen, "Debugging and verifying SoC designs through effective cross-layer hardware-software co-simulation," in *IEEE/ACM Design Automation Conference*, 2016.

[4] K. Campbell, E. Cheng, S. Mitra, and D. Chen, "Cost-effective cross-layer resilience for hardware accelerators," in *SRC TECHCON*, 2017, to appear.

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

[6] 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 *IEEE Intl. Test Conf.*, 2010, pp. 1–10.

[7] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra, "Quick detection of difficult bugs for effective post-silicon validation," in *IEEE/ACM Design Automation Conference*, 2012, pp. 561–566.

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

[9] 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 *Design, Automation, and Test in Europe*, 2015.

[10] 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 *IEEE/ACM Design Automation Conference*, 2011.

[11] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *Intl. Conf. Computer Design*, 2008.

[12] X. Feng and A. J. Hu, "Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification," in *IEEE/ACM Design Automation Conference*, 2006, pp. 1063–1068.

[13] M. Fujita, "Equivalence checking between behavioral and RTL descriptions with virtual controllers and datapaths," *ACM Trans. Design Automation of Electronic Systems*, vol. 10, no. 4, pp. 610–626, Oct. 2005.

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

[15] J. S. Monson and B. Hutchings, "New approaches for in-system debug of behaviorally-synthesized FPGA circuits," in *Intl. Conf. Field Programmable Logic and Applications*, 2014, pp. 1–6.

[16] J. S. Monson and B. L. Hutchings, "Using source-level transformations to improve high-level synthesis debug and validation on FPGAs," in *ACM/SIGDA Intl. Symp. Field-Programmable Gate Arrays*, 2015, pp. 5–8.

[17] N. Calagar, S. D. Brown, and J. H. Anderson, "Source-level debugging for FPGA high-level synthesis," in *Intl. Conf. Field Programmable Logic and Applications*, 2014, pp. 1–8.

[18] L. Yang, M. Ikram, S. Gurumani, D. Chen, S. Fahmy, and K. Rupnow, "JIT trace-based verification for high-level synthesis," in *Intl. Conf. Field Programmable Technology*, 2015.

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

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

[21] 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.

[22] S.-B. Park, A. Bracy, H. Wang, and S. Mitra, "BLoG: Post-silicon bug localization in processors using bug localization graph," in *IEEE/ACM Design Automation Conference*, 2010, pp. 368–373.

[23] T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *Microarchitecture*, 1999, pp. 196–207.

[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 & 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 Architectural Support for Programming Languages and Operating Systems*, 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 *IEEE/ACM Design Automation Conference*, 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] J. G. Tryon, "Quadded logic," in *Redundancy Techniques for Computing Systems*, R. H. Wilcox and W. C. Mann, Eds. Spartan Books, 1962.

[33] 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.

[34] 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 *Microarchitecture*, Dec. 2003, pp. 7–18.

[35] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: Circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[36] H.-H. K. Lee, K. Lilja, M. Bounasser, P. Relangi, I. R. Linscott, U. S. Inan, and S. Mitra, "LEAP: Layout design through error-aware transistor positioning for soft-error resilient sequential cell design," in *IEEE Intl. Reliability Physics Symp.*, 2010.

[37] K. Lilja, M. Bounasser, S. J. Wen, R. Wong, J. Holst, N. Gaspard, S. Jagannathan, D. Loveless, and B. Bhuva, "Single-event performance and layout optimization of flip-flops in a 28-nm bulk technology," *IEEE Trans. Nuclear Science*, 2013.

[38] E. Cheng, S. Mirkhani, L. G. Szafaryn, C. Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra, "CLEAR: Cross-layer exploration for architecting resilience - Combining hardware and software techniques to tolerate soft errors in processor cores," in *IEEE/ACM Design Automation Conference*, 2016.

[39] M. Nicolaidis, "Carry checking/parity prediction adders and ALUs," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 11, no. 1, pp. 121–128, Feb. 2003.

[40] 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.

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

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

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

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

[45] 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 *Intl. Symp. Quality Electronic Design*, Mar. 2005, pp. 364–369.

[46] G. Bosilcaa, R. Delmasa, J. Dongarraa, and J. Langoub, "Algorithm-based fault tolerance applied to high performance computing," *Journal of Parallel and Distributed Computing*, 2009.

[47] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Computers*, 1984.

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

[49] S. J. Piestrak, "Design of residue generators and multioperand modular adders using carry-save adders," *IEEE Trans. Computers*, vol. 43, no. 1, pp. 68–77, 1994.

[50] N. Nedjah and L. M. Mourelle, "Three hardware architectures for the binary modular exponentiation: Sequential, parallel, and systolic," *IEEE Trans. Circuits and Systems I: Regular Papers*, vol. 53, no. 3, pp. 627–633, March 2006.

[51] H. S. Warren, *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[52] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, 2009.

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

[54] 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.

[55] W. Snyder, "Verilator and SystemPerl," presented at North American SystemC User's Group, June 2004.

[56] W. Snyder, "Verilator: Open simulation - growing up," presented at Design Verification Club, Bristol, Jan. 2013.

[57] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *Intl. Symp. Code Generation and Optimization*, 2004, pp. 75–86.

[58] 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. Embedded Computing Systems*, vol. 13, no. 2, Sep. 2013.

[59] A. Papakonstantinou, K. Guraj, J. A. Stratton, D. Chen, J. Cong, and W.-M. W. Hwu, "Efficient compilation of CUDA kernels for high-performance computing on FPGAs," *ACM Trans. Embed. Comput. Syst., App.-Specific Processors*, vol. 13, pp. 25:1–25:26, Sep. 2013.

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

[61] "Clang 3.9 documentation." [Online]. Available: http://llvm.org/releases/3.9.0/tools/clang/docs/

[62] "Cppcheck: A tool for static C/C++ code analysis." [Online]. Available: http://cppcheck.sourceforge.net

[63] "Valgrind." [Online]. Available: http://valgrind.org

[64] "Clang static analyzer." [Online]. Available: http://clang-analyzer.llvm.org

[65] M. Gao, P. Lisherness, and T. Cheng, "On error modeling of electrical bugs for post-silicon timing validation," in *IEEE/ACM Asia and South Pacific Design Automation Conference*, 2012, pp. 701–706.

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

[67] 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 Trans. Nuclear Science*, vol. 58, no. 6, pp. 2804–2806, Dec. 2011.

[68] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Trans. Reliability*, 2002.

[69] C. Bottoni, M. Glorieux, J. Daveau, G. Gasiot, F. Abouzeid, S. Clerc, L. Naviner, and P. Roche, "Heavy ions test result on a 65nm sparc-v8 radiation-hard microprocessor," in *IEEE Intl. Reliability Physics Symp.*, 2014.

[70] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 2011.

[71] T. Calin, M. Nicolaidis, and R. Velazco, "Upset hardened memory design for submicron CMOS technology," *IEEE Trans. Nuclear Science*, 1996.

[72] L. Spainhower and T. A. Gregg, "IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective," *IBM Journal of Research and Development*, 1999.

[73] Z. Chen and J. Dongarra, "Numerically stable real number codes based on random matrices," in *Intl. Conf. Computational Science*, 2005.

[74] V. S. S. Nair and J. A. Abraham, "Real-number codes for fault-tolerant matrix operations on processor arrays," *IEEE Trans. Computers*, 1990.