Decompose and Conquer: Addressing Evasive Errors in Systems on Chip

by

Doowon Lee

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2018

Doctoral Committee:

Professor Valeria M. Bertacco, Chair
Assistant Professor Reetuparna Das
Professor Scott Mahlke
Associate Professor Zhengya Zhang

Doowon Lee

doowon@umich.edu

ORCID iD: 0000-0003-0046-7746

For my wife, Ghaeun

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Professor Valeria Bertacco. She has affected me in many aspects. She is the most enthusiastic person I have ever met; she always inspires me to achieve more than I could imagine. She has offered an extraordinary research mentorship throughout my Ph.D. journey. She has been consistently available whenever I need her thoughtful advice and keen insight. Also, she has been very patient about my research endeavors, allowing me to navigate different research topics. Her guidance on writing and presentation has been helpful for me to become an outstanding professional after graduation.

I also thank my dissertation committee members, Professors Scott Mahlke, Reetuparna Das, and Zhengya Zhang. Their comments on my research projects have been invaluable for me to improve this dissertation. In particular, in-depth discussions with them stimulate me to think out of the box, which in turn strengthens the impacts of the research proposed in the dissertation.

Our research group members have helped enjoy a long journey to the degree at the University of Michigan. First of all, I am grateful to Professor Todd Austin for his advice on my research projects. I am also inspired by outstanding senior students in the research group: Andrea Pellegrini, Debapriya Chatterjee, Ritesh Parikh, Rawan Abdel Khalek, and Biruk Mammo. I was fortunate to have opportunities to collaborate with some of them, especially when I just joined the research group, so that I could quickly grow as an independent researcher. Also, I thank current students in the group: Abraham Addisie, Zelalem Aweke, Misiker Aga, Vidushi Goyal, Timothy Linscott, Pete Ehrett, Andrew McCrabb, Hiwot Kassa, Mark Gallagher, Lauren Biernacki, Leul Belayneh, Nishil Talati, Colton Holoday, and Tarunesh Verma. I also thank former members of the research group: Salessawi Yitbarek, Brendan West, and Opeoluwa Matthews. I appreciate the help that I received from staffs in the Computer Science and Engineering department. Among them, I would like to give a special thank to Alice Melloni.

The two internships that I had during my study are unforgettable. First of all, I would like to appreciate the great hospitality that I received from the members of IBM Research in Haifa, Israel. In particular, I am grateful to Ronny Morad who offered, arranged, and

managed my internship. I also thank Vitali Sokhin, Tom Kolan, Arkadiy Morgenshtein, and Avi Ziv for their mentorships. During my internship at Intel, I gained a broad industry experience, not particularly related to the research conducted for this dissertation, from my mentors, Truc Nguyen and Oleksandr Bazhaniuk.

My Ph.D. studies in Ann Arbor would have been much harder if I had not provided with financial support. I have been generously supported by the Applications Driving Architecture (ADA) Center and the Center for Future Architectures Research (C-FAR), sponsored by Semiconductor Research Corporation (SRC) and DARPA. Also, I have been supported by several fellowships including IBM PhD Fellowship, Rackham Predoctoral Fellowship, and Rackham International Student Fellowship.

My wife, Ghaeun Lim, has been supportive to finish my degree at the University of Michigan. With her support, I have been able to better focus on my study and research. Our daughter, Jiyo Lee, makes us smile every day. In addition, I thank my parents and parents-in-law for their endless support.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF ALGORITHMS

Algorithm

ABSTRACT

Modern computer chips comprise many components, including microprocessor cores, memory modules, on-chip networks, and accelerators. Such system-on-chip (SoC) designs are deployed in a variety of computing devices: from internet-of-things, to smartphones, to personal computers, to data centers. In this dissertation, we discuss evasive errors in SoC designs and how these errors can be addressed efficiently. In particular, we focus on two types of errors: design bugs and permanent faults. Design bugs originate from the limited amount of time allowed for design verification and validation. Thus, they are often found in functional features that are rarely activated. Complete functional verification, which can eliminate design bugs, is extremely time-consuming, thus impractical in modern complex SoC designs. Permanent faults are caused by failures of fragile transistors in nano-scale semiconductor manufacturing processes. Indeed, weak transistors may wear out unexpectedly within the lifespan of the design. Hardware structures that reduce the occurrence of permanent faults incur significant silicon area or performance overheads, thus they are infeasible for most cost-sensitive SoC designs.

To tackle and overcome these evasive errors efficiently, we propose to leverage the principle of decomposition to lower the complexity of the software analysis or the hardware structures involved. To this end, we present several decomposition techniques, specific to major SoC components. We first focus on microprocessor cores, by presenting a lightweight bug-masking analysis that decomposes a program into individual instructions to identify if a design bug would be masked by the program's execution. We then move to memory subsystems: there, we offer an efficient memory consistency testing framework to detect buggy memory-ordering behaviors, which decomposes the memory-ordering graph into small components based on incremental differences. We also propose a microarchitectural patching solution for memory subsystem bugs, which augments each core node with a small distributed programmable logic, instead of including a global patching module. In the context of on-chip networks, we propose two routing reconfiguration algorithms that bypass faulty network resources. The first computes short-term routes in a distributed fashion, localized to the fault region. The second decomposes application-aware routing computation into simple routing rules so to quickly find deadlock-free, application-optimized routes

in a fault-ridden network. Finally, we consider general accelerator modules in SoC designs. When a system includes many accelerators, there are a variety of interactions among them that must be verified to catch buggy interactions. To this end, we decompose such inter-module communication into basic interaction elements, which can be reassembled into new, interesting tests.

Overall, we show that the decomposition of complex software algorithms and hardware structures can significantly reduce overheads: up to three orders of magnitude in the bug-masking analysis and the application-aware routing, approximately 50 times in the routing reconfiguration latency, and 5 times on average in the memory-ordering graph checking. These overhead reductions come with losses in error coverage: 23% undetected bug-masking incidents, 39% non-patchable memory bugs, and occasionally we overlook rare patterns of multiple faults. In this dissertation, we discuss the ideas and their trade-offs, and present future research directions.

CHAPTER 1

Introduction

Modern computer chips have evolved to integrate multiple chip components on a monolithic silicon die, opening the era of systems on chip (SoCs). Nowadays, state-of-the-art SoC designs include tens of microprocessor cores and accelerators, tens of megabytes of cache memory components, distributed on-chip networks, etc. In particular, such designs include a variety of heterogeneous processing units, each of which is designed to efficiently execute demanding applications. Today, this assembly-of-specialized-units approach to SoC design is driving innovation in this domain. Other critical chip components – especially memory components and on-chip networks – have evolved accordingly, to better support heterogeneous processing units.

Ideally, computer chips should provide correct outcomes at all times. Chipmakers, however, are struggling to ensure the correctness, as the complexity of computer chips is growing in the heterogeneous computing systems. Chipmakers often discover that their chips contain bugs after they are released to customers, causing unforeseen and expensive product recalls.

Moreover, transistors in leading-edge, nano-scale technology nodes are extremely small. Thus, they are prone to process variations that result in a wide spectrum of transistor electrical characteristics. As a result, computer chips fabricated with leading-edge technology nodes are more likely to experience wear-out transistor failures during the lifespan of the chip, compared to those fabricated with trailing-edge technology nodes. Note that even a small transistor failure may render the entire chip dysfunctional, so it is important to prevent, or at least recover, from such failures. Yet, today, chipmakers are still unsuccessful in preventing and/or recovering from silicon failures, because of the high overhead entailed in monitoring the billions of transistor devices that comprise modern SoCs.

Chipmakers invest increasingly more time and resources for verification to keep up with design complexity [86, 5]. Thus, design innovations may be hampered by the burden of verification. In addition, due to increasing transistor density, it is increasingly difficult

to achieve silicon chip reliability. These two challenges, the functional verification and the reliability challenges, must be addressed to unlock and accelerate SoC innovation in the coming years.

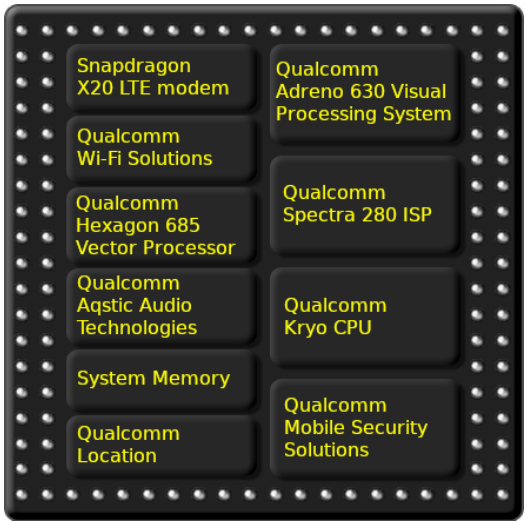
This dissertation's research focuses on these evasive errors and on how to address them efficiently. We propose to decompose software computation and hardware structures involved in addressing them so to solve the smaller, decomposed problems more efficiently. We then assemble the partial results to address the whole problem. Our decomposition methods are designed in a way that the partial results can be easily aggregated to quickly obtain an approximate solution for the whole problem. While achieving efficiency, this aggregation process may not generate the best solution due to a lack of comprehensive reassembly embracing the partial results. In this dissertation, we specialize this *decompose-and-conquer* approach to the major types of components in SoCs.

Dissertation goals. The research presented in this dissertation strives to prevent such erroneous behaviors happening at runtime, by improving the efficiency of finding errors and by reducing the overhead of recovering from them. To this end, we apply the principle of *decomposition* to problems that involve heavy software computation or non-trivial hardware structures. We investigate and learn the reasons for such overheads from prior solutions, which lead us to novel, decomposition-based solutions. Over the course of our research for this dissertation, we looked at each major SoC component: microprocessor cores, memory subsystems, on-chip networks, and accelerators.

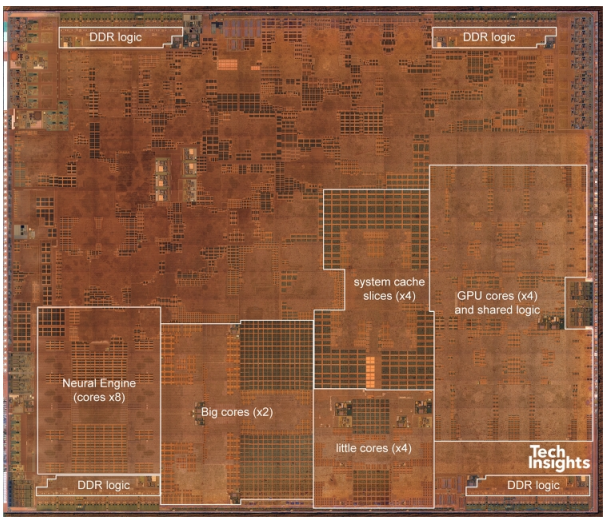
This chapter is organized as follows. We first introduce systems on chip and their components in Section 1.1. We then motivate our research by discussing two challenges that this dissertation is tackling: the functional verification challenge in Section 1.2 and the reliability challenge in Section 1.3. Finally, we outline the solutions that will be discussed in this dissertation in Section 1.4.

1.1 Systems on Chip

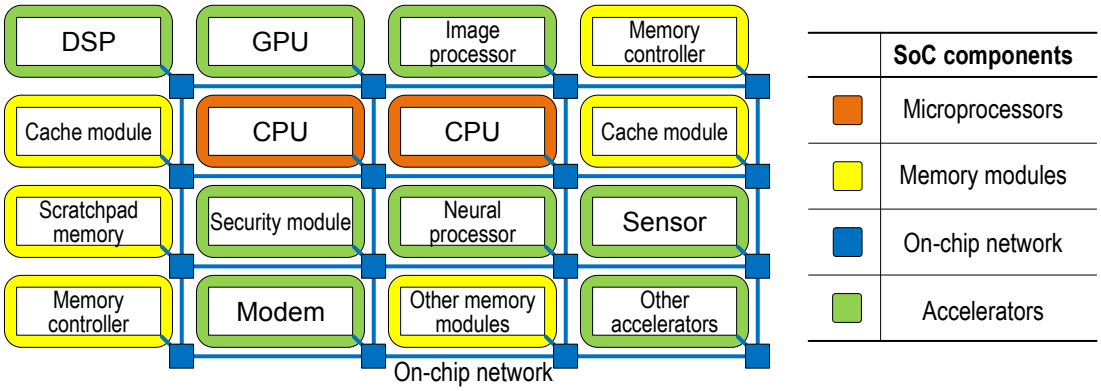
As semiconductor manufacturing technology has kept evolving and improving at a fast pace for over half a century, nowadays, cutting-edge computer chips integrate billions of transistors in a single silicon die. Thanks to this large silicon space, computer chips now consist of many system components that had been previously deployed in separate chips, opening the era of systems on chip (SoCs). There are many benefits of the SoC design paradigm, compared to conventional system designs that are composed of multiple chips



(a) Qualcomm Snapdragon 850 components [13]



(b) A die photo of Apple A12 Bionic [1]



(c) An example SoC design schematic

Figure 1.1: Commercial SoCs and a schematic of an SoC example design

integrated in a printed circuit board (PCB). For instance, SoC designs can eliminate inter-chip communication, which often requires power-hungry high-speed I/O interfaces. Another prominent benefit of SoC designs is a reduction in the manufacturing cost. Fabricating a single chip costs much less than fabricating multiple chips and integrating them on a PCB. We also observe that there are growing potentials with the SoC design paradigm in nano-scale technology nodes. SoC designs can integrate even more components, further improving power efficiency while providing more computation capability, which in turn allows us to enable new computing applications (e.g., artificial intelligence, machine learning, virtual reality, augmented reality) in a small form factor.

Indeed, modern mobile SoCs include tens of components into a single chip, as illustrated in Figure 1.1a. The figure shows major functional blocks in Qualcomm Snapdragon 850 SoC [13], which includes multiple microprocessor cores (Kyro CPU), a graphics processing unit (Adreno 630 GPU), a digital signal processor (Hexagon 685 DSP), an image signal processor (Spectra 280 ISP), system memory, and other components. Figure 1.1b shows a die photo of the Apple A12 Bionic SoC. Some chip components in this SoC have been identified by tech enthusiasts [1], as annotated on the die photo. As the figure shows, this SoC includes not only microprocessor cores, GPU cores, and system caches, but also a neural engine that is dedicated to process machine learning applications. Moreover, it is likely that this SoC includes tens of other unidentified components. These many chip components are connected via on-chip networks that provide efficient, parallel communication among the components. To connect tens or hundreds of SoC components, the communication architecture has evolved to a rich set of on-chip network options, such as multi-layer buses, hierarchical crossbars, and mesh networks. Among them, mesh networks are commonly used in high-end SoCs with many processing units, e.g., Intel Skylake-SP processors [189].

Figure 1.1c shows a schematic of an SoC design example, reconstructed based on these cutting-edge commercial SoCs mentioned earlier. Here, we observe that chip components can be grouped into multiple categories. (1) CPUs (orange boxes) are microprocessor cores with L1 caches co-located with the cores. (2) Memory subsystems (yellow boxes) include L2 and lower level caches, scratchpad memory modules, and memory controllers. (3) Accelerators (green boxes) are processing units that are designed to process specific types of applications. In this dissertation, the term "accelerator" refers to a variety of heterogeneous processing units in SoC designs, except for microprocessor cores and memory modules. (4) Finally, on-chip networks (blue boxes and lines) are communication mediums to transfer data among chip components. We focus on scalable network-on-chip architecture in this dissertation, as the scalability is a key requirement for communication mediums in large

SoC designs in nano-scale technology nodes.

Challenges in systems on chip. While the SoC design paradigm has been widely adopted in the past decade, we observe two major challenges that may limit the sustainability of this paradigm in the future.

(1) **Functional verification challenge**: SoC designs are becoming increasingly complex with respect to the functionalities that they offer. On the contrary, the time to verify and validate SoC designs has remained almost constant over the years. Unfortunately, we observe, from industrialist reports, that the efficiency of functional verification has not kept up with the ever-growing design complexity. For example, in recent Intel processors, many bugs have been found after the processors were shipped to customers, and the number of such bugs is increasing over several generations of Core processors, according to [135]. Thus, it is of paramount importance that we reduce the gap between design complexity and verification efficiency.

(2) **Reliability challenge**: Thanks to advancement in the semiconductor manufacturing technology, the number of transistors integrated in SoC designs is ever increasing. At the same time, the feature size of each transistor is just a few nanometers wide, which corresponds to the length of tens of silicon atoms. Mature semiconductor manufacturing technology should ensure that each transistor operates reliably under the specified operating-condition range. However, reliability requirements are becoming more challenging due to miniscule feature sizes and enormous amounts of transistors. To make things worse, predicting permanent faults in advance (i.e., at design time) that the system may experience at runtime (e.g., wear-out faults) is inherently very difficult, due to the lack of accurate information regarding runtime operating conditions of each individual transistor. Also, the likelihood of fault occurrences is often unpredictable and occurs in a random fashion [114]. In addition, process variation is becoming more prominent as feature sizes keep decreasing. Overall, the reliability challenge is an impending threat to the growth of the SoC design paradigm.

Learning from failures – real-world error examples. To show the significance of these challenges, we show two examples of erroneous behaviors in Figure 1.2, found in Intel processors after they are released to customers. From the descriptions in the specification update documents [104, 108], we identify the erroneous behavior in Figure 1.2a as caused by a design bug, while that in Figure 1.2b is related to a wear-out fault.

Firstly, the problem described in Figure 1.2a is triggered only under certain microarchitectural conditions; the conditions may be very rare, thus they have not been detected during

SKL055	Use of Prefetch Instructions May Lead to a Violation of Memory Ordering
Problem	Under certain micro architectural conditions, execution of a PREFETCHh instruction or a PREFETCHW instruction may cause a load from the prefetched cache line to appear to execute before an earlier load from another cache line.

(a) A problem caused by a design bug in 6th Generation Intel Processor Family [104]

AVR54	System May Experience Inability to Boot or May Cease Operation
Problem	The SoC LPC_CLKOUT0 and/or LPC_CLKOUT1 signals (Low Pin Count bus clock outputs) may stop functioning.

(b) A problem caused by a permanent fault in Intel Atom Processor C2000 Product Family [108]

Figure 1.2: Specification update examples

verification and validation. The instructions mentioned in the description, `PREFETCHh` and `PREFETCHW`, are commonly used in commercial software to improve system performance by hiding memory-access latency. Thus, they are probably well verified for typical use-cases. What makes these instructions buggy is the rare microarchitectural conditions that have not been covered by typical use-cases. This error is very evasive, due to the fact that such rare conditions are unlikely to be exercised within the budgeted amount of time for design and verification.

The problem described in Figure 1.2b manifests only after the chip has been used for a couple of years, according to a news article [15]. As explained in the design revision of the faulty chip [11], this error seems to manifest itself when low pin count (LPC) pins are used for general-purpose input and output (GPIO) signals, a very unlikely use-case.

Common characteristics of design bugs and permanent faults. While design bugs and permanent faults originate from different causes, they share common characteristics that we describe here. Firstly, they cause recurring failures. The system may experience a failure whenever a design bug is triggered or whenever a faulty component is exercised. Thus, the system must patch both bug and fault to avoid failures.

In addition, they occur unexpectedly in a random fashion. Most evasive design bugs are found in rare microarchitectural corner-cases. While verification engineers strive to identify and verify such corner-cases, this endeavor is usually far from complete, due to the limited time allowed for verification. Similarly to design bugs, permanent faults are usually caused by unpredictable physical phenomena. While several fault mechanisms can be predicted analytically or experimentally, as we will discuss in Section 1.3, it is still almost impossible to accurately analyze the massive amount of transistors in SoCs. As a result, the industry practice protects only critical silicon devices, while a majority of silicon devices remain

unprotected.

In summary, we must address these challenges to unlock the potentials of SoC design paradigm in the future. In the following two sections, we discuss each of the challenges in more detail.

1.2 Functional Verification Challenge

According to [203], the objective of functional verification is to remove all bugs from a design prior to chip fabrication. Achieving this goal is becoming increasingly becoming difficult—this section discusses why this is the case.

Growing design complexity. To meet ever-growing computational demands, systems on chip have evolved accordingly to offer higher computational capabilities. Specifically, we observe two innovation vectors in SoC designs: parallelization (i.e., multiple homogeneous processing units) and specialization (i.e., heterogeneous processing units). In addition to these two vectors, we also observe improvements in individual SoC components. Each individual component (e.g., microprocessor core) has been increasingly complex to offer a better performance and more features over time.

(1) **Complexity of parallelization**: SoCs employ increasingly more processing units, such as microprocessor cores, graphics processing units, and other accelerators. These processing units can communicate with each other, and share the common memory space for efficient multitasking and multithreading. To this end, as Chen and Kung [66] predicted a decade ago, today's SoCs deploy on-chip networks and various types of shared memory. To be specific, SoCs often include distributed, shared, multi-level cache structures that maintain cache coherency and memory consistency. Also, their on-chip networks can process many communication requests from all processing components, handling hundreds of, or even more, transactions concurrently. Consequently, parallelization leads to highly sophisticated memory subsystems and on-chip networks across the entire chip.

(2) **Complexity of specialization**: The diversity of SoC components is on the rise. Traditionally, microprocessor cores have been major computation components for a wide range of applications. In contrast, state-of-the-art SoCs include many accelerators, each of which is specialized for a certain domain of applications. For instance, the neural engine in the Apple A12 Bionic SoC in Figure 1.1b is designed for accelerating the computation of machine-learning applications.

This specialization makes SoC verification even more strenuous. Most accelerators,

if not all, share on-chip communication mediums, i.e., memory subsystems and on-chip networks. Thus, these components must be verified extensively over many interaction scenarios. Moreover, various types of accelerators may interact with each other. Some accelerator interactions may not have been verified throughout the verification process of individual accelerators. Thus, a solid SoC integration process should verify all accelerator interactions. However, achieving such complete verification is an almost impossible feat due to increasing heterogeneity.

(3) **Complexity of new features and optimizations**: Each component has been improved for performance, reliability, and efficiency. In other words, each component, even without involving other SoC components, is increasingly sophisticated to support additional features and optimizations over its previous generation. For instance, the verification of microprocessor cores still remains one of the most challenging verification tasks, because of continuous innovations related to their architectures and microarchitectures. In particular, new instructions have been added over several generations of Intel x86-based processors and IBM POWER processors. Moreover, microprocessors implement various advanced features, for instance, for performance and security. For example, Intel's high-end processors offer cache allocation technology (CAT) for isolation and prioritization of key applications, and software guard extensions (SGX) for protecting selected code and data from unauthorized disclosure or modification.

Lack of scalable verification. Functional verification methodologies must progress accordingly with the pace of growing design complexity. However, verification is evolving at a slower pace than design complexity. Consequently, chipmakers spend increasingly more time on verification than on design [86]. Even with more verification efforts, multicore processors are subject to increasingly more escaped bugs [135].

Functional verification involves various activities including formal verification and simulation. Most activities, except for manual labors such as writing or reviewing code, involve software computation or equivalent hardware resources (e.g., simulation accelerators [74], FPGA-based emulation platforms, post-silicon validation platforms). Thus, the vast computation and resources involved in achieving coverage closure are a major showstopper, which can be mitigated by two distinct solution approaches: scalable verification (i.e., raising design abstraction for verification) and efficient utilization of hardware resources dedicated to support verification:

(1) **Challenges with scalable verification**: A successful abstraction model for the design under verification can greatly reduce the computational complexity involved in both formal verification and simulation-based verification, thus, enabling scalable verification.

However, even with successful abstractions, it is very time-consuming to simulate or analyze the model of a large SoC design exclusively with software simulators. In addition, inaccurate abstract models may generate false results (e.g., undetected bugs, falsely detected bugs).

(2) **Limited observability in post-silicon verification or hardware-assisted simulation/emulation**: While hardware-assisted or post-silicon verification can greatly boost the verification process, these verification platforms come with a great shortcoming: limited observability. For instance, post-silicon verification leverages hardware tracing mechanisms to record runtime activities. However, these tracing mechanisms usually have limited capacity and are able to record only a handful of information regarding SoC internal state. Moreover, even if all internal signals could be observed, it is still very challenging to efficiently analyze enormous test results generated on the fly. Thus, to maximize the benefits of these fast verification platforms, we must address their limited observability, as well as the result-checking computation.

Our decomposition strategy for functional verification challenges. We observe that a major obstacle in functional verification is a lack of scalable verification methodologies. To overcome this obstacle, we strive to decompose complex software computations involved in verification into smaller granularities. We focus on architecture-level and microarchitecture-level verification problems, because they enable higher verification efficiencies.

In addition to software computation for verification, this dissertation also considers runtime patching hardware to protect SoC designs from design bugs. Prior patching methods involve global hardware structures that span the entire chip. Such global hardware structures are prohibitive in large-scale SoC designs because of their high implementation overheads in terms of silicon footprint and power consumption. Instead, we propose to decompose global hardware structures to smaller, distributed hardware structures. In particular, we focus on runtime patching for memory subsystems and on-chip networks. This is because many chip components interact with these subsystems, thus even small errors in them may invalidate the entire chip.

1.3 Reliability Challenge

As transistor fabrication technology evolves to shrink the feature size of transistors, it becomes increasingly difficult to ensure the reliability of transistors. Many transistors are prone to failures during their expected lifetime. According to a former Intel fellow, Shekhar

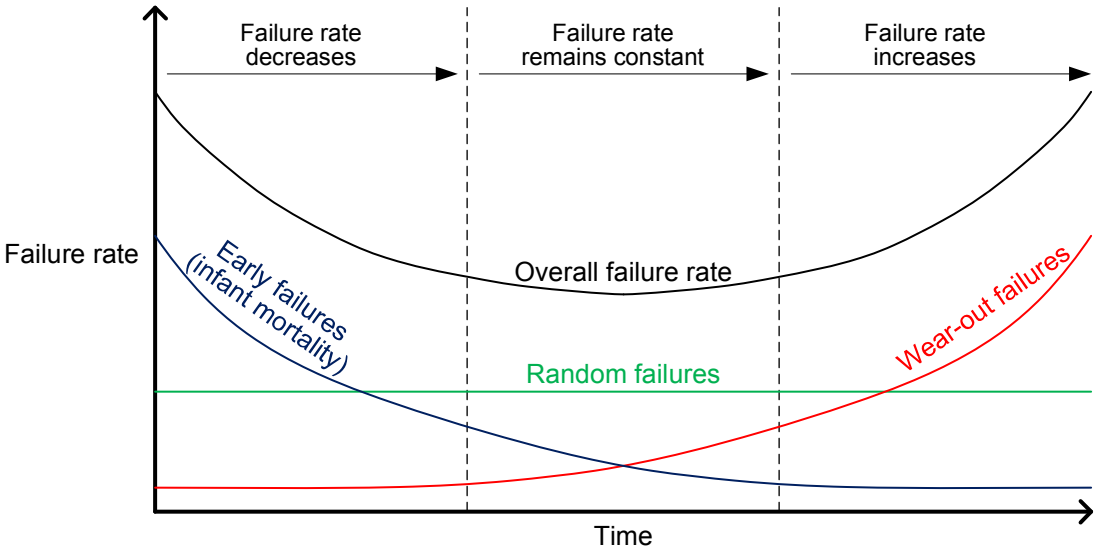


Figure 1.3: Bathtub curve illustrating qualitatively the exposure to faults by a silicon chip over its lifespan (adapted from [3]).

Borkar, in his MICRO 2004 keynote presentation, the company is building "reliable systems with unreliable components." He pointed out three different causes that make a system unreliable: soft errors (transient faults), gradual errors (variations), time-dependent degradation. Soft errors are induced by high-energy particles such as alpha particles and cosmic rays, but these are occasional and do not persist. Gradual errors and time-dependent degradation originate from various sources, rendering a chip dysfunctional so that it would consistently generate erroneous results. In this dissertation, we address only permanent faults, due to their severity.

Figure 1.3 illustrates the probability of permanent failures, showing three distinct periods of time. Each period is dominated by a different cause of failures: the first period—early failures (also known as infant mortality), the second period—random failures, and the third period—wear-out failures. Thanks to the burn-in process, infant mortality has been greatly reducedin modern chips, while the other two remain significant. Specifically, failures due to wear-outs must be reduced so guarantee a sufficient lifespan for semiconductor chips, as Borkar pointed out.

Wear-out mechanisms. There are several wear-out mechanisms identified in the literature [188, 201]. (1) Electromigration (EM) is an atomic motion caused by a current flowing through a line. At high current densities, a significant atomic motion will occur and increase the line resistance, ultimately breaking it. (2) Hot carrier injection (HCI) (Figure 1.4a) is another wear-out mechanism, where high-energy electrons cross the Si/SiO_2 interface

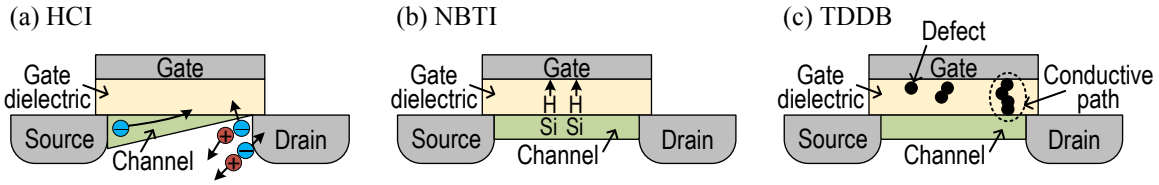


Figure 1.4: Three MOSFET wear-out mechanisms (adapted from [201]). (a) Hot-carrier injection (HCI), (b) negative bias temperature instability (NBTI), and (c) time-dependent dielectric breakdown (TDDB).

while traversing the channel in a field effect transistor (FET). The electrons are trapped in the gate or drain, thus not contributing to the current flowing through the channel. This situation, in turn, degrades the electrical performance of the transistor. (3) Negative bias temperature instability (NBTI) (Figure 1.4b) is caused by an interfacial layer with nitrided silicon dioxide at the Si/SiO_2 interface creating positively charged bulk defects. As a result, an NBTI effect modifies the key parameters of a MOSFET (metal-oxide-semiconductor FET), such as its threshold voltage, linear and saturation drain currents, and transconductance. (4) Time-dependent dielectric breakdown (TDDB) (Figure 1.4c) is a failure mechanism that leads to traps within the dielectric. This breakdown occurs when a high voltage is applied to the gate, creating a relatively strong electric field, yet not strong enough to cause an immediate breakdown. The electric field creates traps within the dielectric, eventually forming a conductive path through the gate oxide to the substrate. Note that there are other wear-out mechanisms that are not mentioned here (e.g., positive bias temperature instability (PBTI)), which also cause permanent faults in similar manners.

Researchers have proposed analytical models to predict wear-out failures (e.g., [200]). However, these models often rely on unpredictable factors (e.g., temperature, transistor activity). Even if these factors can be approximated, such transistor-level prediction may involve prohibitively heavy computation when considering the billions of transistors integrated in SoCs. Thus, instead of relying on analytical prediction models (i.e., design-time wear-out predictions), we advocate runtime solutions to detect the occurrence of permanent faults and recover from them.

Minimizing the impact of faults at runtime. SoCs may experience permanent faults at random locations, impacting both transistors and wires. To overcome these random failures, SoCs can deploy in-field fault detection and diagnosis mechanisms. For instance, built-in self-testing (BIST) can thoroughly examine chip components during the system's booting process. Alternatively, the system can be continuously monitored and examined for faults during its normal operation (i.e., online testing). Unfortunately, both BIST and

online testing may incur a long overhead, impacting the system's availability during testing.

Moreover, once faults are discovered, the SoC must reconfigure itself to bypass them. Fault-bypassed systems, unfortunately, still suffer from performance degradation to some degree, compared to their fault-free counterparts. Thus, it is important to seek a reconfiguration solution that minimizes performance degradation. However, this goal may require significant computation, especially when reconfiguration entails system-wide changes, such as when overcoming faults in an on-chip network.

Our decomposition strategy for reliability challenges. We propose to leverage structural decompositions to address reliability challenges at the microarchitectural level. SoC components consist of multiple microarchitectural components that can be clearly separated for fault analysis. For instance, on-chip networks can be split into individual routers and links. Upon a fault occurrence, fault diagnosis can then analyze only the portion of the network affected by the fault, instead of the whole network. Although a global fault diagnosis could produce more accurate results, it would incur significantly higher runtime overheads, compared to a local analysis.

In this dissertation, we have applied this decomposition strategy to on-chip networks. However, we believe that the other SoC components (i.e., microprocessor cores, memory subsystems, and accelerators) can benefit from the same decomposition strategy. Indeed, several prior works have proposed similar decomposition-based methodologies for microprocessor cores and memory subsystems, as we will discuss in Chapter 2 and the related work sections in Chapters 3 through 5.

It is important to note that our focus is to reduce the overheads entailed by the computations required in detection and recovery. To this end, our decomposition strategy remains at the microarchitectural level, rather than going down to a lower level (e.g., gate or circuit level), since a lower-level approach would be more time consuming and require more spare components, due to its finer granularity.

Differences between design bugs and permanent faults. As we discussed in Section 1.1, design bugs and permanent faults share a major common characteristic: their impact is permanent, so they cause recurring system failures. However, design bugs usually originate from human errors within the design description (e.g., Verilog or VHDL code), while permanent faults are caused by physical phenomena.

Due to this difference, design bugs and permanent faults should be handled differently. Design bugs are usually found in functional corner-cases, which are very hard to foresee and predict at design time. On the contrary, some permanent faults are indeed predictable.

Table 1.1: Proposed solutions in this dissertation

SoC part	Solution	Complex challenge	Decomposition method
μprocessor core	**BugMAPI** (Chapter 3)	Post-silicon verification (bug-masking analysis)	Instruction-level bug-masking probabilities
Memory subsystem	**MTraceCheck** (Chapter 4)	Post-silicon verification (verifying memory orderings)	Incremental result checking
	μMemPatch (Chapter 5)	Runtime bug patching (catching rare μarch state)	Distributed FSMs
On-chip network	**BLINC** (Chapter 6)	Runtime fault recovery (re-routing latency)	Localized short-term route computation
	FATE (Chapter 7)	Runtime fault recovery (application-aware routing)	Localized application-aware routing rules
Accelerator	**AGARSoC** (Chapter 8)	SoC integration verification (verifying acc. interactions)	Disintegrating accelerator interactions into basic elements

For instance, high-activity transistors are more likely to be impacted by electromigration than low-activity transistors. If high-activity transistors could be accurately identified, they could be protected accordingly, although doing so may require significant computation. Having said that, this dissertation targets unpredictable permanent faults that manifest unexpectedly in a random fashion.

In addition, permanent faults can be recovered easily with spare resources, which may entail noticeable area overheads in the design. Smarter fault recovery solutions help reduce these overheads. Design bugs, however, cannot be patched with identical spare resources, because the spares would have the exact same bugs. Instead, design bugs can be bypassed by exploiting functional redundancy in the design.

1.4 Contributions and Outline of This Dissertation

In this dissertation, we propose decomposition-based solutions to overcome functional verification and reliability challenges. As discussed in Sections 1.2 and 1.3, major obstacles to solve are: (1) *complex software computation* and (2) *heavy hardware resources* required to detect and overcome errors. Our solutions target each major SoC part: microprocessor cores, memory subsystems, on-chip networks, and accelerators, as shown in Table 1.1.

Indeed, each SoC part may suffer from distinct subtle errors that may lead to critical failures at runtime. Many researchers have investigated solutions to these errors, as we overview in Chapter 2 and, in more detail, in the related work and background sections throughout the dissertation. We identify inefficiencies in these prior solutions with respect to the software computation they entail and hardware structures they require, and we strive to address these inefficiencies, as shown in the third column of Table 1.1.

The fourth column of Table 1.1 summarizes the decomposition methods that we propose in this dissertation. In the rest of this section, we briefly introduce the contributions of each solution.

BugMAPI [121] (Chapter 3). This solution presents a bug-masking analysis for post-silicon microprocessor verification. To reduce the computation required for the analysis, *we propose to decompose the bug-masking probability computation from a program to its instructions*. Instruction-level probabilities are then assembled by our information-flow analysis framework, which quickly estimates the bug-masking probability of the whole program. We also demonstrate that the results of this analysis can be leveraged to greatly reduce the likelihood of bug masking by deploying a simple code instrumentation process.

MTraceCheck [120] (Chapter 4). Memory subsystems exhibit a variety of memory-ordering behaviors that must be checked for conformance to the memory consistency model. This result-checking requires heavy computation, limiting efficient utilization of post-silicon validation platforms. To reduce this computation, *we propose an incremental verification scheme that checks only incremental differences on the memory-ordering patterns across multiple test runs*. We also present a novel signature that encapsulates memory-ordering patterns, which in turn can maximize the benefit of the proposed incremental verification scheme.

μMemPatch [122] (Chapter 5). Memory subsystems comprise many sub-components that interact with each other. Thus, pinpointing buggy behaviors in them often requires global hardware monitors across the entire chip. To eliminate such global hardware monitors, *we propose a distributed patching solution for memory bugs, where each core in the system is augmented with a small programmable patch module*. Each distributed patch module observes speculative memory operations and data races occurring at the nearby core and L1 cache. The patch module detects microarchitectural-event sequences leading to bug-prone state, then recovers from the bug-prone state.

BLINC [123] (Chapter 6). Routing reconfiguration often involves long latency to find alternative routes when original routes are deemed unavailable (e.g., due to fault occurrences). This long latency may significantly reduce network performance during reconfiguration. To minimize reconfiguration latency, *we propose a routing reconfiguration framework that can quickly compute short-term, alternative routes in a distributed fashion*. The proposed framework enables a transparent, online network testing that continuously ex-

amines network components, while incurring minimal perturbations on normal network operations.

FATE [124] (Chapter 7). Application-aware routing can maximize the performance of fault-ridden networks. However, it involves heavy route-evaluation computation to estimate traffic flow over many route options. *We reduce this computation by deploying localized routing rules, called turn-enabling rules*. These rules guide the route-evaluation process by aggressively pruning undesirable route options, thus, eliminating unnecessary traffic-evaluation computation.

AGARSoC [137] (Chapter 8). SoCs exhibit various accelerator interactions, as they integrate many heterogeneous accelerators. To expedite the verification of accelerator interactions, *we propose to disintegrate accelerator interactions observed from test executions into basic interaction elements.* Afterward, we create new, interesting test-cases by stitching basic interaction elements together. This verification framework utilizes high-level design models to quickly collect probable accelerator interactions.

In Chapter 9, we conclude this dissertation by discussing trade-offs of the proposed decomposition-based solutions. In addition, we discuss some future research directions that can improve the solutions proposed in this dissertation.

To start, the next chapter will provide a short introduction to the extensive body of research covering the topics of validation and reliability for key SoC components.

CHAPTER 2

Related Work

Researchers in academia and industry alike have proposed various solutions to protect system-on-chip (SoC) designs against design bugs and permanent faults. Here, we review some microarchitectural and architectural solutions to address errors in major SoC components. Various solutions have been proposed with the goals of providing error detection and error recovery. Most prior solutions are specialized in either error detection or error recovery, while some offer both detection and recovery. The following sections outline detection and recovery solutions on microprocessor cores, memory subsystems, on-chip networks, and accelerators. Additional discussion of solutions in these domains is provided in the background and related work sections of subsequent chapters; the overview provided here summarizes research contributions in the broad domains of verification and reliability, while, in later chapters, we focus the presentation to work closely related to our solutions.

2.1 Addressing Errors in Microprocessor Cores

Microprocessor cores have been extensively studied with respect to their reliability against faults. Recently, fine-grained microarchitectural solutions have been proposed to address faults while incurring low overheads, compared to conventional coarse-grained (i.e., core-level) solutions. In addition to these fault-tolerant solutions, design bugs have been addressed by various solutions that augment microprocessor cores with small monitor and checker modules, so to readily detect and diagnose the bugs. Unlike these prior works, a solution proposed in this dissertation, BugMAPI, targets bug-masking occurrences that *may* happen during the verification of microprocessor cores. It quantifies the exposure to possibility of bug-masking incidents, and improves the efficiency of post-silicon verification.

Error detection. Many mission-critical systems, such as airplanes, rely on Dual-Modular Redundancy (DMR). DMR leverages a hardware duplicate; it compares the outcome of the duplicate hardware to that of the original one. Mismatches between the two outcomes imply that errors might have occurred either in the original hardware or the duplicate hardware. Similarly, software-based redundancy techniques can be deployed at no hardware cost. In other words, software can run the same program twice to verify program execution results. For instance, SWIFT [168] duplicates a program's basic blocks, catching any discrepancy at the basic-block level. While DMR and SWIFT are very effective in detecting soft errors (transient faults), none of these solutions are effective against design bugs and permanent faults. Specifically, DMR cannot detect design bugs because both copies of the hardware are identically buggy. Software-based duplication (e.g., SWIFT) cannot detect any design bug or permanent fault, as two software executions use the same erroneous hardware. Moreover, the overheads of deploying these solutions are prohibitively expensive: $2\times$ silicon footprint (for hardware duplication) or runtime (for repeated software execution) of an unprotected system. Fortunately, some recent proposals have offered less expensive solutions: for instance, Shoestring [83] takes a probabilistic approach to identify the vulnerable portions of the code (i.e., user-visible faults) so that they can be protected via instruction duplication. However, the fundamental limitations of this family of approaches apply to Shoestring as well, as it can detect neither design bugs nor permanent faults.

DIVA [50] and Argus [141] are two original approaches in the hardware redundancy space. Specifically, DIVA [50] leverages a checker processor that is well verified and also more robust to physical failures than the core processor (i.e., the processor being checked by the checker processor). For instance, the checker processor is crafted for simplicity, while the core processor supports full-fledged features for high performance, such as out-of-order execution and instruction-level parallelism. The checker processor is run immediately after the core processor executes the program code, then the execution results are compared between the two. Overall, DIVA provides very good coverage for error tolerance: both error detection and recovery against both design bugs and permanent faults. However, its area and runtime overheads can be expensive for simple cores (e.g., a core processor with an in-order execution engine that does not have full-fledged performance features).

Argus [141] specifically targets this limitation of DIVA, providing a low-cost error-detection solution for simple cores. It deploys a set of small checkers across the in-order execution pipeline stages of the processor, so as to check the results of program execution at runtime. Specifically, to check both control and data flows, it leverages signatures that are computed statically at compile time, instrumented in program code, and checked at runtime. In addition, arithmetic and logical operations are checked approximately with dedi-

cated sub-checkers, which are much smaller than their regular functional unit counterparts. Nostradamus [152] extends Argus for out-or-order cores, while keeping implementation overheads low.

On a different direction, IFRA [159] offers a set of in-circuit recording structures at the processor's microarchitectural components. It deploys error-detection checkers similar to Argus, but it also provides a snapshot of the microarchitectural activities when the error is detected. The activities are then analyzed to find the exact root cause. Similar to IFRA, BackSpace [76] provides a debugging capability that leverages trace buffers augmenting the processor. From signals stored in trace buffers, BackSpace performs formal analysis to reconstruct the history of activities that led to the buggy state, thus, striving to overcome limited observability in post-silicon verification.

Besides the hardware-based detection mechanisms discussed so far, there is a body of work using software-based testing. To detect and diagnose permanent faults, Constantinides et al. [70] proposed to run periodic, directed tests with specialized Access-Control Extension (ACE) instructions. The ACE instructions provide good accessibility and observability to the underlying microarchitectural state, achieving about 99% fault coverage with less than 6% performance and area overheads. Note that, unfortunately, no design bugs can be detected with this technique.

To detect design bugs, Reversi [198] transforms a given test program into another program whose results can be easily checked. To this end, it exploits reversible instructions in the instruction set. For instance, an addition instruction $r2 = r0 + r1$ can be checked with a pair of $r2 = r0 + r1$ and $r3 = r2 - r1$; here, $r3$ must be equal to $r0$ regardless of the values of $r0$ and $r1$. Foutris et al. [87] propose another self-checking method that exploits functional redundancy within the instruction set. For instance, the result of a multiply-and-accumulate instruction can be checked by comparing it with the result of a sequence of multiply and addition instructions; these two results must be identical in the absence of errors. This functional redundancy-based method is more flexible than Reversi, as it can utilize equivalent operations across different classes of instructions. For example, to detect design bugs at the floating-point unit, floating-point instructions can be compared with integer counterparts, or vice versa.

QED [129] is another software-based testing approach that is specialized for detecting electrical errors in post-silicon validation. Its approach is similar to SWIFT in using duplicated instructions. In QED, the results of the duplicated instructions are compared immediately with those of their original counterparts. Thus, QED achieves a significant speedup in detection time, compared to a conventional application-level checking, where results are checked only after the application finishes.

Cheng et al. [67] conducted a comprehensive study on soft-error-tolerant mechanisms for microprocessor cores. They discuss the results of quantitative comparisons across various fault-tolerant mechanisms, spanning across different layers of resilience: from the circuit level, to the logic level, to the architectural level, to the software level, to the algorithmic level. They conclude that a carefully-optimized combination of circuit-level hardening, logic-level parity checking, and microarchitectural recovery provides a cost-effective solution against soft errors. Unfortunately, this work provides little insights on how to address permanent faults and design bugs. However, a similar comprehensive, cross-layer evaluation would be beneficial to minimize the overheads in addressing permanent faults and design bugs, compared to individual single-level solutions.

Error recovery. Once design bugs or permanent faults are detected in a system, the system must be repaired to bypass the faulty components so that failures can be avoided at runtime. To this end, Triple-Modular Redundancy (TMR) deploys two additional copies of the original hardware. The three copies of the hardware run the same software. When there is any discrepancy on the results among the three, the majority of the results is considered to be correct. This majority voting scheme can efficiently tolerate soft errors, because each soft error usually affects a specific transistor or wire in one of the three hardware copies. When it comes to permanent faults, however, its recovery capability is very limited. This is because the entire faulty processing unit has to be disabled even upon a single fault occurrence.

To improve error tolerance to permanent faults, [94, 161, 160] have proposed fine-grained repair solutions at the microarchitecture level. In multicore systems, there are multiple, identical hardware components for each microprocessor pipeline stage (e.g., fetch, decode, execute, and writeback). Upon a fault occurrence, the pipeline can be adjusted to exclude the faulty component and to use a healthy component available in another pipeline. When a second fault occurs, unlike TMR, it does not necessarily disable another pipeline, except for the case where it impacts the same pipeline stage as the first fault. Note that these fine-grained solutions share the same decompose-and-conquer idea as solutions proposed in this dissertation. These solutions decompose the processor pipeline into individual stages, performing the recovery at the pipeline-stage granularity. In particular, StageNet [94] uses full-crossbar switches that are placed between two consecutive pipeline stages (e.g., fetch-decode). The switches enable flexible connections between the consecutive pipeline stages. Viper [161] and Cobra [160] strive to improve fault tolerance by providing a flexible pipeline scheme. In these solutions, a group of instructions are treated as a bundle, which is a basic unit of instruction processing. Individual bundles contend with other

bundles to obtain pipeline resources required to make progress until they arrive at the last pipeline stage (i.e., the commit stage).

While these fine-grained solutions exhibit a very good error tolerance, they often come with high implementation overheads. Unlike them, another body of solutions tries to keep implementation overheads low while providing sufficient error tolerance to support the expected lifetime of the chip.

Among these solutions, [163] exploits the inherent redundancy in many-core systems with homogeneous ISA, and advocates for architecture-level fault tolerance. The proposed method entails a very small extra hardware for thread migration between faulty cores and healthy cores. For instance, when the proposed method discovers that a core cannot properly execute a special instruction, the corresponding thread is then migrated into another core that can handle this instruction. DIVA [50] is a similar architecture-level solution, leveraging two processors (i.e., a core processor and a checker processor) sharing the same ISA. Its error correction is instant; once a discrepancy is found, the result from the checker processor overwrites the result of the core processor.

Notice that all the solutions mentioned earlier require some sort of dedicated hardware. Such dedicated hardware could be relatively expensive for simple cores like the OpenRISC 1200 core. Software-only solutions are more promising for these simple cores. For example, Detouring [142] provides a software translation mechanism to bypass faulty hardware components, by translating the instructions that require the faulty unit into an equivalent sequence, similarly to [87].

2.2 Addressing Errors in Memory Subsystems

Memory subsystems are crucial components that enable high-performance SoCs, because they are shared by most processing units, handling data requests from them. Memory subsystems consist of multiple main memory channels, multiple levels of caches, and other types of on-chip memory modules, such as scratchpad memory modules. Many proposals strive to address either faults or design bugs in memory subsystems. With respect to faults, many solutions strive to protect memory cells, because they are usually smaller than logic gates thus more vulnerable to faults. In addition, another body of solutions strive to address design bugs in complex features of memory subsystems, such as cache coherence and memory consistency. Achieving this goal is becoming more difficult, because of the wide range of memory operations deployed in heterogeneous processing units. Note that our solutions, discussed in Chapters 4 and 5, concern these subtle design bugs in memory subsystems.

Error detection. The reliability of memory subsystems has been one of the major concerns in realizing large-scale computing facilities, such as supercomputers and data centers. Indeed, unreliable memory subsystems can make the entire system worthless in the worst case. One of such catastrophic failures happened in the Virginia Tech's advanced research computing facility built in 2003. The memory modules used in the entire facility had to be replaced within three months of deployment, due to frequent random failures. It turned out that these frequent failures were originated from unprotected, non-ECC (Error-Correcting Code) memory, which is vulnerable to cosmic rays impacting memory cells [130].

Among fault-tolerant solutions, ECC memory provides very good protection against soft errors for memory cells in both DRAM main memory and SRAM cache memory. Moreover, to address permanent faults, spare rows and columns of memory cells are widely deployed in practice. These spare components, however, often come with high area overheads; the overheads would be more significant at future technology nodes where higher failure rates are expected to be. Researchers strive to reduce these overheads by leveraging cross-layer approaches. For instance, ArchShield [149] employs online testing to identify faulty cells in DRAM and expose the fault locations into the operating system. To this end, a portion of DRAM memory is reserved for storing both a fault map and replicated data. As a result, ArchShield eliminates the necessity of spare resources.

While faults can be detected and overcome by the aforementioned solutions, these cannot also bypass design bugs, which are unfortunately increasingly common in memory subsystems, as more heterogeneous processing units (microprocessor cores and accelerators) are integrated into the system. In particular, there are two critical functionalities that are prone to design bugs:

- Cache coherence: Memory subsystems should maintain data coherence among memory components (e.g., caches, scratchpad memory modules, and main memory modules).
- Memory consistency: Memory subsystems should guarantee the ordering among memory operations specified by the memory consistency model.

Validating cache coherence and memory consistency is a difficult, time-consuming task, because these features involve complex interactions among multiple processing units and memory modules. There are many different interactions among these components, some of which might generate invalid outcomes of memory operations with respect to coherence and consistency. Prior research on cache-coherence and memory-consistency validation can be classified into two groups: formal methods and testing-based methods.

(1) Formal methods: Mur-ϕ [79] is a model checker that is widely used to formally verify cache coherence protocols. It enumerates all possible states of a given cache-coherence protocol specification, and checks if the desired properties (e.g., a single writer and multiple readers) hold at each state. Recently, there is an increasing investigation of solutions that leverage formal methods for memory consistency verification. For instance, PipeCheck [131] proposes a microarchitectural happens-before graph that specifies ordering relationships among pipeline stages and among instructions. It then use the Coq theorem prover [6] to detect memory consistency violations. This theorem prover is also utilized by a couple of other works: CCICheck [139] to verify the interface between cache coherence and memory consistency, and Alglave [23] to verify several relaxed MCMs. Note that these Coq-based proposals use relatively small litmus tests, because the theorem prover must enumerate every possible memory-access interleaving for the given tests. Both Mur-ϕ and PipeCheck rely on abstract models of the designs under verification. These abstract models are described in domain-specific languages that are amenable to be fomally verified. Note that formal verification is conducted on the model descriptions (e.g., happens-before graphs), but not on the design descriptions used for implementation (e.g., hardware descriptions written in Verilog). Consequently, these formal methods may lead to incorrect verification outcomes: false positives or false negatives.

(2) Testing methods: TSOtool [96] is a memory-consistency validation tool that utilizes pseudo-random instruction tests. This tool has been widely deployed to validate several SPARC processors [202]. This tool creates relatively long multithreaded test-cases to exhibit various memory orderings at runtime. On the contrary, [134, 24, 25, 34] leverage litmus tests, each of which is a very small multithreaded test-case targeting a specific corner-case in the system's memory ordering behaviors. Each litmus test usually includes 2 to 4 threads, and each thread includes several instructions (less than 10 instructions). Litmus tests can be either carefully hand-crafted by verification engineers [34] or automatically generated by using formal methods [26, 133]. Note that both methods require significant knowledge about the system's memory consistency model. Instead of relying on such knowledge, [81] advocates to use a genetic algorithm (GA) to generate randomized test programs, then evolve them iteratively. In each iteration, test programs are morphed to target corner-cases. To achieve this goal, the genetic algorithm's crossover process is designed to diversify memory-access interleavings observed during test runs.

Another group of proposals [143, 63, 136] strives to dynamically validate memory ordering patterns at runtime or during post-silicon validation, using hardware monitors and checkers. Meixner and Sorin [143] propose to augment separate checkers for intra-thread ordering, inter-thread ordering, and cache coherence. Chen et al. [63] propose to utilize

a centralized graph checker that validates memory consistency and cache coherence as a whole. Mammo et al. [136] propose to modify cache structures to track the history of memory accesses. The logged memory accesses are then analyzed by a graph checking software.

Note that the self-checking mechanisms [198, 87, 129] presented in Section 2.1 are ineffective in validating memory consistency, because of the wide diversity of behaviors that may be observed, even in executions that fulfill a given memory consistency model. In Chapter 4, we strive to efficiently validate such breadth of behaviors.

Error recovery. As explained earlier, soft errors can be effectively detected and recovered by using ECC, and permanent faults can be patched with sparing cells or cell-remapping [149]. Besides these hardware-based solutions, there are software-based approaches that overcome faults in cache/memory structures. To this end, functional redundancy in the instruction set, as detailed in [87], can be exploited; for instance, if whole-word accesses are faulty, then each of these accesses can be translated into multiple partial-word accesses [142]. Moreover, the operating system can bypass faulty chunks of memory structures by not allocating them [149, 142].

Checkpoint-and-rollback mechanisms can be deployed to recover from faults at the architectural level. To this end, SafetyNet [186] records a global checkpoint across the entire multicore system, by coordinating local checkpoints of the system's individual components (register files, caches and memory controllers). These local checkpoints are recorded in small dedicated hardware modules. If a fault is detected, SafetyNet rolls back to a globally-consistent checkpoint. During recovery, it reconfigures the system if the fault turns out to be permanent. ReVive [165] leverages distributed parity protection, as well as a logging mechanism similar to SafetyNet. Notice that the checkpoint-and-rollback mechanisms of these two proposals are not designed to address design bugs.

To address design bugs, Caspar [199] advocates deploying small programmable logic blocks residing at each cache module in the system. The programmable logic blocks are configured to contain two pieces of information regarding a bug-prone cache-coherence transition: a cache coherence state and a transition condition. Once a bug-prone transition is detected, Caspar triggers a recovery process that allows only one in-flight memory access at a time. It also utilizes a checkpoint-and-rollback mechanism for recovery as in SafetyNet. Note that Caspar is a specialized solution to address incorrect coherence protocol implementations, while μMemPatch, a solution presented in this dissertation, addresses a wider range of memory subsystem bugs. However, Caspar shares a decompose-and-conquer trait with μMemPatch by sharing distributed distributed programmable logic blocks.

2.3 Addressing Errors in On-Chip Networks

On-chip communication infrastructures are becoming increasingly complex in systems on chip, as the number of components integrated into a monolithic chip grows steadily. Since the communication layer must serve many concurrent requests, its architecture has evolved from simple busses to full-fledged on-chip networks, which comprise tens of hundreds of discrete routers and links spanning across the entire chip. Among various options in network topologies, meshes have been widely investigated, because they provide very good scalability to large SoC designs. Thus, this dissertation proposes solutions targeting mesh-based topologies as well.

Error detection. To catch errors in on-chip networks, researchers have proposed various methods to validate the operations of on-chip networks at runtime.

As an example, ForEVeR [156] strives to formally verify three safety properties that must hold for on-chip networks: no data corruption, no flit drop, and no new flit generation. It specifies 83 property statements, which are verified using Synopsys Magellan, a commercial formal property verification tool. ForEVeR also deploys a checker logic to verify network-level properties at runtime. Note that the verification of network-level properties needs to examine the entire network – usually, formal verification tools do not scale well to this large design size.

In contrast, NoCAlert [164] does not rely on design-time guarantees of correctness, and uses hardware assertions to detect fault occurrences. The assertions check four types of invariants: bounded delivery, no data corruption (no packet mixing), no flit drop, and no new flit generation – these invariants are similar to the safety properties checked by ForEVeR, and are checked across 32 assertions embedded in microarchitectural blocks: routing computation units, virtual channel and switch allocation arbiters, crossbars, buffers, input and output ports.

SafeNoC [18] also specializes on runtime validation. In particular, it checks errors when packets arrive at their destinations (i.e., end-to-end checking). To this end, it leverages a small checker network, alongside the regular network, that delivers small signatures of the data packets. At the destination, each signature is then compared with its corresponding data packet for integrity checking. SafeNoC also deploys a timeout scheme to detect any dropped packet.

Finally, Park et al. [158] propose to verify transferred data at every hop and every cycle. Thus, any wrong data can be instantly detected and corrected by leveraging a small extra buffer embedded in each port. When an error is detected, the flit stored in the buffer is

retransmitted. This same buffer is also used to recover from deadlocks. Note that, however, this solution is tailored to address soft errors – it is unclear how it can be repurposed to address design bugs or permanent faults.

Another group of solutions, including [93, 125, 84, 90], focuses on error diagnosis in on-chip networks. Several solutions of them [93, 125, 84] employ dedicated testing hardware blocks that can generate test patterns. Each generated test pattern can detect a specific fault, e.g., stuck-at-0/1 or crosstalk, in either router or link. Since the testing scheme is time-consuming, because it must examine individual routers and links. [93] uses multi-cast messages; these messages have deterministic routes that are scheduled offline. [90] shares the same idea as [93], but it can verify the correctness of adaptive routing, which does not follow deterministic routes.

Error recovery. Researchers have shown that spare resources can effectively overcome permanent faults in on-chip networks. In [61], spare routers are placed at the top row of a mesh network. Unlike a conventional mesh network where each processing unit has a fixed connection to one router, the work proposed in [61] offers flexible connections between routers and processing units – each processing unit can utilize one of its adjacent routers. When a fault is detected, the network initiates a reconfiguration process to change the connections between routers and processing units, as well as the wire connections between routers. Lehtonen et al. [125] advocate deploying spare links to replace faulty links. In their proposal, the exact locations of faulty links can be identified during detection and diagnosis, and spare links can replace any regular links. While offering little performance degradations upon fault occurrences, these sparing solutions often entail significant area and power overheads, and once the spares are used up, no more faults can be tolerated.

As an alternative to spare-based solutions, *routing reconfiguration* is a cost-effective mechanism to tolerate faults. Many routing-reconfiguration solutions are fundamentally similar to each other; they strive to discover fault-free routes for all source-destination pairs in the network. In doing so, it is often preferred to consider only deadlock-free routes.[1] As shown in [72], the network can be deadlock-free if there is no cyclic resource dependency among packets in transit.

The turn model [91] offers a very efficient way to ensure deadlock-freedom in mesh networks. In this model, every router enforces a designated turn[2] in the router to be prohib-

[1]Once a network is deadlocked due to conflicting packets, the network must be recovered to a deadlock-free state before advancing execution, which often requires additional hardware resources.

[2]The original turn model [91] defines a turn as a directional change in mesh networks (e.g., from x direction to y direction or vice versa). But this concept has been generalized later in [157] to indicate a progression from an input link to any output link at a router, regardless of a directional change.

ited. Wu [205] improves the turn model to enhance the ability of the network to tolerate a few faults. While these turn models involve only very simple hardware resources for route computation, they do not provide significant tolerance to faults, as also discussed in [169].

Similarly, researchers have investigated local routing reconfiguration based on bypass rules to reroute around faults (i.e., faulty routers or links) using local connectivity information [208, 80]. In these solutions, each router monitors the availability of its neighboring components. When the router discovers that a neighboring component has become unavailable, it avoids transmitting towards that direction. While solving the issue of preventing packet loss, these solutions may fall into livelock situations (i.e., a packet keeps moving with no progress toward its destination), due to the short-sightedness of their detouring approach. [208] provides a bypass rule against a faulty router, which utilizes deadlock-free detour paths surrounding the router. [80] employs an adaptive selection between the X and Y directions to bypass a faulty link. Due to the localized nature of these solutions, they also can only sustain a few faults before the network becomes disconnected.

To maximize fault tolerance, turn models have been expanded to allow arbitrary prohibition of turns [171, 22, 157]. uLBDR [171] presents a hardware-efficient methodology to place turn restrictions, performing route computation only with a handful of flip-flops embedded in each router. Ariadne [22] achieves more flexible routing than uLBDR, at the cost of extra hardware overheads to deploy routing tables, instead of a few flip-flops. The routing tables are re-computed upon each fault occurrence, finding routes that obey turn restrictions placed in the network. Unfortunately, this reconfiguration process takes thousands of clock cycles in a typical 8×8 mesh network: indeed, its time complexity is $O(n^2)$, where n is the number of nodes. Unlike bidirectional-turn restrictions imposed by uLBDR and Ariadne, uDIREC [157] leverages uni-directional turn restrictions to improve fault tolerance and minimize performance impacts, at the expense of additional route-computation efforts.

There are also routing reconfiguration proposals that do not rely on turn restrictions for deadlock-freedom. Immunet [166] employs a small route table for fault-ridden situations; these small route tables provide a ring path among surviving network components, and are reconfigured after each fault occurrence. Note that a similar ring topology is also used in a checker network in SafeNoC [18].

Another class of route reconfiguration relies on a search process of fault-free routes [68, 197]. In [68], upon a fault occurrence, every node in the network sends route-search packets to its neighboring nodes. When the neighbors receive a route-search packet, they send an acknowledge packet to the source node, and then forward the received route-search packet to their neighbors. Later, the source node analyzes all the acknowledge packets it has

received, then builds an array of fault-free routes. Similarly, [197] proposes a backtracking-based route-search process; this proposal constructs a list of forwarding directions for each source-destination pair. Finally, [162] leverages stochastic communication to select the forwarding direction for each packet. While it can provide high fault tolerance at low silicon cost, it also suffers from a high performance impact, due to the use of non-minimal paths and the lack of delivery-guarantees.

2.4 Addressing Errors in Accelerators

Research proposals have focused on major accelerators, such as graphics processing units (GPUs) and neural processors. These major accelerators have been widely investigated with respect to their reliability against faults. Solutions for generic accelerators have been proposed as well, proposing to utilize high-level accelerator models to verify low-level accelerator designs. Note that these prior solutions focus on verification of individual accelerators, while our proposal, AGARSoC (see Chapter 8, focuses on verifying interactions among multiple accelerators.

Error detection. Among various types of accelerators, GPUs have been heavily investigated with respect to their reliability. Jeon and Annavaram [111] propose a lightweight error-detection solution, called Warped-DMR, by leveraging opportunistic spatially- and temporally-redundant execution of code. They claim that many GPGPU applications utilize only a few threads, leaving the rest threads under-utilized. Warped-DMR exploits these under-utilized GPU threads to verify computation results by utilizing (1) inactive threads within a warp (spatial, intra-warp DMR) and (2) replayed computation (temporal, inter-warp DMR). Sabena et al. [175] take a similar approach using duplicated computations, but they propose to partition the GPU's streaming multiprocessors into two groups: a regular computation group and a checker group, where the checker group performs the same computation as the regular group. They experimentally compare error-detection rates of the two different redundancy methods, a spatial-based method and a temporal-based method, and found that the latter results in a higher detection rate. Tselonis et al. [194] focus on the impacts of permanent faults in GPU's register files. They found that many permanent faults are unnoticeable; they either do not alter the correctness, or incur minimal performance degradation. Li et al. [128] propose a dynamic reliability management framework for GPU applications. The proposed framework injects additional software reliability enhancement code, which is designed to catch soft errors at runtime, into GPU applications using a just-in-time compilation environment.

The impacts of soft errors in deep neural networks has been explored in [126]. This work investigates how soft errors propagate through the outcome of neural networks. Specifically, it evaluates the impacts of a few parameters with respect to error propagation, such as the network topology and the bit position. They find that a symptom-based detector can catch anomalies using input-value bounds, where symptoms are learned using representative test inputs. This work also demonstrates that soft errors could be catastrophic in mission-critical neural networks such as those in self-driving cars.

Campbell et al. [58] propose a quick error-detection methodology for general accelerators using high-level synthesis tools. They propose to augment the synthesized hardware accelerator with a signature generation module that probes the accelerator's registers at runtime. The proposed method collects signatures from (1) simulations of the high-level model and (2) executions of the hardware accelerator, then compare the signatures between the two. The proposed method can detect both logic and electrical errors.

Error recovery. For GPUs, Tselonis and Gizopoulos [195] present a reliability evaluation framework, which is evaluated on NVIDIA's GPUs. Based on their findings, they propose to protect a handful critical components in the GPUs to attain the best protection at low cost: register files, shared memory, SIMT stacks, and instruction buffers.

A few prior works focus on neural network accelerators. Temam [191] presents a fault-tolerant neural network accelerator that employs spatially-expanded neurons. He found that spatially-expanded neurons are beneficial for both energy efficiency and fault tolerance, compared to a conventional neural network composed of time-multiplexed neurons. He also investigates the impacts of permanent faults at the transistor level, that is, how the transistor failures translate to incorrect computation outcomes.

A more recent work by Li et al. [126] focuses on protecting against soft errors on deep neural network accelerators. They show that selective latch hardening can greatly reduce failure rates while incurring moderate area overheads. Note that similar hardening techniques have been widely deployed in microprocessor cores.

Zhang et al. [207] present a fault-tolerant architecture for systolic array-based architecture like Google Tensor Processing Unit (TPU) [113]. They empirically show that permanent faults can significantly reduce computation accuracy. To mitigate this degradation, they propose to bypass faulty array elements and re-train the network accordingly.

Note that the aforementioned solutions mostly rely on conventional techniques that are widely used in other domains (i.e., microprocessor cores and memory subsystems): hardening registers and latches, adding ECC mechanisms to memory components, and bypassing faulty components. Moving forward, we briefly discuss research opportunities related to

more efficient error-recovery solutions for accelerators as a future research direction in Chapter 9.

The next chapter begins the discussion of our solutions addressing errors in systems on chip. Specifically, Chapter 3 addresses the challenge of verification of microprocessor cores.

CHAPTER 3

Probabilistic Bug-Masking Analysis in Instruction Tests

In this chapter, we discuss evasive design bugs in microprocessor cores and how these bugs can be addressed efficiently in post-silicon verification. Microprocessor cores have been evolving to meet ever-growing computation requirements. In particular, a noticeable evolution is happening in instruction sets that microprocessors offer; high-performance microprocessors implement hundreds of advanced instructions to efficiently process complex applications. Moreover, microprocessor cores implement full-fledged features for high performance and power efficiency, such as speculative executions and power saving modes. Indeed, each execution of a test may result in microarchitectural activities that have not been observed during other executions of the same test, depending on the microarchitectural state when the test is run. Post-silicon instruction tests hence strive to verify functional corner-cases that originate from a combination of complex instruction interactions under various microarchitectural states.

Post-silicon microprocessor verification uses constrained-random instruction tests that are tailored to stimulate rarely-occurring, unexpected functional corner-cases. The goal is to run as many constrained-random tests as possible within the time allowed for verification, so to improve the chance of discovering bugs. To this end, post-silicon verification leverages lightweight result-checking mechanisms that verify only the architectural state after each test is completed. Note that such lightweight, end-of-test checking is beneficial especially in post-silicon verification, where observing intermediate results comes at a great cost (e.g., hardware trace mechanisms, software instrumentations). However, this end-of-test checking becomes problematic for long tests that consist of hundreds or thousands of instructions. When bugs manifest themselves in the middle of a test execution, subsequent instructions may purge the impact (i.e., erroneous results) caused by the bug. Such bug-masking incidents are very undesirable, but likely to happen in instruction-based tests. If test results could be checked immediately after a bug manifests, the bug would have been exposed.

This chapter focuses on how to analyze the possibilities of bug-masking incidents in instruction tests. To this end, a conventional bug-masking analysis evaluates such possibilities for a given test program. It relies on many bug-injection simulations, each of which simulates how a bug-impacted system behaves for the given test program. However, this conventional dynamic analysis is extremely costly, as it requires many dynamic simulations to evaluate possible bug scenarios. We continue our discussion on this dynamic analysis in Section 3.1.1.

To minimize the computation requirements for bug-masking analysis, we propose a solution that *decomposes the analysis problem of considering an entire test program to smaller components: the individual instructions*. We then address these smaller problems by computing bug-masking probabilities at the instruction-level, a much more straightforward challenge, which can be solved even analytically (Section 3.4.1). Once we have solved the instruction-level problems, our proposed solution, called BugMAPI, can tackle the original complex problem by *combining the bug-masking probabilities through an analysis of the information-flow paths* (i.e., use-def chains in data-flow analysis) of the test program.

The BugMAPI solution, which stands for **Bug**-**M**asking **A**nalysis with **P**robabilistic **I**nformation-flow, is based on static information-flow analysis with the following improvements:

1. We compute and utilize the bug-masking probability for each instruction type.
2. We develop a heuristic to derive the bug-masking exposure of an instruction sequence from the individual instruction probabilities.

3.1 Background and Motivation

Post-silicon verification and validation. High-end microprocessors often include complex features that are hard to verify in the early stages of verification. Therefore, post-silicon validation, that is, the validation effort carried out on the first silicon prototypes, aims at catching all the remaining bugs that were not detected in the pre-silicon verification stage [148]. One of the most difficult aspects in the deployment of post-silicon validation, however, is the extremely limited observability into the design's internals, as we discussed in Section 1.2. This aspect makes bug detection and diagnosis challenges of their own. Even more importantly, bugs may manifest during a test's execution, but become masked and go unnoticed by the time the test completes. The outcome in these situations is that the post-silicon test simply wastes precious prototype's execution cycles. To overcome these

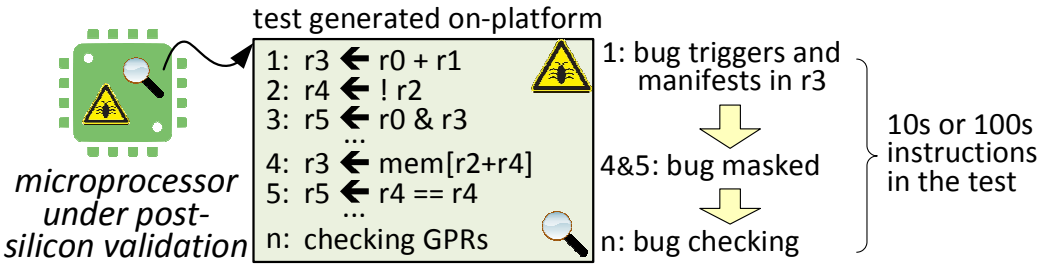


Figure 3.1: The bug-masking problem in post-silicon random instruction tests. In this example code, the results of instructions 1 and 3 are overwritten by instructions 4 and 5 so that any bug occurring or propagating through the former two instructions would not be detected at the end of the test.

challenges, two groups of solutions were presented in Section 2.1. The first group [159, 76] leverages dedicated hardware resources to trace microarchitectural activities at runtime. The second group [198, 87, 129] advocate software-based, self-checking solutions. Unlike these solutions, BugMAPI requires neither hardware resources nor self-checkable software. Rather, it is a purely software-based analysis that strives to accurately compute the possibility of bug-masking incidents in instruction tests, due to limited observability.

Test programs can be tailored with sophisticated generation algorithms (i.e., directed tests). Or, they can be quickly generated in a constrained-random manner (i.e., random tests) where instructions are randomly chosen within instruction candidates guided by simple test-generation algorithms (e.g., those using test templates). This latter type of test generation is more important in post-silicon verification, as most directed tests have been already verified in pre-silicon verification. It is also particularly valuable in trying to exercise unidentified corner-case scenarios, because a vast number of variants can be generated with little designer's effort [146, 192, 19]. One additional benefit of random tests is that they can often be generated directly on the microprocessor under verification (i.e., on-platform generation), to efficiently use scarce post-silicon validation platforms available.

Bug masking occurs when, *due to the limited observability in post-silicon validation platforms, a bug becomes undetectable after some amount of computation has occurred past its manifestation*. Indeed, in post-silicon validation, it is often possible to only monitor architectural registers and memory, usually only at the end of the test execution, while in pre-silicon validation, more detailed information is available (e.g., cycle-based microarchitectural state). An example of this problem is illustrated in Figure 3.1, where the first instruction triggers a silicon bug, leading to an incorrect value being written in register `r3`. However, the erroneous value in `r3` is subsequently overwritten, and thus undetectable by the time the test completes.

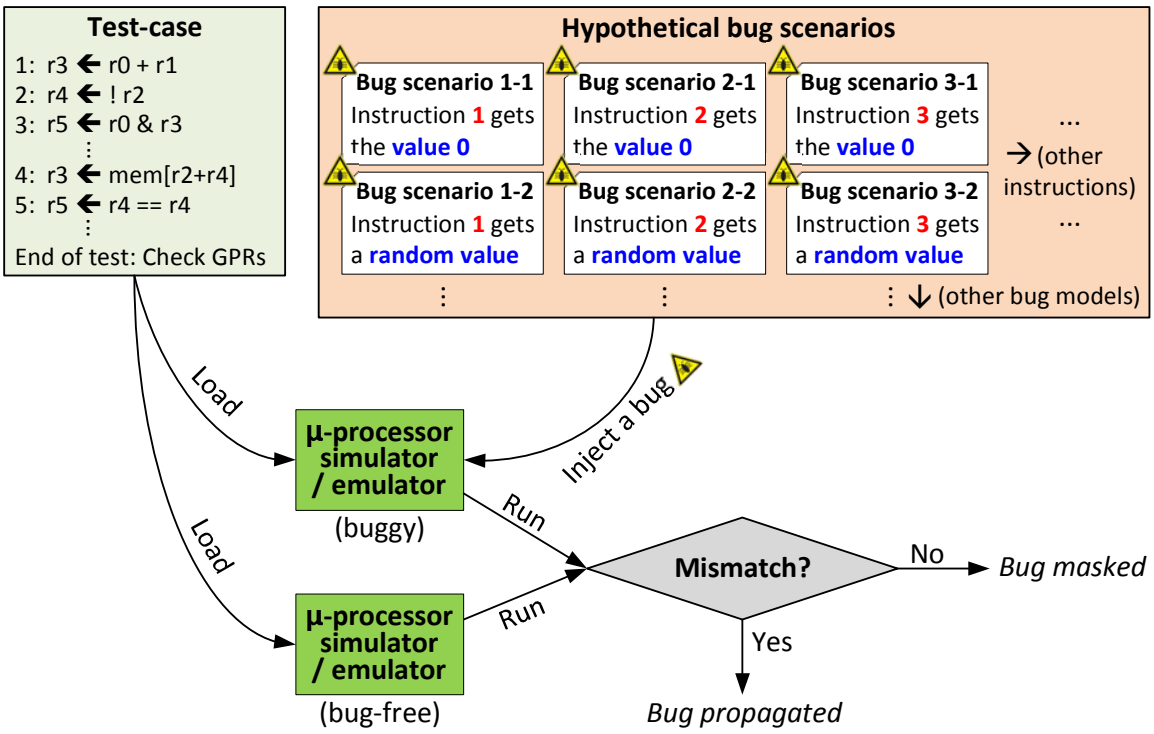


Figure 3.2: Dynamic bug-masking analysis flow. Each of the hypothetical bugs is injected at runtime during the simulation of the test-case. At the end of the test-case, the results of the test-case (e.g., general-purpose register values) are checked with the bug-free simulation results.

During the post-silicon validation process, identifying situations that may lead to bug-masking provides great benefits in improving the quality of the validation effort. For instance, as we demonstrate later in Section 3.6, it allows us to guide the application of small modifications to test programs so that we can minimize bug-masking incidents thus attain higher coverage from a regression suite. Other applications of a bug-masking evaluation include assessing the quality of a test: tests to be included in key regressions can be selected based on having a low bug-masking likelihood. In addition, in test generation, test-templates can be selected and designed to limit the generation of sequences that are susceptible to masking.

3.1.1 Dynamic Bug-Masking Analysis

In order to accurately identify bug-masking incidents that may happen during the execution of a test-case, we can perform an exhaustive dynamic analysis following the two steps below.

Step 1: Identifying bug scenarios. We identify all possible bugs that are likely to be found during the executions of test-cases. Note that we strive to catch *unexpected* bugs while running test-cases; if bugs were known *a priori*, we would fix the bugs without relying on test-cases. In other words, we should take into account all such bug possibilities in this step. To this end, we use hypothetical bug scenarios, as illustrated in Figure 3.2. As illustrated in the figure, a bug may alter the result of instruction 1, overwriting the result with value 0 (bug scenario 1-1). Or, the same instruction may spuriously get a random value (bug scenario 1-2), etc.

To obtain real-world insights on how typical post-silicon bugs manifest themselves, we discussed with research engineers working at a post-silicon verification team at IBM Research – Haifa. From their experiences, buggy result values exhibit certain patterns (i.e., they are not truly random). See [62] for more details. Based on their suggestions, in developing this project, we made the following assumptions on bugs:

- A bug manifests as an incorrect architectural state.
- A bug affects only one instruction in the test-case.

Our bug model is simplistic, as we do not take into account microarchitectural structures. Microarchitectural bug modeling would capture more realistic bugs at the expense of much more bug scenarios to be examined. We further discuss our bug model we evaluated in Section 3.5.1.

Step 2: Bug-injection simulation or emulation. For each bug scenario, the microprocessor under verification must be simulated or emulated to see if any buggy result can be observed at the end of the test. To this end, the microprocessor is executed twice: a buggy execution (i.e., a test-run with a bug injected) and a bug-free execution (i.e., a test-run with no bug injected). The results of the two executions are then compared to find any mismatch, as shown in Figure 3.2.

Note that each simulation takes a significant amount of time. A typical dynamic analysis framework requires a separate design simulation or emulation for each bug scenario. Thus, the number of executions for a given test-case is multiplied by the number of bug scenarios to be simulated. Also, as per our bug model, the number of bug scenarios is proportional to the number of instructions and the number of bug types. These multiplicative factors (i.e., hundreds or thousands) are usually much larger than a typical repetition count of a test-case (i.e., tens).[1] In other words, the efforts of evaluating possibly-masked bugs

[1] In this chapter, we only consider test-cases that generate deterministic outcomes. In other words, every

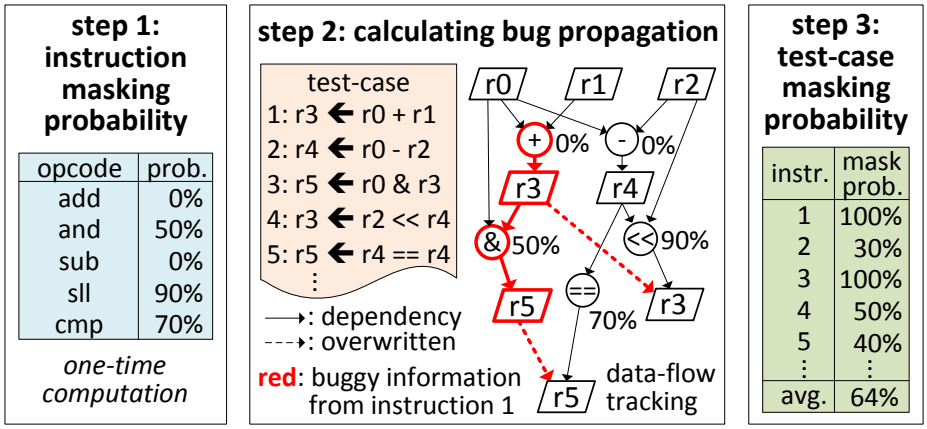


Figure 3.3: Overview of BugMAPI probabilistic bug-masking analysis. BugMAPI applies static instruction-flow analysis to instruction test sequences by leveraging accurate, pre-computed masking probabilities for each instruction, and by deriving overall test-sequence masking rates using information-flow tracking.

outweigh actual bug-hunting efforts. In summary, this dynamic analysis can be prohibitive, because of the analysis's high computation involved in the bug-injection campaign.

3.2 BugMAPI Overview

BugMAPI applies static code analysis to instruction test sequences. The results of the analysis report which instructions in the sequence are likely to mask potential bugs occurring during their execution. To this end, we first introduce a basic approach in Section 3.3; we then improve on it in Section 3.4 by taking into account the type of instruction in the analysis. In computing the bug-masking probability of a test-case, we leverage a novel decomposition technique to break the probability computation into the fine-grained instruction-level computation, then use the data-flow analysis to compute the bug-masking probability of the test-case.

Figure 3.3 outlines BugMAPI. Our solution considers a test sequence and applies three steps to it: the first step assigns a bug-masking probability to each instruction in the test sequence, based on its functionality (Section 3.4.1). The second step tracks data dependencies between instructions and computes the probability that data affected by a bug propagates to the end of the test, thus remaining observable by end-of-test checkers. It also leverages an approximation technique to reduce the computational complexity of this step

iteration of a test-case should generate the same results under bug-free circumstances. In these test-cases, each test-case is repeatedly executed tens of times under different initial microarchitectural states. In Chapter 4, we consider non-deterministic outcomes in multithreaded test-cases due to memory-access interleavings. These test-cases use much higher iteration counts than deterministic test-cases.

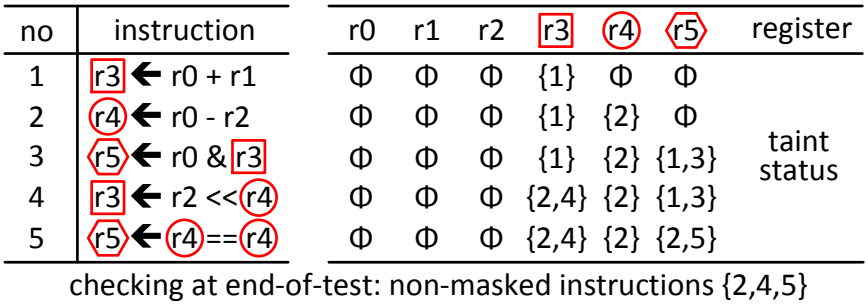


Figure 3.4: Baseline information-flow analysis. Our baseline analysis propagates information flow leveraging instructions' inputs and outputs. In the best case, checkers at the end of the test could reveal any bug that occurred at instructions 2, 4, or 5. Bugs manifesting at instructions 1 or 3 are masked.

(Section 3.4.2). The last step derives the bug-masking probability of the overall test, from those computed for the individual instructions.

Our endeavor focuses on improving the accuracy of a typical static analysis, by leveraging an accurate masking-probability computation for each instruction. Note that, to be efficient, BugMAPI does not take into account concrete values (e.g., operand values in arithmetic instructions) in determining if a bug is masked by an instruction, while dynamic analysis accounts for concrete values. Overall, our static approach reduces computation requirements by three orders of magnitude, compared to a simulation-based dynamic analysis. This benefit comes at the cost of lower accuracy, due to the lack of dynamic information.

3.3 Baseline Static Information-Flow Bug-Masking Analysis

We first developed a simple information-flow tracking technique that determines the observability of buggy values in registers and memory, which we refer to as our *baseline analysis*. This analysis is static, in contrast to popular taint-tracking solutions in the security domain [77, 154]. Our choice is driven by the need of post-silicon validation environments to be efficient and keep a low-computation profile, since the execution of the test itself is extremely fast.

Figure 3.4 illustrates an example of the baseline analysis, applied to a 5-instruction sequence. For each instruction, we extract inputs and outputs as per the ISA specification. Then, for each resource in the test program (e.g., registers), we proceed instruction-by-instruction and compute which instructions can propagate their taint status to it. In other words, instruction X propagates the taint status to register Y if a bug manifesting in instruction X could affect the value of register Y [77]. For instance, after the first instruction, `r3`

has a taint list containing instruction 1, while all other registers have empty lists. Note that the third instruction uses the `r3` value computed in the first instruction and writes to register `r5` so that the taint status of `r5` becomes $\{1, 3\}$. At the end of the analysis (last line), the taint status lists indicate which instructions' results impact the status of each register at the end of the test program. Thus, by computing the union of all the taint lists, we find which instructions could expose bugs that are not masked by the program.

Note that inaccuracies are possible because this baseline analysis does not take into account the distinct operation of each instruction. For instance, with reference to Figure 3.4, at the end of the program, `r3` should reveal bugs occurring during the execution of the second instruction. However, if `r2` = 0, any erroneous value stored in `r4` would be masked in the fourth instruction, clearing the taint in `r3`. Consequently, our baseline analysis provides conservative bug-masking results and false-negatives (i.e., bugs predicted as detectable, but masked in reality) may indeed occur. In the next section, we try to overcome precisely this limitation.

3.4 Probabilistic Information-Flow Bug-Masking Analysis

BugMAPI refines the baseline analysis from the previous section by taking into account the instruction type in computing bug-masking probabilities. As an example, Figure 3.5 shows this refinement for three instructions from the Power ISA [98] (`add`, `and` and `sld`). For the sake of this example, let's assume that bugs manifest as single-bit flips in a source register (Sections 3.5.1 and 3.5.5 discuss the actual bug-models used in our evaluation). The goal here is to compute the probability that a single-bit flip in input registers RA, RB or RS propagates to the output register value. In the case of `add`, the bit flip propagates in all cases, except for overflow, so masking probability is practically 0%. In the case of an `and` instruction (middle of Figure 3.5), there is a 50% chance that the bit flip is masked by a 0-value in the corresponding bit position of the other operand. For `sld` (shift left doubleword), we computed that the bugs in the shift amount RB are masked 91% of the execution times (because only the least significant bits are used by the instruction), while the source register RS is masked 49% of the times. Note that the baseline analysis is equivalent to assigning a 0% bug-masking probability to all the input-output flow paths, hence it is clear that the baseline analysis would lead to a much higher false-negative rate.

BugMAPI computes the bug-masking probabilities for each instruction of a processor's ISA using the process discussed below in Section 3.4.1. This computation is only carried out once per ISA. It then considers the test-case under analysis, applies the individual probabilities to each instruction and computes how they propagate to the end of the test

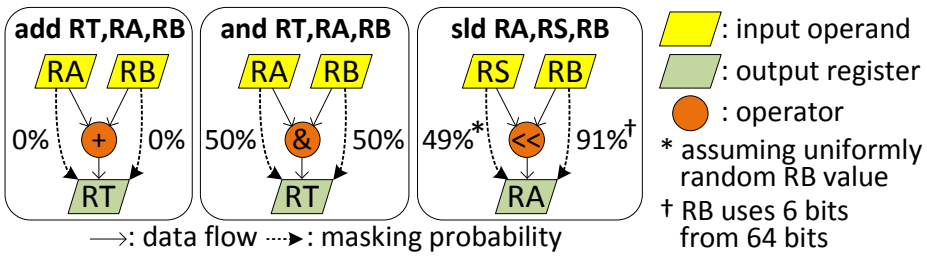


Figure 3.5: Examples of input-output bug-masking probabilities.

program, similarly to our baseline analysis (details are in Section 3.4.2). Finally it can compute the masking probability of the entire program, for instance, as an average masking rate over the individual contributions of the instructions in the program. Note that the last step can be tailored to the specific validation goal. For example, when computing the average, the contribution of specific instructions can be weighted to tilt the criticality of masking to certain units of the processor (e.g., floating point), or certain high-interest program constructs.

3.4.1 Bug-Masking through an Instruction

When computing the masking probability of an instruction, we consider each input and output (i/o) pair and compute the probability independently for each pair. Note that it is possible that the probabilities through multiple i/o pairs are correlated. For instance, the addo instruction in Power ISA sets the overflow flag when appropriate. Thus a single buggy input operand may affect both the target register and the overflow flag. However, to keep our computation manageable, we disregard these second-order effects.

We developed two approaches to compute masking probabilities of instructions. The first is an analytical approach, where we investigate the ISA specification (e.g., [98]) to understand the specific functionality of each instruction, and compute the masking probability for each i/o pair mathematically. We used this model mostly for branch and load/store instructions.

The second is an experimental approach whereby we execute the instruction in an instruction-set simulator (e.g., gem5 [56]) with 1,000 sets of random input values. For each i/o pair, we run each input set twice, one with the random input generated and the other with a buggy version of the input set, and then we compare the output to determine if the bug was masked. Based on the overall findings, we can derive the masking probability with high confidence. To attain efficiency in this process, we develop assembly programs that encapsulate the input generation, the repeated executions and the probability computation, all together. Developing those programs is the only part of this approach that requires

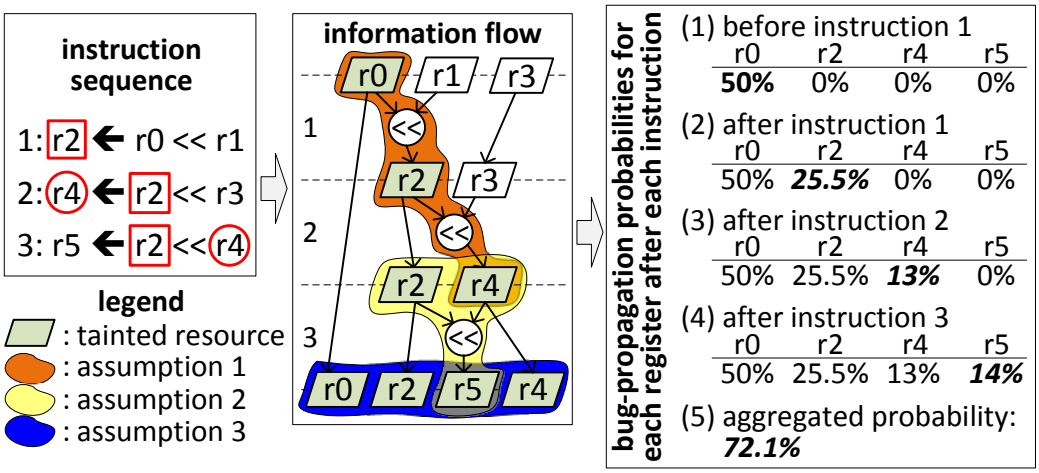


Figure 3.6: Example of bug-propagation analysis over an instruction sequence. Initially, only `r0` carries a buggy value with a 50% probability. Through the execution of instructions 1–3, the buggy value may propagate also to `r2`, `r4` and `r5`. We derive the propagation probabilities on the right table by leveraging our three simplifying assumptions.

engineering effort.

Both solutions have their own advantages. The experimental approach can be more accurate, and it can be easily adapted to a new bug model. The analytical one is beneficial when developing the assembly program is too time-consuming, and for instructions whose masking probabilities tend to be unaffected by the specifics of the bug model.

3.4.2 Bug-Masking over an Instruction Sequence

The goal of this step is to compute, for each instruction in a sequence, the likelihood that a bug manifesting at the instruction propagates to the end of the sequence, revealing an incorrect test outcome. Computing this likelihood accurately is computationally intensive: a simulation-based dynamic analysis would have to consider a vast number of executions to properly randomize all the input sources for all the instructions in the sequence. A static analysis could be very involved because of how a buggy instruction can have a large fanout, impacting many other registers and resources over the test execution, all of which would have to be tracked to determine if they eventually become masked.

Because of the high cost of an accurate analysis, we chose to design a high-quality approximation for it. The calculation makes three simplifying assumptions, which we discuss below and illustrate with an example in Figure 3.6. Note that for sake of simplicity, we perform the analysis using bug-propagation probabilities, which are the complement of bug-masking ones.

Assumption 1 — *Instructions are independent among each other.* As a result, we can

calculate the bug-detection probability through multiple instructions, by simply calculating the product of each instruction's probability of propagating an erroneous value. As shown in Figure 3.6 (dark orange region), a bug in `r0` propagates to `r4` through instructions 1 and 2. Consequently, the probability that a bug manifests in `r4` through `r0` is the product of the probability of having a bug in `r0` before the program starts, times the two instructions' bug-propagation probability.

Assumption 2 — *Inputs of an instruction are independent among each other.* Because of this assumption, we derive the bug propagation likelihood to an output as the complement of the probability that both i/o pairs are masked. With reference to the example in Figure 3.6 (yellow area), the probability that a bug propagates from either `r2` or `r4` to `r5` is: $P((r2 \rightarrow r5) \cup (r4 \rightarrow r5)) = 1 - P((r2 \rightarrow r5)^c \cap (r4 \rightarrow r5)^c) = 1 - (1 - P(r2 \rightarrow r5)) \times (1 - P(r4 \rightarrow r5)) = 1 - [1 - P(r2) \times P(RA \rightarrow RT)] \times [1 - P(r4) \times P(RB \rightarrow RT)] = 1 - (1 - 0.255 \times 0.51) \times (1 - 0.13 \times 0.09) = 14\%$.

Assumption 3 — *Resources are independent among each other.* This assumption is useful in calculating the final bug-propagation probability from an instruction through an entire sequence. It allows us to compute the overall bug-propagation probability from the probability that the bug had propagated to any of the monitored resources (e.g., registers). With reference to the example, in the blue region of the figure, we indicate that the final resources available are `r0`, `r2`, `r4` and `r5`. `r0` is included because it is the only register that propagated the bug until before instruction 1. By the end of the code snippet, bugs in `r0` may be reflected in erroneous values in `r0` or any of the other registers in the blue area. Using the independence assumption, and a similar calculation as in the previous paragraph, the probability that the bug manifests at the end of the sequence is: $P(r0 \cup r2 \cup r4 \cup r5) = 1 - P(r0^c \cap r2^c \cap r4^c \cap r5^c) = 1 - (1 - P(r0)) \times (1 - P(r2)) \times (1 - P(r4)) \times (1 - P(r5)) = 1 - (1 - 0.5) \times (1 - 0.255) \times (1 - 0.13) \times (1 - 0.14) = 72.1\%$.

3.5 Experimental Evaluation

In this section, we first discuss our bug models (Section 3.5.1) and computational cost (Section 3.5.2). We then present BugMAPI's characterization (Section 3.5.3) and accuracy (Section 3.5.4) by comparing against a dynamic analysis, and conclude with a brief discussion (Section 3.5.5).

BugMAPI is implemented in Python, and it is evaluated with test-cases generated for two ISAs: IBM Power and DEC Alpha. For IBM Power, we generated bug-masking probabilities for approximately 4,000 i/o pairs corresponding to all instructions for the POWER8 processor. Note that our implementation for Power does not accurately take into account

memory address disambiguation, and it assumes that store operations have no overlapping addresses. For DEC Alpha, however, we did implement a simple disambiguation mechanism by dedicating a pool of registers only to load/store base addresses. We only included a subset of 118 instructions and 218 i/o pairs in our Alpha ISA evaluation.

We used Threadmill [19] to generate IBM Power tests, and a simple in-house test-generator for Alpha ISA tests. Note that BugMAPI is not yet capable of handling multi-threaded tests, analyzing only one thread at a time. We discuss multi-thread related issues in multi-threaded tests in Section 3.5.5.

3.5.1 Bug Models

The goal of our bug model is to capture many corner-case functional bugs, representative of those often detected in post-silicon validation. To this end, we inject bugs that *slightly alter the architectural state* (i.e., registers and memory locations). Note that this model mimics microarchitectural bugs that ultimately modify the architectural state. For instance, a malfunctioning cache will be eventually revealed when its incorrect data is read by a load operation. However, purely microarchitectural bugs that lead to execution delays or bugs that are usually detected by system's hangs cannot be detected by BugMAPI and thus our models are not concerned with capturing them.

In our reference dynamic analysis, we inject bugs by modifying a register value, or a memory location, right after it has been updated. We considered five types of modifications that could be forced by the bug, each corresponding to a distinct bug model, as we discuss in Section 3.5.5. After a preliminary analysis, we settled for the *random-value* option because we believe that it resembles more closely what occurs in practice. With this model, a register value or a memory location affected by a bug is replaced by a random value. Note that when injecting bugs into memory locations, we are careful to generate a value matching the original bit-width. For instructions that trigger multiple updates, we only inject a bug in one output value at a time.

3.5.2 Performance Analysis

We evaluated BugMAPI's execution time on a dual Intel Xeon system. To analyze 100 instances of the longest Alpha test (the last bar of Figure 3.8), BugMAPI took approximately 11.3 seconds. The dynamic analysis for the same tests took 61 minutes of ISS simulations using 7 cores (one simulation per each bug injection, running one core per simulation). From these values, we estimate that BugMAPI provides a *three-orders-of-magnitude speedup* over a dynamic analysis. In addition, the probability computation

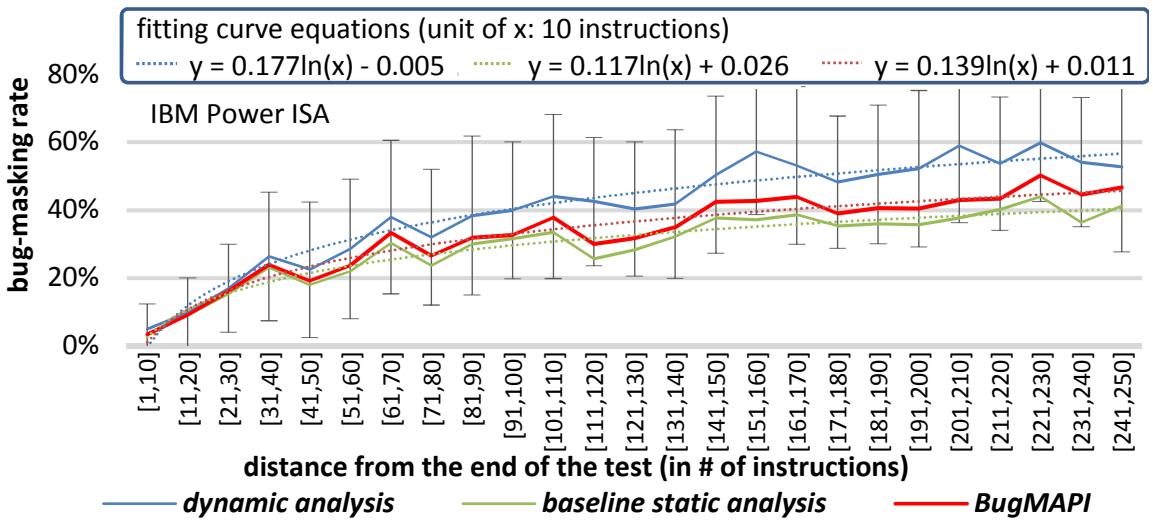


Figure 3.7: Bug-masking rate for individual program instructions, sorted by increasing distance from the end of the test program. Each data point is obtained by averaging over all instructions in the distance window and over 30 distinct randomly-generated tests.

described in Section 3.4.1 took approximately 2 minutes for the subset of instructions of the Alpha ISA and less than one hour for all the instructions of the POWER8 ISA. Note that this step is only required once per ISA, and is easily absorbed over the large number of tests evaluated.

3.5.3 Bug-Masking Characterization

In Figure 3.7, we plot the bug-masking rate of an instruction as a function of its distance from the end of the test program for IBM Power ISA. This evaluation was carried out to gather a sense of how much program length affects masking probabilities, and to place the accuracy of BugMAPI with respect to a dynamic analysis and a baseline static analysis. Each data point is obtained as the average of the bug-masking rate over all instructions in the windows indicated: for example, the first data point on the left averages the last 10 instructions of the test, while the rightmost one averages the instructions that are between 241 and 250 instructions before the end of the test. Moreover, each data point is obtained by averaging over 30 distinct test programs.

The diagram compares three solutions: a dynamic analysis (where bugs are detected by comparing against an equivalent execution with no bug injection), the static baseline analysis, and BugMAPI. As expected, the bug-masking rate is higher for instructions further away from the end of the test where the checking occurs. However, it is encouraging to notice the quantitative values of this trend: even after 200 instructions, the average bug has only a masking probability of 40–50%, based on BugMAPI's and the dynamic simula-

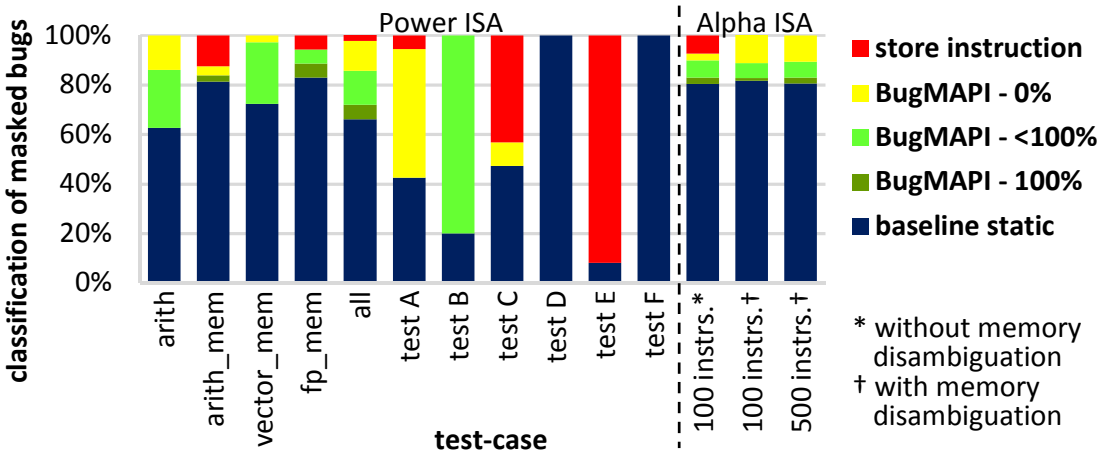


Figure 3.8: Classification of masked bugs. Masked bugs are classified based on the ability of our baseline analysis and BugMAPI to identify the masking.

tion's estimates. Note that BugMAPI's analysis is only 21% less accurate (by comparing fitting curve equations) than the dynamic one, while achieving a three-orders-of-magnitude speedup.

3.5.4 Accuracy

To evaluate BugMAPI's accuracy, we deployed a perfectly accurate dynamic analysis for a number of test programs and then classified each bug that this analysis found to be masked, based on BugMAPI's analysis of that same test. The tests we considered are 11 industry-strength tests for Power ISA and 3 fully random tests for Alpha ISA. Five of the Power ISA tests were generated using templates that select instructions randomly from a given set: *arith* uses only arithmetic fixed-point instructions, while *arith_mem* selects from loads and stores as well as arithmetic ones. *vector_mem* and *fp_mem* choose instructions from vector and floating-point instructions, respectively. The remaining six tests target the memory subsystem including memory consistency and address translation. One test program has been generated for each template, varying in length from 26 instructions (test *D*) to 917 instructions (*all*). For the Alpha ISA, we report average results over 100 tests with the specified characteristics. For example, in the tests "*without memory disambiguation*," memory addresses are always treated as non-overlapping.

Our classification provides the following categories:

1. *baseline static*: Bug found masked by the baseline static analysis.

2. *BugMAPI – 100%*: Bug found masked by BugMAPI with 100% probability, but not included in the previous group.

3. *BugMAPI – <100%*: Bug found masked by BugMAPI with less than 100% probability.

4. *BugMAPI – 0%*: Bug found NOT masked by BugMAPI.

5. *store instruction*: Bug injected in store instructions or other instructions affecting a store instruction. This corresponds to the accuracy penalty due to lack of memory address disambiguation.

Figure 3.8 reports the findings of our analysis. For instance, we injected 269 bugs in test *arith*, one for each of its instructions. Among those, 171 were found masked by the dynamic analysis. Our classification found that 107 of the 171 were already masked by our baseline static analysis, an additional 40 were flagged by BugMAPI as possibly masked (<100%) and 24 went undetected by it (0%).

Overall, the baseline static analysis was able to detect, on average, only 62% of the masked bugs in the Power ISA tests, while BugMAPI improves the detection by an additional 15% (100% and <100% categories) for a total of 77% masked bug identification overall. Note that BugMAPI accuracy varies greatly with the contents of the test: it is very high for compute-intensive tests (first five in the plot), but less in memory-intensive tests (letter-named tests). Moreover, as we pointed out in Section 3.5.2, our Power ISA analysis has no memory disambiguation capabilities, which contributes to some low-accuracy results, e.g., test *E*. Another type of limitation is highlighted by test *A*, where 11% of the instructions led to execution failures in dynamic analysis (e.g., due to stores with misaligned address). However, BugMAPI cannot recognize these situations because of its static nature.

The portion of Figure 3.8 related to the Alpha ISA focuses on investigating the impact of memory disambiguation. The first two test-programs in this group differ only on that aspect. Note that disambiguation provides a slight improvement for the baseline static analysis, which is an improvement that BugMAPI benefits from. Finally, the last two test-programs evaluate the accuracy impact due to test length: note that the accuracy of bug-masking assessment remains stable even though the length is increased by five times.

Sources of inaccuracy. BugMAPI strives to keep the computation lightweight at the cost of some accuracy. We have identified a few sources of inaccuracy. First, our solution tracks information flow at a coarse granularity: a whole register or memory location except for a few special-purpose registers, whose fields are treated independently (e.g., FPSCR in Power ISA). While a finer-granularity analysis (e.g., bit-level) would lead to a more accurate estimate, it would be much more computation-heavy. Our simplifying assumptions

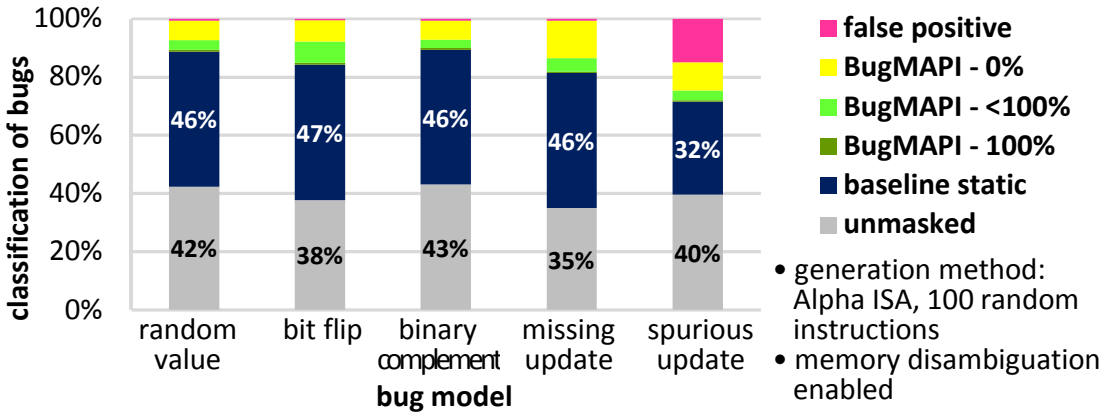


Figure 3.9: BugMAPI's accuracy on various bug models.

(Section 3.4.2) also contribute to inaccurate estimates, because they ignore correlations between instructions, inputs and resources. We observed a minimal fraction of unmasked bugs that are reported masked by our baseline analysis; 1 out of 998 unmasked bugs for Power ISA tests, and 35 out of 4,105 unmasked bugs for Alpha ISA tests.

Finally, BugMAPI fails to recognize correlations between instructions due to test-template structures. For instance, when computing a memory address, multiple instructions are involved that are NOT independent from each other. To address this limitation, it would be possible for BugMAPI to analyze the instruction block as a single instruction with several i/o pairs, and then use those probabilities in the sequence where the block is embedded.

3.5.5 Discussions

Accuracy sensitivity on bug models. We performed additional experiments to measure the sensitivity of BugMAPI to other bug models, using five different types of bug manifestations: *random value overwrite*, *single-bit flip*, *binary complementation*, *missing update* and *spurious update*. Bugs in the first three types are manifested as an incorrect value in the correct target register or memory location. In the fourth type, the target maintains its previous value, ignoring its new value. Lastly, *spurious update* updates a random target other than the correct one. Figure 3.9 shows the accuracy for each bug model. As shown in the bottom of each bar, 38–43% of the bugs are unmasked and detected at the end of tests. Among masked bugs, our baseline analysis identifies 53–81% of them, and BugMAPI adds 6–13% on top of the baseline analysis. While false positives remain minimal for the first four bug models (less than 1%), we find a 15% false positive rate with *spurious update*. This is because our analysis does not track a buggy value manifesting at a random register or memory location other than the correct target.

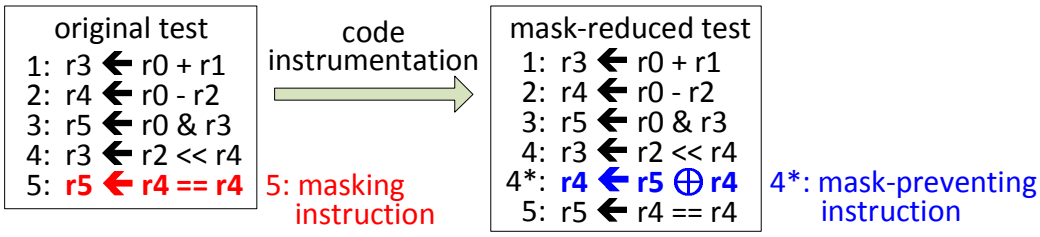


Figure 3.10: Mask-preventing code instrumentation. BugMAPI can identify masking instructions so that mask-preventing instructions can be inserted.

Applying BugMAPI to multi-threaded programs. Analyzing multi-threaded programs can be challenging because of the difficulty in characterizing interactions among threads [170]. In these programs, for instance, inter-thread data dependency may arise through memory accesses to shared memory regions. To extend our analysis to these programs, we can mark inter-thread load and store instructions, applying inter-thread analysis to them. A major challenge comes from non-deterministic execution in multi-threaded programs: inter-thread data dependency may change time to time. One solution for this challenge is to profile the frequency of each data-dependency path, then compute a weighted average across all paths. This approximation may not accurately predict bug-masking occurrences for a specific execution of a program, but it can at least estimate the overall bug-masking likelihood of the program.

3.6 Application: Masking Reduction

The results of BugMAPI's static analysis can be used in various ways to expedite post-silicon validation. In this section, we showcase a BugMAPI's application that reduces bug-masking occurrences in random instruction tests, as illustrated in Figure 3.10. In this application, we first run BugMAPI to collect all instructions that possibly expunge the results of any prior instruction by overwriting its target register. We then instrument mask-preventing instructions that use the value to be overwritten as a source operand so that any buggy value in the operand can be delivered to a propagated (non-overwritten) register.

We implement a simplistic mask-preventing code instrumentation that inserts `xor` instructions before mask-causing instructions identified by our analysis. In the random test shown in the left side of Figure 3.10, instruction 5 is identified as a mask-causing instruction because it expunges the results from instructions 1 and 3. We then insert an `xor` instruction using the operands of instruction 5: the target register (`r5`) and one of the source registers (`r4`), as shown in the right side of Figure 3.10. Note that this register selection

Table 3.1: Number of bugs masked/unmasked by dynamic analysis for our enhanced tests

Test-case	Number of activated bugs		
	Unmasked	Masked	Total
Original tests	4,138 (43%)	5,486 (**57%**)	9,624
Mask-reduced tests	8,893 (72%)	3,384 (**28%**)	12,277

does not require additional registers to be reserved for inserted instructions. Also note that the `xor` instruction itself does not mask any buggy value from its source operands, because its bug-masking probability is 0%. We ignore mask-causing store instructions in order to not perturb memory access patterns, limiting our mask-prevention capability to some degree. While not implemented here, a pair of load and `xor` instructions can further reduce such masking occurrences.

Table 3.1 shows the number of bugs that are classified as either unmasked or masked, throughout our dynamic analysis experiments. We used Alpha ISA with 100 instructions per test, and generated 100 different instruction sequences (10,000 instructions in total), using the random-value bug model and injecting a single bug to each instruction in the tests, including the instructions that we added to reduced the masking rate. As shown in the table, 57% of activated bugs in our original random tests end up being masked. However, this bug masking rate is significantly lowered by our mask-reduction technique, to 28%. Note that our code instrumentation increases the length of each test, due to inserted `xor` instructions.

Our instrumentation does not eliminate all masking occurrences because of two limitations: lack of patches for mask-causing store instructions, and false negatives in our analysis, as discussed in Section 3.5.4.

3.7 Related Work

Test-case generation for microprocessor verification. Randomly generated test-cases are often used to trigger any hard-to-find, unexpected bugs in microprocessor validation [19, 192]. For instance, [19] creates random test-cases by leveraging designer-made test-templates, each targeting a specific test scenario. The tests often select high-quality instructions (i.e., those likely to discover bugs) among those specified in the templates by solving constraint satisfaction problems. Other approaches leverage formal models, often specified through architectural description languages [146].

Researchers have also proposed solutions that create tests with self-checking properties to overcome low observability in post-silicon validation without additional instru-

mentations (e.g., scan chain, design-for-debug network) [198, 87, 129], as we detailed in Section 2.1. Note that, however, tests of this type may not be representative of realistic program sequences.

Information-flow analysis has been investigated extensively in the computer security area [77, 174, 154]. The information flow occurs when information is transferred among distinct program objects (e.g., variables) [77]. To secure the program execution, access-control policies enforce confidentiality throughout the program [174]. In this context, taint analysis (e.g., [154]) takes a dynamic approach that uses run-time information to detect confidentiality violations. Our bug-masking analysis adopts the principle of information-flow from the security area, but with a completely different goal: we aim at identifying perturbations due to bug occurrences that may be erased by the program's subsequent computation. Moreover, in our case, all instruction resources are potential sources of bugs, while in taint analysis only data from untrusted devices is tracked. Information-flow analysis has also been used in soft-error analysis, for instance, [83], discussed in Section 2.1, but without leveraging an instruction's unique characteristic to accurately estimate the propagation probability. Finally, [82] shares with us the deployment of information-flow analysis concepts to hardware verification by annotating RTL designs to compute observability coverage metrics.

3.8 Summary

In this chapter, we proposed BugMAPI that help catch subtle design bugs in post-silicon microprocessor validation. BugMAPI is a novel, lightweight analysis technique to quickly compute the bug-masking probabilities of test programs. It performs a static information-flow analysis for a given test program, estimating the likelihood that bugs go undetected by the checkers at the end of the test program.

BugMAPI decomposes the probability computation from the program level to the instruction level. It first computes the bug-masking probability of each instruction's input-output pair either analytically or experimentally. It then leverages probabilistic approximations to compute the test program's overall bug-masking probability. Experimentally, we have found that BugMAPI provides 77% of the accuracy of a dynamic simulation, at three orders of magnitude shorter computation time.

In the next two chapters, we transition to discussing how to address the correctness of the memory subsystem in systems on chip. This is a particularly challenging problem because of the high number of components that affect the memory interactions of a modern system on chip.

CHAPTER 4

Efficient Post-Silicon Memory Consistency Validation

In this chapter, we propose an efficient validation framework to address design bugs in memory consistency. Memory consistency is a key feature required for efficient multitasking and multithreading in multicore systems on chip (SoCs). Unfortunately, the difficulty of memory consistency validation is on the rise, because of growing complexity of memory subsystems. Memory subsystems usually deploy multiple levels of cache modules, distributed across the entire chip. These cache modules maintain data coherence with each other. Some cache modules are dedicated to specific microprocessor cores, while others are shared over multiple ones. In addition to caches, memory subsystems also include multiple main-memory channels and scratchpad memory components. All these memory components constitute a large, shared memory space, creating various interactions among the components. Consequently, memory operations experience varying memory-access latency, and hence the order in which memory operations complete is also affected by system-level microarchitectural state.

Memory consistency validation strives to check if the system's memory ordering behaviors obey its memory consistency model. To fully verify the memory-access ordering patterns, the system must be tested over all possible interleavings, an almost impossible feat. Instead, we strive to test as many orderings as possible within the time allowed for validation. To this end, we choose to utilize post-silicon validation platforms, because they can execute programs at the system's native speed, thus validating many ordering patterns quickly.

In post-silicon memory-consistency validation, we notice that result-checking is the key bottleneck in the entire validation flow. During result-checking, we must consider happens-before relationships among individual memory operations in multithreaded instruction tests. This justification process strives to detect contradictory happens-before relationships, if they exist. In finding contradictory relationships, we construct a constraint

graph where vertices represent memory operations and edges indicate partial orders between two memory operations. We then find a cyclic dependency in the graph by applying topological sorting, which requires significantly more efforts than the other validation steps (e.g., test execution).

Thus, the key contribution of this effort is on reducing the computational complexity of this result-checking computation (i.e., topological sorting). Traditionally, topological sorting is applied to the entire constraint graph corresponding to each test run. Note that post-silicon validation relies on repetitive test runs to fully verify different microarchitectural behaviors that the system may exhibit. We noticed that these repetitive test runs exhibit similar memory ordering patterns. This similarity, in turn, provides us an opportunity to slash the computational complexity by leveraging decomposition.

The validation framework we propose in this chapter is called MTraceCheck. MTraceCheck strives to boost the performance of checking the correctness of memory-access interleaving observed during the test runs. It does so by *decomposing the problem of validating many large constraint graphs, into smaller problems*. Specifically, for each new graph, we extract the portion that has never been observed before, and then we simply verify that the portion extracted does not contain any cyclic dependency. To further streamline the problem, we approximately sort the graphs by smallest incremental difference using a signature-based approach.

Here, we perform *no reassembly computation* from decomposed graphs, even if it may miss intricate memory ordering violations, which could only be caught by sophisticated inferences applied across decomposed graphs. Having said that, we show experimentally that our solution experiences practically no accuracy degradation in bug-injection experiments conducted on a full-system multicore simulator.

4.1 Background and Motivation

Memory consistency model. In this chapter, we focus on checking that the memory-access interleavings observed during tests executions comply with the memory consistency model (MCM). An MCM provides an interface between hardware designers and software developers, by specifying what memory re-orderings are acceptable and could be observed during a software execution [185, 20, 119, 180, 21]. For software developers, they must be aware of the memory re-ordering behaviors allowed by the MCM when they code multithreaded programs. Memory operations can be re-ordered, thus, executed in an order different from the order specified in the program, as long as such re-ordering is allowed by the MCM. For hardware designers, on the contrary, they should ensure the system ex-

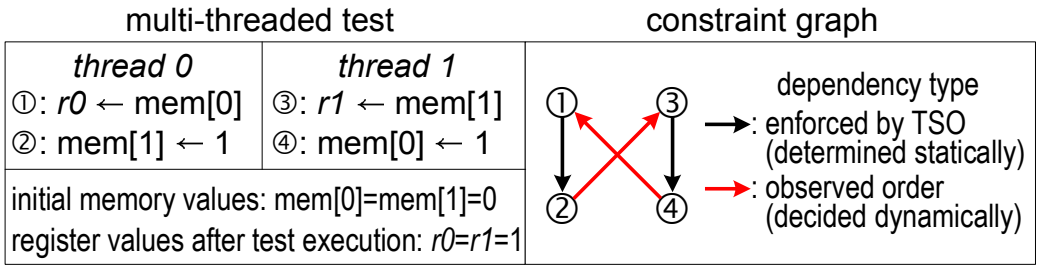


Figure 4.1: A two-threaded test program execution and its constraint graph. Among various test results, `r0=r1=1` is considered invalid in the TSO model. Correspondingly, its constraint graph identifies a cycle.

hibits no memory re-ordering behaviors disallowed by the MCM. In other words, memory subsystem components must not re-order memory operations if such re-ordering behaviors can be exposed to software (i.e., at the architecture level). Here, we strive to catch design bugs made by hardware designers; when they fail to abide by the MCM, the hardware implementation may exhibit memory re-ordering behaviors that software developers consider invalid.

There are a wide spectrum of MCMs proposed in the literature. At one end of the spectrum, MCMs impose almost no memory ordering constraints, allowing a variety of re-orderings among memory operations. Such relaxed MCMs (e.g., relaxed memory order (RMO) [202, 36, 140]) are beneficial to improve the overall system performance, as they allow more memory operations to be processed in parallel, without being subject to ordering constraints. At the other end of the spectrum, MCMs can impose all memory ordering constraints (i.e., load→load, load→store, store→load, store→store). Such strong MCMs (e.g., total store order (TSO) [102, 180], which relaxes the store→load constraint from the aforementioned four constraints) are very intuitive for software developers; they can easily write multithreaded programs for a system that implements a strong MCM. However, the system may not able to fully utilize memory subsystems, because many memory operations in multithreaded programs must be serialized due to ordering constraints. For instance, with reference to the simple program in Figure 4.1, there are 4 different outcomes allowed by the RMO: `r0` with either 0 or 1, and `r1` with either 0 or 1. Some of these outcomes are forbidden in the TSO, which disallows the outcome `r0=r1=1`. As program's size increases, so does the number of memory accesses and of re-orderings.

Constraint graph of memory operations. To validate the correct implementation of the MCM in post-silicon, or any other simulation-based environment, the observed ordering of memory operations must be recorded on the fly. These access logs are then checked

against the re-ordering rules allowed by the MCM of the multi-core architecture. As we also discussed in Section 2.2, several works have proposed to use a constraint graph to validate the observed access sequence (e.g., [57, 96, 63, 65, 136, 23, 131, 139, 132]). The vertices of this graph correspond to each memory operation (either store or load), while edges mark a dependency between two operations. Figure 4.1 provides the constraint graph for a simple two-threaded program execution under the TSO model. The outcome shown in the figure reveals a consistency violation because there is a cyclic dependency in the corresponding constraint graph. Indeed, in TSO store operations cannot be speculatively performed before a preceding load operation, thus `r0` and `r1` cannot be both equal to 1 at the end of the execution. The violation is thus indicative of a microarchitectural optimization that allows multiple outstanding memory operations, against the model's requirements.

Constraint graph construction has also been thoroughly studied in recent years. In this chapter, we adopt the same notation as in recent previous work [23, 131] and model three types of observed edges as follows: *reads-from* (rf), *from-read* (fr), and *write serialization* (ws). Besides observed edges, we also model intra-thread consistency edges as defined by the MCM. For instance, in TSO, the only reordering allowed is that loads can be completed before the preceding store operation.

Note that checking for cyclic dependencies in a constraint graph usually requires heavy computation. There are two conventional approaches: topological sorting and depth-first search (DFS). For both methods, the computational complexity is known to be $\Theta(V + E)$ where V and E are the set of vertices and edges, respectively [71]. In post-silicon validation, where many test results are generated at the system's native speed, checking these graphs can be the key bottleneck of the entire memory validation process, as we will show in Section 4.5.3.

To **capture memory-access interleavings** at runtime, test programs must be instrumented as in prior works [96, 136]. Specifically, every store operation is assigned a unique ID, which is the value actually written into memory, so that the operation can be easily identified by subsequent loads. In addition, memory-access interleavings are captured by inspecting the values read by load operations. Thus, we can establish the uniqueness of a test execution based on its reads-from relationships; two executions have experienced distinct memory access interleavings when they exhibit at least one different reads-from relationship.

Testing framework. In evaluating the proposed solution, we adopt a constrained-random testing framework as follows: we first generate a number of test programs that are designed

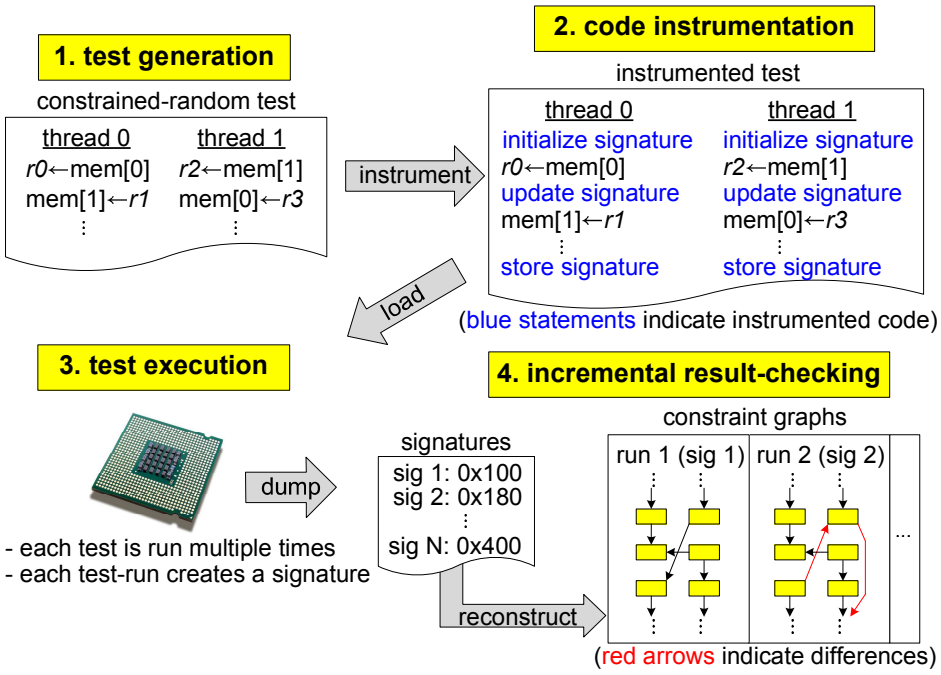


Figure 4.2: MTraceCheck overview. Multi-threaded constrained-random tests are generated and then augmented with our observability-enhancing code, which generates a memory-access interleaving signature at runtime. After multiple test runs, signatures are collectively checked by exploiting their similarities, accelerating the validation process.

to likely exhibit numerous distinct memory-access interleavings. Each test program is run repeatedly until rare interleavings are observed. There are a number of prior studies focusing on test generation (see Section 4.8); in contrast, we focus mainly on the subsequent verification steps. Specifically, we strive to (1) improve the observability of the memory-access interleavings occurring during test execution while minimally perturbing the original access sequences. We also want to (2) reduce the computation requirements for violation checking, so to boost the coverage and efficiency that can be attained in validating the MCMs in post-silicon [115].

4.2 MTraceCheck Overview

Figure 4.2 summarizes the MTraceCheck validation flow in four steps.

Step 1: Test generation. MTraceCheck generates multi-threaded tests that are designed to stimulate rare memory-access interleavings. Specifically, the test generator has several parameters to guide the generation process: the number of threads, the number of memory operations, the number of memory locations, etc. The test generator can create three types

of memory operations, load, store, and fence operations. And the test generator takes the occurrence frequency of each type as a parameter. The address of each load or store operation is randomly chosen within the shared memory space.

Step 2: Code instrumentation. The generated tests are augmented with our observability-enhancing code, which computes a compact signature that corresponds to the observed memory-access interleaving for a given test execution. To this end, we first perform a static code analysis on the generated multi-threaded test to identify memory operations that share the same address. We then insert a signature-update code after each load operation to log the value read by the load operation. We describe this step in detail in Section 4.3.

Step 3: Test execution. The instrumented tests are run multiple times in a post-silicon validation platform. Throughout the test execution, each thread computes its own signature capturing the values read by load operations in the thread. These per-thread signature are collected later to form an execution signature when all threads are completed.

Step 4: Incremental result checking. After the test execution is done, we collect all the execution signatures from the repeated runs. We reconstruct the constraint-graphs that are represented by the signatures. We then check the graphs in a collective manner that detects and exploits similarity among different runs (thus, distinct memory-access interleavings). This collective graph checking is enabled by our novel decomposition technique that extracts the differences among graphs. Only the differences are incrementally checked by the proposed graph checking scheme. This step is detailed in Section 4.4.

4.3 Code Instrumentation

This section discusses our code instrumentation approach that records the memory-access interleavings observed during a test's execution in post-silicon validation. Our solution is inspired by the control-flow profiling methodology proposed in [53]. However, instead of profiling control-flow paths, we repurpose the profiling framework to log memory-access interleavings in a compact signature format, that will then be categorized and checked during the subsequent checking step, discussed in Section 4.4.

4.3.1 Memory-Access Interleaving Signature

To track memory-access interleavings, we need to log the values that are being loaded throughout the test execution. This task can be carried out by simply flushing the values

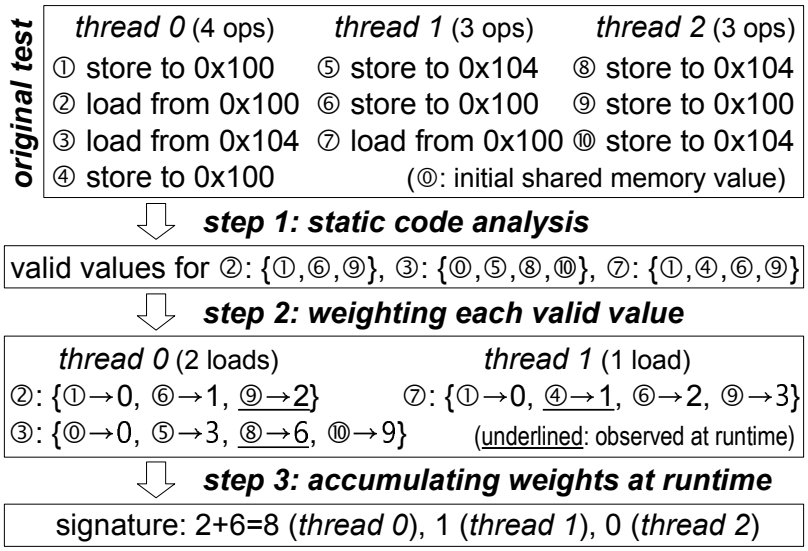


Figure 4.3: Memory-access interleaving signature. Each signature value corresponds to a unique memory-access interleaving, observed during the test run. For each load, we profile all possible values, and each is assigned an integer weight. At runtime, the weights of the observed values are accumulated, forming a per-thread signature. The per-thread signatures are then gathered to form an execution signature.

loaded into the register file to a dedicated memory area, as illustrated in [96]. Lacking a dedicated storage resource and transfer channel, this task is bound to interfere with the execution flow of a test, possibly altering the latency and interleavings experienced by the test's accesses. Moreover, it burdens overall execution time with additional memory store operations.

Thus, to reduce the impact of data logging, we introduce the new concept of *memory-access interleaving signature*: MTraceCheck bypasses the frequent storing of loaded values by computing a compact signature of the loaded values. To this end, we must augment the original test with signature-computation code. Note that our signature computation resembles the path-encoding computation in [53].

Figure 4.3 illustrates our code instrumentation process. In the first step, we perform a static code analysis to collect all possible values that could be the result of each load operation, as illustrated in the first and second boxes. This static analysis can attain perfect memory disambiguation if the test generator is configured to do so. This is straightforward to attain with a constrained-random test generator. In the case of general software test programs, it is still possible to do this analysis either (1) by excluding from the signature computation all the memory addresses that cannot be statically disambiguated, or (2) by dynamic profiling of the actual accesses using a binary instrumentation tool.

We then assign a weight to each loaded value, as illustrated in the third box of the

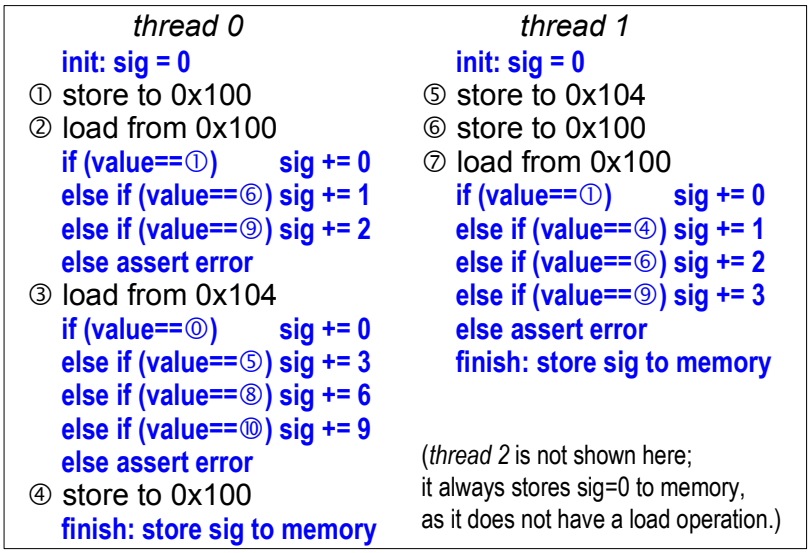


Figure 4.4: Code instrumentation. A signature is computed as the test runs, using chains of branch and arithmetic instructions. The computed signature is then stored in a non-shared memory region at the end of test.

figure. The weights are integer values consciously assigned to obtain unique final signature values. Specifically, we use consecutive integers for the first load operation. Then, if the first load operation could retrieve n distinct values, we use multiples of n for the weights of the second load operation, and so on. For instance, load operation ② can only read the value stored by either ①, ⑥ or ⑨. The weights for these values are assigned to 0, 1 and 2, respectively. Moving to the next load operation, ③, we use multiples of 3 for the weight of each possible loaded value because we had three options in the prior load operation. If there were another load operation after ④, we would use multiples of 12 for its weight, because there are already 3×4 combinations for the previous loads. By allocating non-aliasing weights (except for 0) to each loaded value and operation, we can guarantee a unique correspondence between signatures and set of loaded values, that is, a 1:1 mapping between signatures and interleavings. In the figure, suppose that two underlined values (⑨ and ⑧) are observed in thread 0, then the signature value for this thread would be 8. Note that weights are assigned and accumulated independently for each thread. Finally, we form the *execution signature* for the test execution by concatenating the per-thread signatures obtained.

Figure 4.4 illustrates an example of instrumented code corresponding to the original code shown in Figure 4.3. The `sig` variable is initialized at the beginning of the test, and increased after each load operation. Specifically, each load operation is followed by a chain of branch statements depending on the loaded value. While not essential for observability

enhancement, we append an additional assertion at the tail of the chain so that obvious errors (e.g., a program-order violation) can be caught instantly without running a constraint-graph checking. This instrumentation does not perturb the sequence of memory accesses in the test execution, unless an error is caught by the assertion. With branch predictors in place, MTraceCheck only slightly increases test execution time as shown in Section 4.5.3. Note that this approach may entail minimal false-negatives due to these added branch operations, which could alter the original branch-prediction pattern of the test.

Algorithm 4.1 Signature decoding procedure

1: **Input**: $signature$, $multipliers$, $store\_maps$
2: **Output**: $reads\_from$
3: **for** each load L from last to first in test program **do**
4: $multiplier \leftarrow multipliers[L]$
5: $index \leftarrow signature \;/\; multiplier$
6: $signature \leftarrow signature \,\%\, multiplier$
7: $reads\_from[L] \leftarrow store\_maps[L][index]$
8: **end for**
9: **return** $reads\_from$

4.3.2 Per-thread Signature Size and Decoding

In a relaxed MCM with no memory barrier, the signature size is proportional to the number of memory-access interleavings that are allowed by the MCM. Namely, the more diversified the interleavings possible, the larger the signature. This relationship does not hold for stronger MCMs, such as TSO, as we discuss later in Section 4.7.

We estimate the size of the signatures generated under the assumption that all accesses can be disambiguated and used for signature computation (as it is the case for constrained-random tests). Let T be the number of threads, S and L the number of stores and loads per thread, respectively, and A the number of shared memory locations. If assuming that addresses are uniformly randomly chosen, each load operation could read the same expected number of distinct values. The value read during a load is either the last stored value from the same thread (the 1 in the expression), or any of the stored values ($\frac{S}{A}$) from any of the other threads ($T-1$). Hence, we estimate that each per-thread signature must be capable of representing

$$\texttt{signature cardinality} = \left\{1 + \frac{S}{A}(T-1)\right\}^{L}$$

distinct values. Note that the L power is required to extend our estimation to all the load

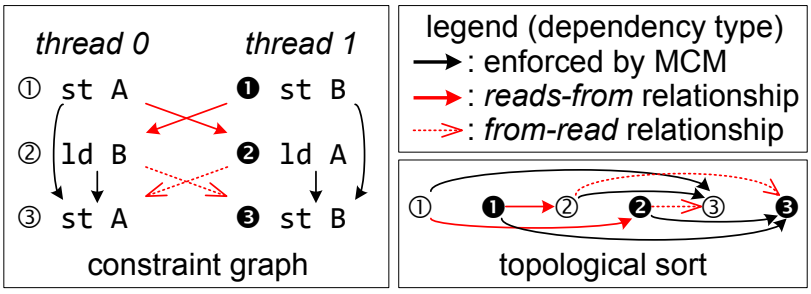


Figure 4.5: Topologically-sorted constraint graph. The constraint graph on the left can be topologically sorted as shown on the right, indicating no consistency violation.

operations in the thread. The execution signature requires T times that space, since each thread generates a per-thread signature. As an example, with S=L=50, A=32 and T=2, a per-thread signature must be capable of representing $\{1+\frac{50}{32}(2-1)\}^{50} \approx 2.7\times10^{20} \approx 2^{68}$ sets of loaded values. Thus, 68-bit of storage are required for each thread. In the systems that we used in our evaluation (Section 4.5.1), registers are either 64-bit or 32-bit wide; thus, a 68-bit signature must be split into multiple words. To support multi-word signatures, we statically detect overflow when instrumenting the tests with observability-enhancing code. When an overflow is detected, we add another register to store the signature for the thread (and another variable `sig2` in Figure 4.4); we then start over the signature computation in the new register, resetting the weight multipliers. In Section 4.5.4, we provide detailed experimental results on the average signature size for various constrained-random tests. We further discuss how to reduce the signature size in Section 4.7.

Reconstructing memory-access interleavings and constraint graphs. Algorithm 4.1 summarizes our decoding process for a per-thread signature. The goal of the reconstruction process is to translate a `signature` into a set of observed `reads-from` relationships, which will allow us to build a constraint graph for the corresponding test execution. The algorithm we developed first splits the execution signature into a set of per-thread signatures. Then it applies reconstruction to each signature obtained, starting from the last load operation in the test, and walking backwards toward the first one. Note that we maintain a special table for each test under analysis, called the `multipliers`. The table keeps track of the weight multiplier used for each load instruction in the test, and it is generated during test instrumentation. Simultaneously, we maintain the index-store mapping table for each load operation, called the `store_maps`. Thus, during reconstruction, our algorithm searches the weight multiplier to use for the load operation from the `multipliers`, and the corresponding store operation using the `store_maps` (third box in Figure 4.3). Then the `signature` is decreased accordingly to remove the weight component of the load just

reconstructed.

Note that all other information necessary to build the constraint graph is gathered statically during the instrumentation process. That is, the `multipliers`, the `store_maps`, the intra-thread dependencies specified by the MCM (e.g., load →load) and the write-serialization order. We then use this information to generate constraint graphs, as discussed in Section 4.1, one for each distinct test execution.

4.4 Collective Graph Checking

As briefly introduced in Section 4.1, topological sorting is a classic solution to check for a memory consistency violation. *A topological sort of a directed graph is a linear ordering of all its vertices such that if the graph contains an edge (*u*,*v*), then* u *appears before* v *in the ordering* [71]. If a constraint graph cannot undergo topological sorting, that is an indication that a cyclic dependency is present, hence a violation of the MCM occurred. Note, however, that if some dependency edges are missing, false negatives may result from the analysis. For instance, topological sorting may not consider inferred dependency edges, such as those introduced in [48]. Figure 4.5 shows an example of a constraint graph and its topologically sorted correspondent. In this example, there is no violation and thus a topological sort exists.

In prior works, where topological sorting was applied to constraint graphs obtained from post-silicon tests, the graph obtained from each test execution was analyzed individually. However, as we illustrate in Section 4.5.2, we observed that *many test runs exhibit similar memory-access interleaving patterns*. Thus, we propose to leverage the similarity among test runs to reduce the computation required to validate each execution. Below, we present a *collective* constraint-graph checking solution: we first investigate similarities among constraint graphs (Section 4.4.1), and then present our novel re-sorting technique that can quickly validate a collection of constraint graphs (Section 4.4.2). In the following discussion, we assume that all duplicate identical executions have been removed beforehand. In our implementation, duplicate executions are filtered out when execution signatures are sorted (Section 4.4.1).

4.4.1 Similarity among Constraint Graphs

Constraint graphs generated from a same test program have all the same vertices but possibly different edges. Thus, we identify the difference between two constraint graphs by tagging these differing edges. After we isolate the portion of a graph that differs from a previously-analyzed graph, we can isolate the discrepant portion for the sake of topological

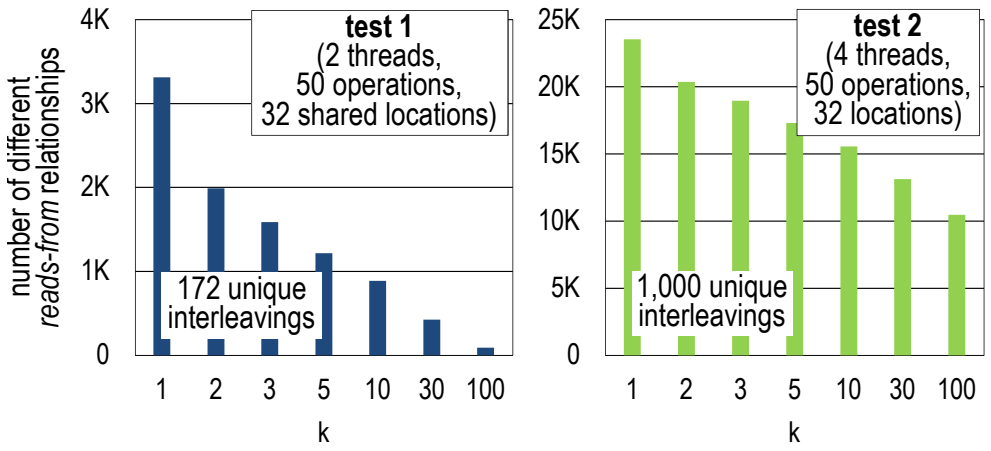


Figure 4.6: Measuring similarity among constraint graphs using k-medoids clustering. The number of differing reads-from relationships decreases as k increases. However, the cluster tightness is reduced with a more diverse pool of possible interleavings (right graph).

sorting. Thus, the graph-checking computation can be greatly reduced by incrementally verifying, that is, sorting only this portion.

Limit study – k-medoids clustering. While this incremental verification is promising, finding the graph most similar to the one under analysis is non-trivial and often computation-heavy. To gain insights on this aspect, we performed a preliminary study to identify a handful of graphs representative of the entire set of constraint graphs generated from many executions of a test. Specifically, we conducted a k-medoids clustering analysis [12] for constraint graphs from each of two test programs, measuring the total number of different reads-from relationships from the closest medoid graph. The selected k medoid graphs are considered representative of the entire set. Here, reads-from relationships are obtained from an in-house architectural simulator, which selects memory operations to execute in a uniformly random fashion, one at a time, and without violating sequential consistency (SC) [119]. For simplicity, we assumed single-copy store atomicity [48] for this limit study.

Figure 4.6 shows the number of total differing reads-from relationships for varying k values. In test 1, where we obtained 172 unique executions out of 1,000, the reads-from relationships decrease rapidly as k increases. However, in test 2, where every execution is unique, many discrepant reads-from relationships persist into high k values, indicating that the representative medoids are vastly different from the individuals. Moreover, the computational complexity of finding the optimal solution of k-medoids clustering is very high [12]. Thus, using this approach is computationally prohibitive, and it negates the performance benefits of a fast topological sorting.

Our solution – sorting signatures and diffing corresponding graphs. Instead of finding the graph most similar to the one under analysis, MTraceCheck opts for using a lightweight computation that finds a graph sufficiently similar to it. For this purpose, we re-use our memory-access interleaving signatures, introduced in Section 4.3. Once we collect all signatures from multiple executions of a test, we sort the execution signatures in ascending order. Since adjacent signatures in the order correspond to graphs with small differences between each other, we can use the graph analyzed for one signature as the basis against which to analyze the next one.

As noted in Section 4.3.1, execution signatures are generated by concatenating per-thread signature words from all test threads. Specifically, we place the signature word from the first thread in the most significant position, and the one from the last thread in the least significant position. If per-thread signatures require multiple words, we place the first signature word in the most significant position, and the last in the least significant position. To evaluate the impact of this data layout, we also performed a sensitivity study, placing signature words from related code sections in different threads near each other. This alternative layout, however, led to worse similarity between constraint graphs from adjacent signatures.

4.4.2 Re-sorting Topological Order

With the signatures sorted in ascending order, MTraceCheck examines each of the corresponding constraint graphs for consistency violations. Our checking starts with the first constraint graph in the sorted order. This first-time checking is performed as a complete, conventional graph checking, topologically sorting all vertices. Starting from the second graph, our checking algorithm strives to perform a partial re-sorting only for the portion that differs from the prior graph.

The re-sorting region is determined by two boundaries: leading boundary and trailing boundary. Only the vertices between the two boundaries need to be re-sorted. The leading (trailing) boundary is the first (last) vertex in the original sorting that is adjacent to a new backward edge from the graph under consideration. Note that neither forward nor removed edges need be considered here, since they only release prior sorting constraints. If there is no new backward edge, re-sorting is unnecessary and is thus skipped. Our re-sorting method is as precise as the conventional topological sorting; we omit the formal proof here due to limited space.

Figure 4.7 illustrates our re-sorting procedure on four constraint graphs obtained from a two-threaded program. The first run's topological sort is computed using a conventional complete graph checking. For the second graph, MTraceCheck examines each of the newly

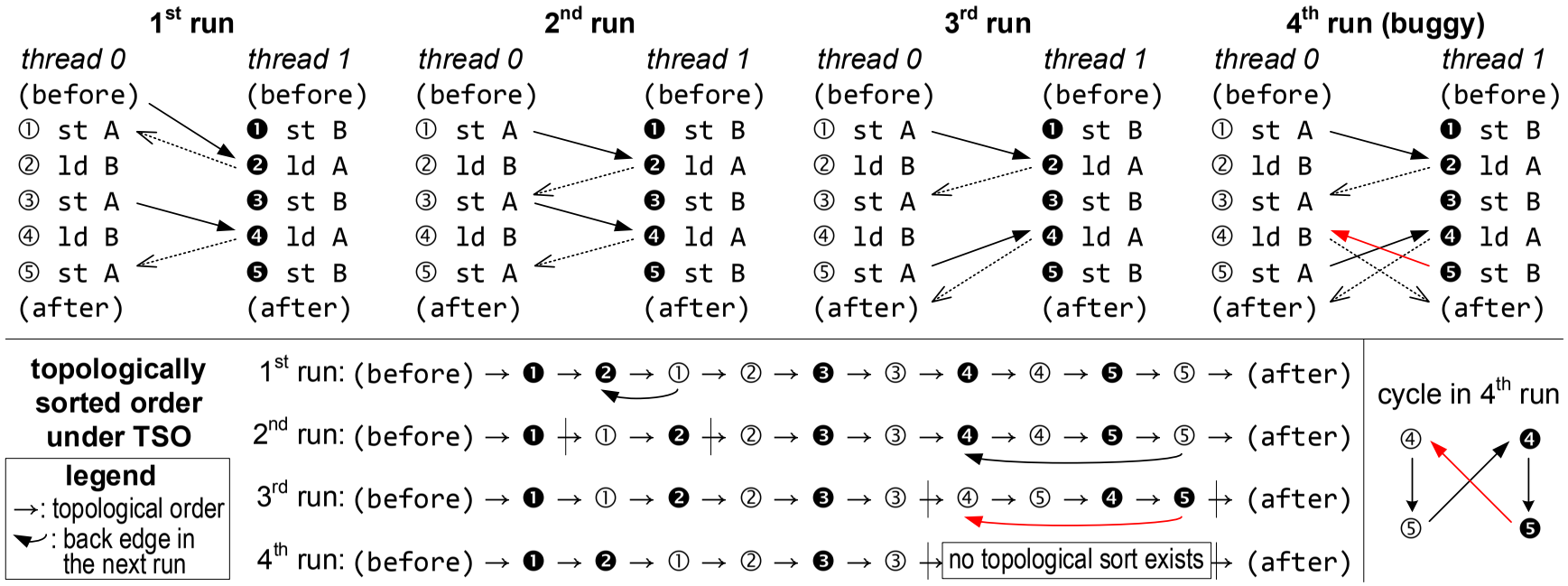


Figure 4.7: Re-sorting a topological-sorted graph for a series of test runs. To check for a consistency violation, the topological sort from the previous run is partially re-sorted for the next run. The sorting boundaries (leading and trailing) are calculated from backward edges. When no topological sort exists in the segment between the boundaries, it indicates a violation.

Table 4.1: Specifications of the systems under validation

System 1. **x86-64** – Intel Core 2 Quad Q6600

MCM	x86-TSO [102, 180]
Operating frequency	2.4 GHz
Number of cores	4 (no hyper-threading support)
Cache architecture	32+32 KB (L1), 8 MB (L2)
Cache configuration	Write back (both L1 and L2)

System 2. **ARMv7** – Samsung Exynos 5422 (big.LITTLE)

MCM	Weakly-ordered memory model [36, 34]
Operating frequency	800 MHz (scaled down)
Number of cores	4 (Cortex-A7) + 4 (Cortex-A15)
Cache architecture	A7: 32+32 KB (L1), 512 KB (L2) A15: 32+32 KB (L1), 2 MB (L2)
Cache configuration	Write back (L1), write through (L2)

added edges in the graph, ①→❷ and ❷→③. Only the former is backward, as shown by the backward arrow below the topological sort of the first run. Thus, ❷ is the leading boundary and ① is the trailing one. The order of the two nodes is then simply swapped to achieve a new topological sorting, as shown in the second run's diagram. Similarly, in the third run, four vertices must be re-sorted. The fourth run exposes a bug: indeed there is no topological sort for the four affected vertices due to the backward edge ❺→④. The absence of topological sort is also illustrated by the cycle highlighted in the bottom right part of the figure.

4.5 Experimental Evaluation

4.5.1 Experimental Setup

Systems under validation. MTraceCheck was evaluated in two different systems, an x86-based system and an ARM-based system, as summarized in Table 4.1. For each system, we built a bare-metal operating environment (i.e., no operating system) specialized for our validation tests. In our x86 bare-metal environment, the boot-strap processor awakens the secondary cores using interprocessor interrupt (IPI) messages, followed by cache and MMU initializations. Test threads are first allocated in the secondary cores, and then in the boot-strap core, but only when no secondary core is available. In our ARM bare-metal environment, the primary core in the Cortex-A7 cluster runs the Das U-Boot boot loader [7], which in turn, calls our test programs. At the beginning of each test, the primary core powers up the secondary cores. The secondary cores are then switched to supervisor mode

Table 4.2: Test generation parameters

Number of test threads	2, 4, 7
Number of memory operations per thread	50, 100, 200
Number of distinct shared memory addresses	32, 64, 128

and their caches and MMUs are initialized. Note that the primary core remains in hypervisor mode to keep running the boot loader.[1] Test threads are allocated in the big cores in the Cortex A15 cluster, then in the little cores in the Cortex A7 cluster.

Test generation. Table 4.2 shows key parameters used when generating constrained-random test programs. We chose 21 representative test configurations by combining these parameters, as shown on the x-axis of Figure 4.8. The naming convention for the 21 configurations is as follows: `[ISA]-[test threads]-[memory operations per thread]-[distinct shared addresses]`. For example, `ARM-2-50-32` indicates a test for the ARM ISA with 2 threads, 50 memory operations using 32 distinct shared memory addresses. For each test configuration, we generated 10 different tests, and run each 5 times. The tests perform load and store instructions, transferring 4 bytes each, with equal probability (i.e., load 50% and store 50%).

Each test run executes a loop that encloses the generated memory operations so as to observe various memory-access interleaving patterns. Unless otherwise noted, the iteration count for this loop is set to 65,536. In addition, the beginning of the loop includes a synchronization routine waiting until the previous iteration is completed, followed by a shared-memory initialization and a memory barrier instruction (`mfence` for x86 and `dmb` for ARM). The synchronization routine is implemented with a conventional sense-reversal centralized barrier. To remove dependencies across test runs, we applied a hard reset before starting each test run.

4.5.2 Non-determinism in Memory Ordering

The dark-blue bars in Figure 4.8 present the number of unique memory-access interleaving patterns across various test configurations, measured by counting unique signatures. Each multi-threaded test program runs 65,536 times in a loop, except for `ARM-2-200-32*` where each test program runs 1 million iterations. We then average our findings over 5 repetitions of the same experiment. To summarize the figure, we observe almost no duplicates in several configurations on the left side of the linear-scale graph, while we ob-

[1] We could not launch a test thread in the primary core, because of the discrepancy of the operating mode, causing unexpected hangs.

serve very few distinct interleavings in several test configurations on the right logarithmic-scale graph. For instance, ARM-7-200-64 presents 65,536 unique memory-access interleaving patterns (100%), while ARM-2-50-32 only reveals 10.7 unique patterns on average (0.02%).

Among the three parameters of Table 4.2, the number of threads affects non-determinism the most. We measured 6.6 distinct patterns in ARM-2-50-64, 22,124 patterns in ARM-4-50-64, and 65,374 patterns in ARM-7-50-64. Note that the total number of memory operations are different in these three configurations. To keep the total number of operations constant, we compared configurations with the same number of operations (ARM-2-100-64 vs ARM-4-50-64), also observing a significant increase (123 vs 22,124).

Also, the number of memory operations per thread affects non-determinism, but less significantly than the number of threads does. We observed 10.7 patterns in ARM-2-50-32, 508 patterns in ARM-2-100-32, and 35,679 patterns in ARM-2-200-32. We also found that increasing the number of shared memory locations leads to fewer accesses to a same location, thus reducing the number of unique interleaving patterns. ARM-2-200-64 exhibits only 9,638 patterns, which is fewer than the 35,679 patterns in ARM-2-200-32.

In most seven-threaded configurations, we found that almost all iterations exhibit very different memory-access patterns. This is partly because of the relatively low iteration count, which is set to 65,536. To evaluate the impact of the iteration count, we performed a limited sensitivity study for ARM-2-200-32, where we compare the results from two iteration counts: 35,679 unique interleavings out of 65,536 (54%) vs. 311,512 unique interleavings out of 1,048,576 (30%). We expect a similar trend in the seven-threaded configurations, that is, that the fraction of unique interleavings decreases as the iteration count increases.

In the x86-based system, we observe that interleaving diversity is more limited compared to the ARM-based system, because the MCM of the x86-based system is stricter. We believe that the MCM is the major contributor for this difference, among other factors, such as load store queue (LSQ) size, cache organization, interconnect, etc. Note that, for comparison fairness, we used the same set of generated tests (i.e., same memory-access patterns) for both systems.

Impact of false sharing of cache line. In cache-coherent systems like the ones we evaluated here (Table 4.1), the MCM is intertwined with the underlying cache-coherence protocol. Data is shared among cores at a cache-line granularity, so placing multiple shared words in a same cache line creates additional contention among threads, further diversifying memory-access interleaving patterns. For the dark-blue bars in Figure 4.8, only one

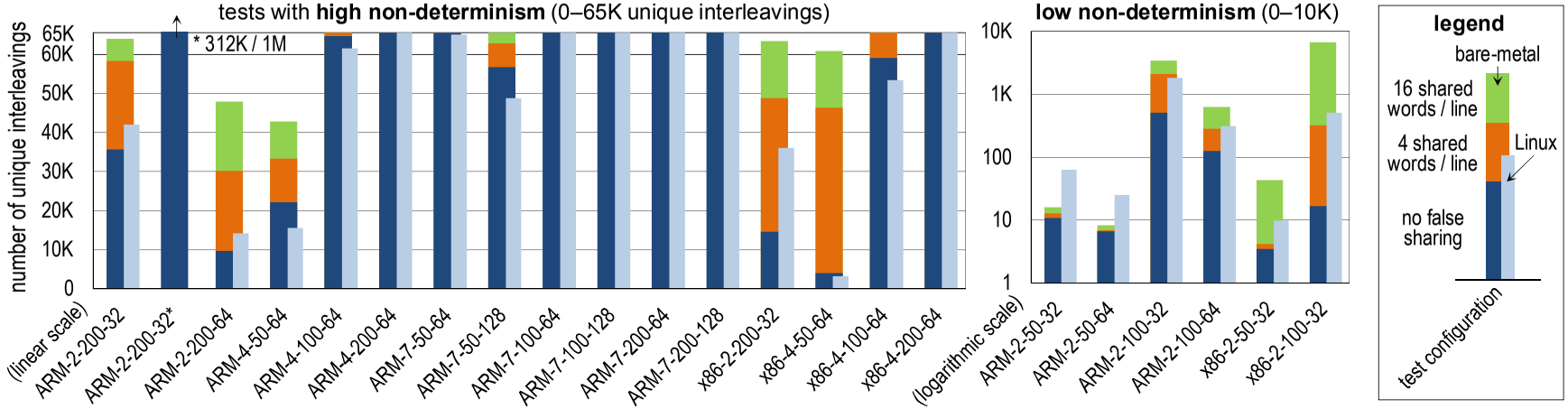


Figure 4.8: Number of unique memory-access interleavings. Unique memory-access interleavings diversify as more threads share the same memory space. Short two-threaded tests only exhibit a handful of distinct interleavings, while long seven-threaded tests generate new unique interleavings in almost all iterations. False sharing increases contentions, thus further diversifying interleavings. The OS contributes to creating additional unique interleavings in two-threaded tests, while the opposite trend holds in four-threaded and seven-threaded tests.

shared word (4 bytes) is placed in each cache line (64 bytes), thus no false sharing exists.

The orange and green bars in Figure 4.8 present the numbers of unique memory-access interleaving patterns for two different data layouts; 4 and 16 shared words per cache line, respectively. As expected, false sharing contributes to diversifying memory-access interleaving patterns. `x86-4-50-64` shows the most dramatic increase among others: from 3,964 (no false sharing) to 46,266 (4 shared words) to 60,868 (16 shared words) unique interleavings. Also, the increase is more marked for the x86-based system than for the ARM-based system.

Impact of the Operating System. Our bare-metal operating environment allows only the test program to run when testing is in progress; there is no interference with other applications. However, when our test programs are running under the control of an operating system, some test threads can be pre-empted by the OS scheduler. In addition, the layout of shared memory may be shuffled by the OS.

To quantify the perturbation of an OS, we re-targeted the same set of tests to a Linux environment: Ubuntu MATE 16.04 for the ARM-based system and Ubuntu 10.04 LTS for the x86-based system. Test threads are launched via the m5threads library [56], although, after launch, synchronizations among test threads are carried out by our own synchronization primitives as in the bare-metal environment.

The light-blue bars in Figure 4.8 report our findings for the Linux environment with no false sharing. There are two noticeable trends. Firstly, in two-threaded tests, the number of unique interleavings increases compared to the bare-metal counterparts. In both four-threaded and seven-threaded tests, on the contrary, the opposite tendency holds. We believe that fine-grained (i.e., instruction-level) interferences dominate in two-threaded tests, while coarse-grained (i.e., thread-level) interferences dominate in more deeply multi-threaded setups, such as four-threaded and seven-threaded tests.

4.5.3 Validation Performance

In this section, we measure two major component costs of validation time. We first evaluate the time spent checking for MCM violations on a powerful host machine equipped with an Intel Core i7 860 2.8 GHz and 8 GB of main memory, running Ubuntu 16.04 LTS. We then report the time spent executing tests in our ARM bare-metal system. We carried out the violation checking on a host machine, instead of the system under validation, because of the heavy computation involved.

Figure 4.9 compares the topological-sorting time for our collective graph checking solution, normalized against a conventional approach that checks each constraint graph in-

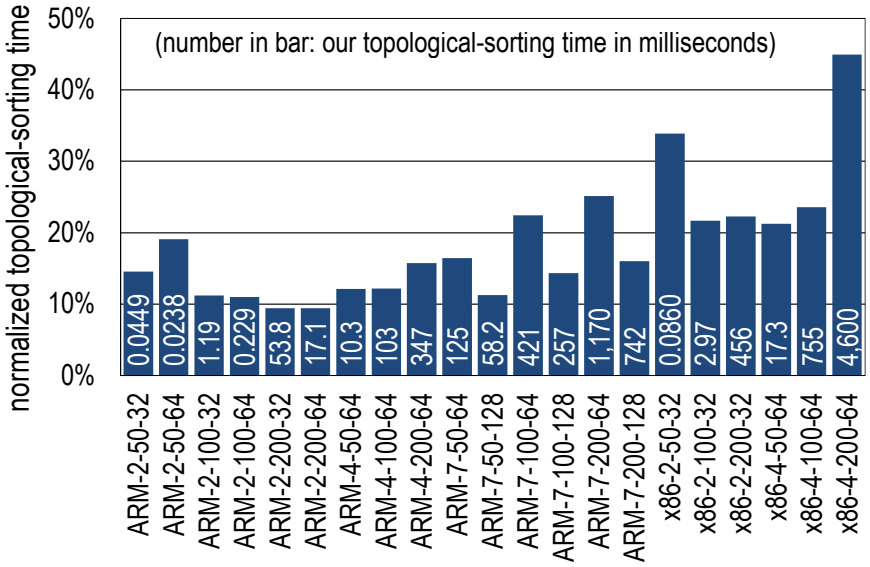


Figure 4.9: MCM violation checking – topological sorting speedup. MTraceCheck reduces overall topological sorting time by 81% on average, compared to a conventional individual-graph checking solution.

dividually. We observe that our collective checking greatly reduces overall computation, consistently across all test configurations. Compared to the conventional solution, MTraceCheck takes only 9.4% (`ARM-2-200-64`) to 44.9% (`x86-4-200-64`) of the time spent by the conventional topological sorting, achieving an 81% reduction on average. We also observe a noticeable difference between the ARM and x86 platforms: the benefit of our technique is smaller in x86. We provide insights on this difference in Section 4.7.

In this evaluation, for reproducibility, we adopted a well-known topological-sort program, `tsort`, included with GNU core utilities [10]. We then modified the original `tsort` to support multiple constraint graphs generated from the same test. Specifically, for both the conventional and MTraceCheck techniques, vertex data structures are recycled for all constraint graphs, while edge data structures are not. The measured time excludes the time spent in reading input files, thus assuming that all graphs are loaded in memory beforehand. We ran `tsort` 5 times for each evaluation to alleviate random interferences due to the Linux environment. Due to extremely slow communication between our bare-metal systems and the host machine, we used the signatures obtained in the Linux environment (the light-blue bars in Figure 4.8). For fairness, we considered only unique constraint graphs for both the conventional and our techniques.

Figure 4.10 summarizes the test execution time, measured using performance monitors [36]. We measured three components of MTraceCheck's execution: (1) the execution time of the *original test*, (2) the execution time of our observability-enhancing code (*signature*

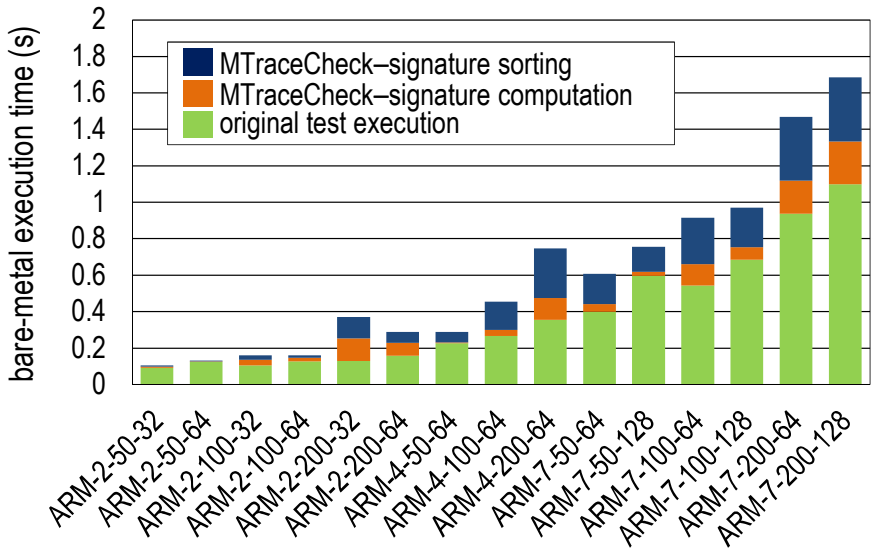


Figure 4.10: Test execution – MTraceCheck execution overhead. Our signature computation / sorting requires 22% / 38% of the original execution time on average, respectively.

computation), and (3) the time spent in sorting signatures (*signature sorting*).[2]

The *original test* takes 0.09–1.1 seconds to run 65,536 iterations. Our *signature computation* minimally increases the execution time: from as low as 1.5% (`ARM-2-50-64`) to up to 97.8% in an exceptional worst case (`ARM-2-200-32`). The lowest increase can be explained by noting that the test generated only 6.6 unique interleaving patterns, thus the branch predictor can almost perfectly predict branch directions for our instrumented code. On the contrary, `ARM-2-200-32` exhibits a wide variety of distinct interleaving patterns, thus the performance penalty of the branch mis-predictions becomes noticeable. Our *signature sorting* algorithm also contributes to increasing the execution time, ranging from 3.9% (`ARM-2-50-64`) to 93.5% (`ARM-2-200-32`). While not shown in the graph, we observed a 140% signature-sorting overhead in the 1 million-iteration version of `ARM-2-200-32`.

4.5.4 Intrusiveness

To quantify the intrusiveness of MTraceCheck, we measured the amount of memory accesses unrelated to the original test execution, as shown in Figure 4.11. Compared to previous work [96], where all loaded values were stored back to memory, our signature-based technique requires only 7% additional memory accesses on average, ranging from 3.9% (`ARM-2-100-64`) to 11.5% (`ARM-7-200-64`). The variation can be explained by noting that test configurations providing higher data contention, i.e., tests with more threads,

[2]We implemented the signature-sorting program by using a balanced binary tree, written in C. We ran this program on the primary core in the Cortex-A7 cluster, after test executions completed.

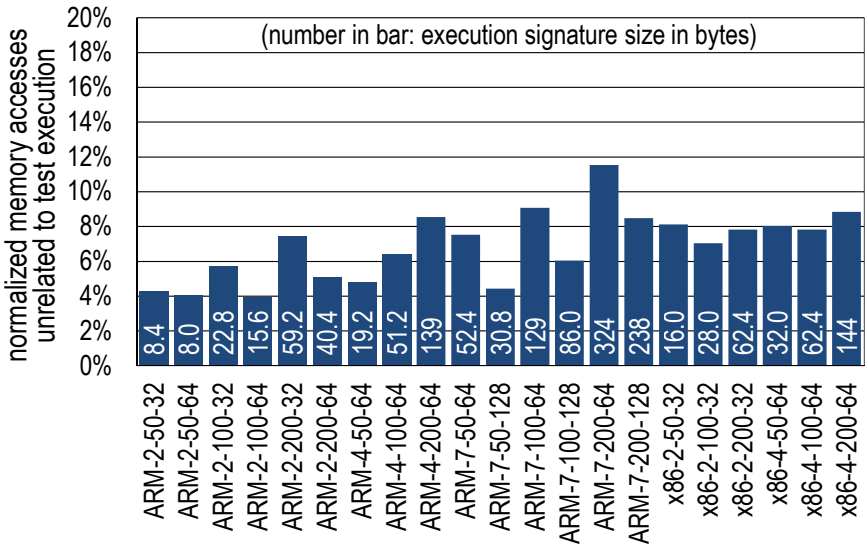


Figure 4.11: Intrusiveness of verification. Memory accesses unrelated to test execution are minimal compared to a register-flushing approach, only 7% on average.

more memory operations, and fewer shared locations, lead to a bigger per-thread signature footprint and in turn, to more data transferred due to signature collection.

Inside each bar of the figure, we report the average size of an execution signature. In low-contention configurations, the signature size is bounded by the register bit width. For instance, we need 16 bytes in `x86-2-50-32`. Note that in this configuration, the per-thread signature size merely exceeds 32 bits per thread, which leads to an execution signature size of 8.4 bytes in `ARM-2-50-32`. However, the instrumented code uses the entire 64 bits of a register, even when fewer are needed. In high-contention configurations, the gap between 32-bit and 64-bit registers becomes narrow, as the per-thread signature requires multiple words. An extreme case is illustrated by `ARM-7-200-64`, where the signature size is 324 bytes on average (46 bytes per thread).

A drawback of our solution is its increased code size. Figure 4.12 quantifies this aspect across various test configurations. We measured only the size of test routines, excluding initialization and signature sorting sections. The ratio of the size of the instrumented test to the original test ranges from 1.95 (`ARM-2-50-64`) to 8.16 (`ARM-7-200-64`). While this increase is conspicuous, the code size is still small enough to fit in the system's L1 instruction caches for all test configurations. For example, in `ARM-7-200-64`, the instrumented code is 189 KB; when divided by the number of threads, each core's code becomes 27 KB, fitting well in the 32 KB L1 instruction cache. Thus, the increased code size merely impacts locality aspects in instruction caches. A similar observation is drawn from Figure 4.10, where signature computation marginally increases the execution time.

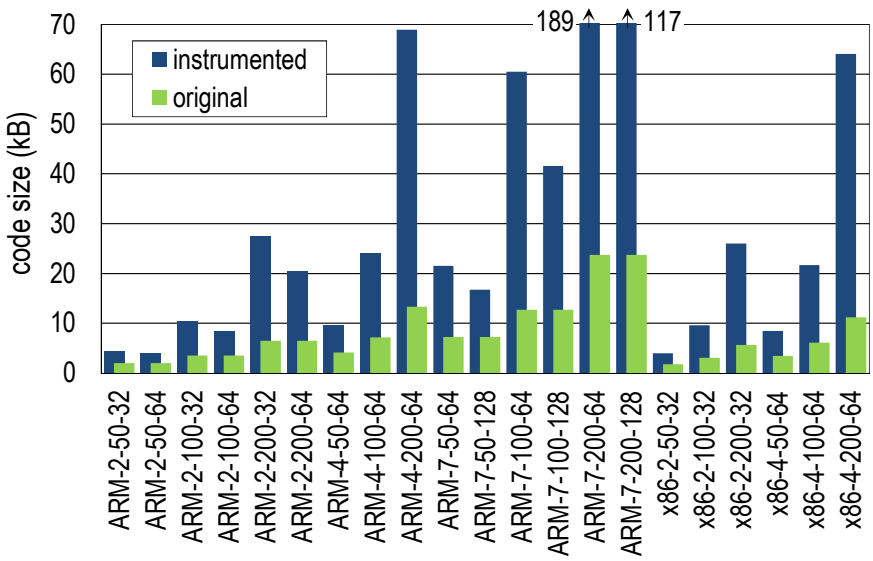


Figure 4.12: Code size comparison. Instrumented code is 3.7 times larger than its original counterpart, on average. However, all instrumented tests are still small enough to fit in the L1 caches.

4.6 Bug-Injection Case-Studies

We performed bug-injection experiments using the gem5 simulator [56]. Three real bugs, which have been recently reported in previous work [81, 131, 118], were chosen for injection. We confirm that these bugs had been fixed in the recent version of gem5 (tag `stable_2015_09_03`). To recreate each of the bugs, we searched the relevant revision from the gem5 repository [9] and reverted the change.

We configured gem5 for eight out-of-order x86 cores with a 4×2 mesh network and a MESI cache coherence protocol with directories located on the four mesh corners. For bugs 1 and 3, we purposefully calibrated the size and the associativity of the L1 data cache (1 KB with 2-way associativity) so as to intensify the effect of cache evictions under our small working set, while the rest of configuration is modeled after a recent version of the Intel Core i7 processor. We compiled our tests with the m5threads library to run simulations under gem5's syscall emulation mode.

4.6.1 Bug Descriptions

Bug 1 – load→load violation 1 (protocol issue). Bug 1 (fixed in June 2015) is modeled after "MESI,LQ+SM,Inv" described in [81]. This bug is a variant of the Peekaboo problem [185], where a load operation appears to be performed earlier than its preceding load operation. It is triggered when a cache receives an invalidation for a cache line that is currently

Table 4.3: Bug detection results

Bug	Test configuration	Bug detection results
1	`x86-4-50-8` (4 words/line)	1 test, 29 signatures
2	`x86-7-200-32` (16 words/line)	11 tests, 12 signatures
3	`x86-7-200-64` (4 words/line)	All tests (crash)

transitioning from shared state to modified state. When the cache receives the invalidation message, subsequent load operations to this cache line must be re-tried after handling the received invalidation. However, this bug prevents the cache from squashing the subsequent load operations, causing a load→load violation.

Bug 2 – load→load violation 2 (LSQ issue). Bug 2 (fixed in March 2014) is reported by two independent works: [131] and [81]. This bug is similar to bug 1 in its manifestation, but caused by an LSQ that does not invalidate subsequent loads upon the reception of an invalidation message.

Bug 3 – race in cache coherence protocol. Bug 3 (fixed in January 2011) is modeled after the "MESI bug 1" detailed in [118], which is also evaluated in [81]. This bug is triggered by a race between an L1 writeback message (PUTX) and a write-request message (GETX) from another L1.

4.6.2 Bug-Detection Results

For each bug, we deliberately chose the test configuration in Table 4.3 after analyzing the bug descriptions,[3] and generated 101 random tests. The iteration count was greatly reduced to 1,024 from 65,536 in Section 4.5, because of the much slower speed of gem5 simulations, compared to native executions.

We report our bug-detection results on the third column of Table 4.3. Overall, MTraceCheck successfully found all injected bugs. Note that bug 1 was detected by only 1 test (out of 101): in this test, we found that 29 signatures (out of 1,024 unique signatures) exhibited invalid memory-access interleavings. Bug 2 was easier to expose than bug 1; 10 tests revealed 1 invalid signature each, with one test revealing 2. The impact of bug 3 is much more dramatic, crashing all gem5 simulations with internal error messages (e.g., protocol deadlock and invalid transition).

[3] We were able to find bugs 2 and 3 without much effort in selecting test configurations. For the bug 1, we also tried the same seven-threaded random test, as well as a few handcrafted litmus tests, to no avail.

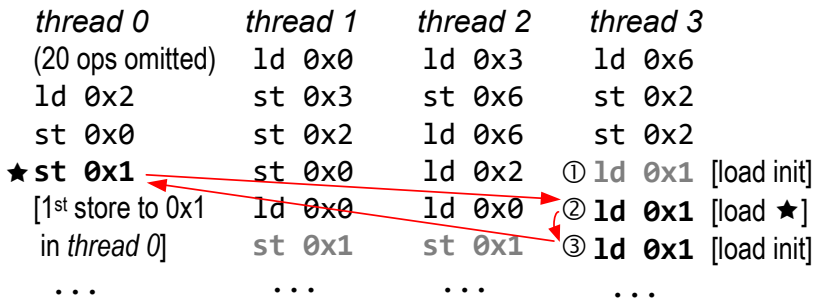


Figure 4.13: Detected load→load ordering violation. Loads ② and ③ show an invalid re-ordering due to bug 1.

Figure 4.13 provides a snippet of the test exposing bug 1. The first 20 memory operations of thread 0 are omitted for simplicity. The starred instruction ★ is the first store to memory address `0x1`. Both threads 1 and 2 perform a store operation to `0x1`, while thread 3 executes three consecutive load operations from `0x1`. Among these three loads, the first and the third read the initial value of the memory location, while the second reads the value from ★. From the second and third loaded values, we identify a cyclic dependency illustrated with red arrows in the figure: ★ happens before ② (reads-from), which should happen before ③ (load→load ordering), which in turn happens before ★ (from-read).

4.7 Insights and Discussions

We attribute the speedup of our collective graph checking (Figure 4.9) to two factors. Firstly, we notice that many constraint graphs can be validated immediately without modifying the previous topological sort. This situation is highlighted in the left-most bar (ARM) of Figure 4.14, where all constraint graphs beside the first do not require any re-sorting. `tsort` unwittingly places store operations prior to load operations since stores do not depend on any load operations in absence of memory barriers, as it is the case for our generated tests. Note that `tsort` is MCM-agnostic, so it still needs to check every new backward edge.

Secondly, for x86 tests, we also observe that a majority of constraint graphs are incrementally checked. The blue bars in Figure 4.14 show the percentage of such graphs, ranging from 82% (`x86-2-50-32`) to almost 100% (`x86-4-200-64`). For these graphs, we measure the average percentage of vertices affected by re-sorting in the line plotted in the figure: this percentage ranges from 21% (`x86-4-50-64`) to 78% (`x86-4-200-64`). It is the high incidence of re-sorting that negatively affected the violation checking performance boost shown in Figure 4.9.

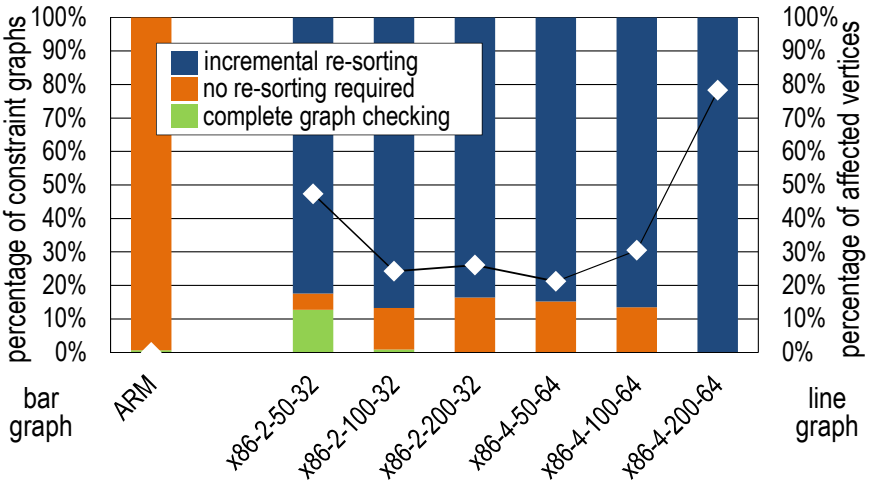


Figure 4.14: Breakdown of our collective graph checking. For ARM tests, most of graphs do not require any re-sorting. For x86 tests, up to 16% of the graphs can bypass re-sorting. Among the others, up to 78% of the vertices are affected.

Pruning invalid memory-access interleavings. Our code instrumentation (Section 4.3) makes a conservative assumption to support a wide range of MCMs in a single framework; each memory operation can be independently reordered, regardless of its preceding operations. This conservative assumption comes at the cost of increased signature and code sizes (Figures 4.11 and 4.12). To address these two drawbacks, we can leverage microarchitectural information when instrumenting observability-enhancing code (*static pruning*). For instance, the number of outstanding load and store operations are often bounded by the number of LSQ entries, etc. Using this additional information, we can reduce the number of options for loaded values (as in [181]), thus decreasing the sizes of instrumented code and signatures. In our real-system evaluations, however, we could not gather sufficient microarchitectural details to employ this optimization.

Moreover, in a strong MCM (e.g., SC, TSO), we can also apply a runtime technique to reduce signature size (*dynamic pruning*). At runtime, each thread would track the set of the recent store operations performed by other threads, computing a frontier of memory operations. Any value loaded from a store operation behind this frontier would be considered invalid. However, with this dynamic pruning, signature decoding would become more complicated as the length of signature would vary. Moreover, instrumented code size would increase further due to store operation tracking and frontier computation.

Store atomicity. Our proposed solution (Sections 4.3 and 4.4) makes no assumption on store atomicity (single-copy atomic, multiple-copy atomic, or non-multiple-copy atomic). However, our evaluation (Sections 4.5 through 4.6) does not include a single-copy atomic

system. We expect that constraint graphs for such systems would need additional dependency edges [48, 132]. Also, we expect that our collective checking would still perform better than conventional checking, but MTraceCheck's advantage would decrease due to larger re-sorting windows.

4.8 Related Work

Memory consistency validation. Section 2.2 provides a short introduction to various memory consistency validation solutions. As explained there, these solutions are generally classified into two groups: formal methods and testing-based methods. MTraceCheck improves on the prior work in the latter group by reducing the computation requirements of constraint-graph checking and by minimizing memory-access perturbation. Also, unlike many prior hardware-based solutions, MTraceCheck is a purely software-based approach that does not require any hardware support.

Test generation for memory consistency validation. As introduced in Section 2.2, various litmus tests have been developed and are widely used in memory consistency validation. When validating full systems, however, these litmus tests are often insufficient, as illustrated in [81]. Thus, constrained-random test generators (e.g., [96, 173, 167]) strive to validate corner-cases uncovered by the litmus tests.

In our evaluation, we only used a constrained-random generator; however, advanced test generation can be paired with our code instrumentation and checking techniques. For instance, program regularization [64] proposes to augment test programs with additional inter-thread synchronizations so as to simplify the frontier computation (as discussed in Section 4.7).

4.9 Summary

In this chapter, we presented MTraceCheck, a memory consistency validation framework. The proposed framework strives to efficiently validate a wide range of memory-access interleaving patterns observed in multicore systems by leveraging a novel, incremental approach; we analyze the similarity of the patterns across repetitive runs of a same test program, then decompose them to extract the differences among the runs. Only these incremental differences are then verified by the proposed graph checking algorithm. We also present a novel concept: memory-access interleaving signatures, which encapsulate observed memory-access interleavings into a compact integer value. This signature is used

to estimate the similarity across test runs so that we can approximately sour out analyses to entail minimal incremental effort. Our signature computation is also designed to minimally perturb the test execution, providing a viable solution to the limited-observability issue in post-silicon validation.

In our evaluation, we applied MTraceCheck to two real systems: a quad-core x86 desktop and an octa-core ARM single-board computer. We tested these systems with various configurations, consistently achieving a 5× speedup in the result-checking computation, due to topological sorting. In addition, our signature-based solution reduces the number of memory accesses required to store the execution traces; we measured a 93% reduction in the number of memory operations for this task alone.

So far, we addressed the problem of memory subsystem correctness at design time. The next chapter continues the discussion of memory subsystem correctness, this time through a patching solution that can recover from runtime errors.

CHAPTER 5

Patching Design Bugs in Memory Subsystems

This chapter presents a patching solution to bypass design bugs in memory subsystems. Memory subsystem components, including multiple levels of caches and multiple main-memory channels, are distributed across the entire chip and interact with each other in various ways. Thus, memory subsystems exhibit a wide range of functional behaviors, which cannot sometimes be foreseen, and should be verified and validated at design time. Unfortunately, some of these behaviors are buggy and escape into the final product.

To gain insights on the impact and breadth of this problem, we analyzed several product errata documents to understand errata bugs that cause incorrect results of memory operations. We found that a majority of these bugs manifest themselves only in very rare microarchitectural conditions. For instance, memory ordering and cache coherence bugs manifest only when some rare interactions occur among cores and caches, thus they often go unnoticed during verification and validation processes. However, after deployment (i.e., at runtime), these bugs may lead to critical system hangs or subtle incorrect program results. Thus, the system must be able to detect them and recover from them using a runtime approach.

To accurately detect bugs at runtime, ideally, we would need a system-level patch module that can probe numerous signals in the entire memory subsystem. Implementing such a global patch module is an almost impossible pursuit in large-scale systems on chip (SoCs), due to an enormous amount of signals that must be tapped. In this dissertation, we propose to *decompose a global patch module into smaller distributed modules*, where each of the modules is attached to a corresponding node (i.e., a core and the co-located L1 data cache) in the system. Each module monitors microarchitectural events occurring in their corresponding node.

Note that our distributed patch module has no capability of observing microarchitectural events at other components (the other nodes, lower level caches, on-chip networks, etc.). It, however, strives to grasp activities happening in these components by observing

cache eviction messages arriving at the node. From these messages, it can catch bug-prone data races that may alter the results of memory operations at the core. In other words, *these messages provide a summary of outside activities*, which allow us to attain a level of monitoring capabilities, which is similar to a global patch module. Each of the distributed patch modules is implemented through simple finite state machines (FSMs), which are designed to catch sequences of microarchitectural events that may lead to bugs. When one such sequence is identified at runtime, our proposed solution, called μMemPatch, takes action to bypass the bug manifestation. To this end, it utilizes four fine-grained, microarchitectural bug-elusion methods.

5.1 Motivation

Industry practices to patch design bugs are (1) microcode patching and (2) software editing or re-compilation. These solutions, however, may not be the best approaches to address memory subsystem bugs. They are agnostic to microarchitectural states that are prone to bugs. Microcode patching is applied in an early stage of the microprocessor pipeline, while memory subsystem bugs are mostly related to the back-end pipeline stages. Similarly, the software editing or re-compilation methods are conducted on software – it does not account for any runtime microarchitectural states.

A few programmable hardware patching solutions have been proposed recently [150, 177, 199]. These solutions strive to accurately detect bugs at runtime by monitoring control signals across the entire chip [150, 177] or cache coherence events [199]. (Refer to Section 2.2 for details.) They strive to minimize their performance impact by observing specific signals and flagging only the few events that may lead to a bug manifestation. However, these solutions are ineffective when addressing complex bugs involving a past *sequence of microarchitectural events* – such as those related to memory ordering and cache coherence. Indeed, they can only identify patterns of control signals or combinations of cache coherence states and transitions.

Inefficiency of prior patching methods. To illustrate the contributions of μMemPatch, we illustrate a typical memory subsystem bug in Figure 5.1 and explain why prior patching methods are inefficient in handling it. The bug is a "read-after-read hazard" in the ARM Cortex-A9 processor [35, 39], which manifests as an incorrect ordering between two load operations to the same address. When deploying microcode patching to fix this bug, one should insert a fence operation right after (A), so that (B) cannot be speculatively executed ahead of (A). However, to do so, one must first know which loads are prone to this bug,

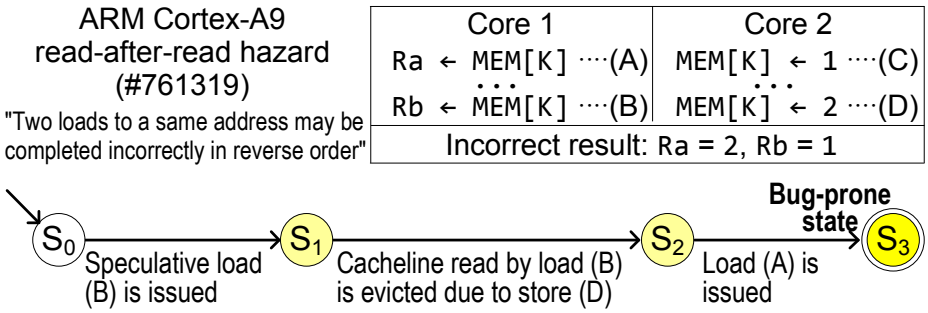


Figure 5.1: ARM's read-after-read hazard example. Top – When the load (B) is executed ahead of the preceding load (A) to the same address K, (A) might read a more recent value than (B). This reordering could occur for many reasons, e.g., (A)'s address is resolved later than (B). Bottom – The hazard can be captured by the FSM monitoring microarchitectural events.

by comparing load addresses at the fetch or decode stages – a challenging feat, since many load addresses are not even available in early pipeline stages. Alternatively, one could conservatively insert a fence operation whenever an unresolved load address is encountered at these stages, which may overly serialize memory operations. Note that the performance cost of a fence operation is high, about 15% over an unpatched program with no fence operations inserted [193]. In practice, the manufacturer's suggestions are to either add a fence as mentioned earlier, or use an exclusive load instruction (LDREX) instead of a normal one (LDR). All these solutions restrict unrelated memory operations to a great degree. Prior hardware-based patching solutions [150, 177] rely on existing control signals to identify buggy microarchitectural states. However, it is possible that not all events involved in a microarchitectural event sequence leading to a bug have dedicated control signals. For instance, to detect the read-after-read bug, the load address must be compared against the addresses of subsequent load operations. In the absence of such comparison logic, those prior solutions could not detect this bug.

5.2 μMemPatch Overview

Figure 5.2 shows how μMemPatch operates in three steps and a system diagram with μMemPatch's hardware additions.

Step 1: Root-cause analysis. To understand the characteristics of memory subsystem bugs, we first conduct a comprehensive study to collect design flaws described in product errata documents published by major microprocessor vendors. In addition, we also investigate open-source multicore design implementations in order to further understand the exact

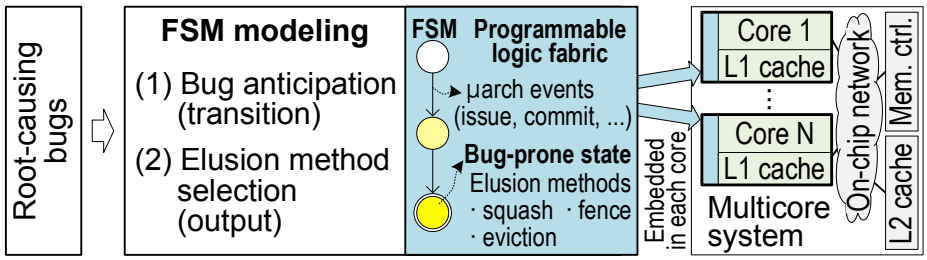


Figure 5.2: μMemPatch overview. μMemPatch strives to overcome design flaws in multicore memory subsystems at runtime. It first conducts an offline analysis to root cause a bug, and generates a finite state machine (FSM) that can detect event sequences leading to it. The FSM is then deployed in a programmable logic block: when it identifies one such sequence, it initiates an action to circumvent the bug.

root causes of the bugs.

Step 2: Bug detection and elusion. The intuitions obtained from step 1 lead us to develop our novel patching solution that captures a sequence of bug-triggering microarchitectural events. We model each bug-triggering sequence as a finite state machine (FSM), where the transition corresponds to a microarchitectural event and the output corresponds to the choice of a bug-elusion action. We investigate a few bug-elusion methods and discuss trade-offs therein.

Step 3: Embedding patch modules to a programmable logic. The FSMs are embedded on a programmable logic attached to each core. We measure the overhead of the programmable logic to catch various bugs.

Note that, in designing the bug-anticipation FSM, we focus mainly on microarchitectural events in the *back-end* stages of out-of-order processors, and those at the interface between a core and its L1 data cache. This design decision is derived from observations in our errata investigation, that led us to conclude that a majority of bugs are caused by rare combinations of *speculative execution* and *data races among cores*. The speculative execution is captured by microarchitectural events in the back-end stages, and many data races can be monitored at the core-cache interface. This observation allows us to use a distributed patching approach: the same FSM is deployed next to each core, and it monitors only signals local to its core. For simplicity, we evaluate μMemPatch only in homogeneous multicore systems in this dissertation, although it can be tailored for heterogeneous multicore systems.

5.3 Classifying Memory Subsystem Bugs

We conducted a comprehensive literature survey to understand memory subsystem bugs that slip through the verification and validation processes. We considered product errata documents available online, and analyzed each erratum related to the product's memory subsystem. Specifically, we examined errata for multicore products from three companies: ARM [37, 41, 35, 39, 40, 38, 42, 43, 44, 45, 46, 47], Intel [104, 106, 107, 105, 103, 101, 100], and AMD [27, 29, 28, 30, 31, 33, 32]. Among these companies, ARM provides much more detailed descriptions, while Intel and AMD offer brief summaries. Table 5.1 classifies bugs based on two criteria: root cause (by row) and symptom (using colors). When multiple root causes are involved (e.g., prefetch and address translation), we consider only the primary one.

5.3.1 Root Causes of Memory Subsystem Bugs

We identify 10 distinct bug root causes, listed in the first column of Table 5.1. Root causes in row 1 through 5 are mostly core-side, while those in row 7 through 9 are cache-side. Row 6 reports a root cause that relates to both core and cache. In summary, we observe two major bug causes: (1) deeply speculative executions in modern out-of-order cores and (2) data races across cores/caches.

Core-side bugs. *Speculative execution* is one of the major sources of memory subsystem bugs. Instructions executed out-of-order must be squashed and re-executed if the speculation could lead to incorrect results. A representative bug in this group is the read-after-read hazard in Figure 5.1. While this bug has been fixed in recent ARM Cortex processors, those processors are still subject to similar buggy behaviors. For instance, bug #777771 in ARM Cortex-A15 reports a read-after-read hazard when a speculative load is an *unaligned memory access* crossing a cacheline boundary (64 bytes). Intel processors are also prone to a similar bug (Intel Core2 Quad AV76). Furthermore, #826978 in Cortex-A57 and #826969 in Cortex-A15 report yet another read-after-read hazard when crossing a data-bus boundary (16 bytes). These escaped bugs are caused by an improper handling of *multiple micro-operations*.

Address translation may also be incorrect in the rare circumstance that a memory access crosses the 4KB page boundary (ARM Cortex-A15 #775620). Recent ARM Cortex processors support TLB maintenance instructions, which may cause an indefinite stall in the presence of a continuous stream of hardware prefetch requests from another core (ARM Cortex-A72 #838569). *Hardware prefetchers* can induce an occasional deadlock (ARM

Table 5.1: Classifications of memory subsystem bugs

	Root cause	Erratum ID	Symptoms
Core-side bugs	1. Speculative executions	AMD-147, AMD-530, ARM-761319, ARM-826974, ARM-836019	Deadlock, Corrupted data, Starvation, Wrong order, Other, Either
	2. Unaligned memory accesses	Intel-AV76, AMD-92, AMD-105, AMD-551, ARM-854671, ARM-777771, ARM-811819, ARM-768277, ARM-817722	
	3. Multiple micro-operations	ARM-826969, ARM-855830, ARM-826978, ARM-828023	
	4. Address translations	ARM-775620, ARM-813469, ARM-798797, ARM-838569	
	5. Prefetches	Intel-SKL055, ARM-745469, ARM-571620, ARM-751473	
Both core and cache	6. Micro-architectural hazards	ARM-823274, ARM-761320, ARM-775619, ARM-801819, ARM-836969, ARM-851024, ARM-829070, ARM-843419, ARM-823273, ARM-834921, ARM-851023, ARM-851022, ARM-853676, ARM-770356	
Cache-side bugs	7. Cache coherence interactions	ARM-814169, ARM-775621, ARM-798870, ARM-763126, ARM-845920, ARM-821523, ARM-819472, ARM-824069, ARM-855873, ARM-855872, ARM-836870, ARM-806969, ARM-853971, ARM-848970, ARM-854172, ARM-854222	
	8. Unconventional caching types	Intel-BV24, Intel-AK85, Intel-AV3, AMD-81, AMD-97, ARM-761273, ARM-791671, ARM-791620, ARM-793369, ARM-754327, ARM-794074, ARM-845369, ARM-799271, ARM-774769, ARM-773023, ARM-852421, ARM-832071, ARM-822769, ARM-827971, ARM-872696, ARM-832075, ARM-817170, ARM-826975, ARM-858072	
	9. Cache maintenance instructions	Intel-BJ21, AMD-144, AMD-461, AMD-579, AMD-691, ARM-571771, ARM-827671, ARM-826319, ARM-827496, ARM-813420, ARM-853871, ARM-814670, ARM-854173, ARM-855423	
	10. Others	AMD-372, ARM-845719	

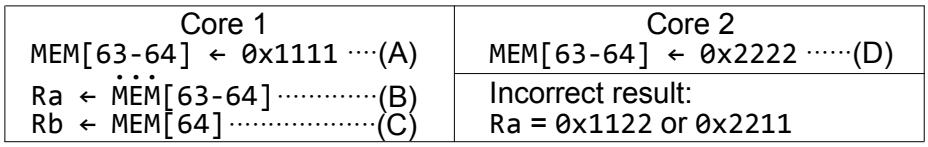


Figure 5.3: Unaligned writes in gem5 may not produce the intended result.

Cortex-A5 #745469) or corrupt data (ARM Cortex-A9 #751473). Also, *prefetch instructions* are prone to memory ordering violations (Intel Core SKL055).

Cache-side bugs. *Cache coherence* bugs are common, and often related to a chip's underlying interconnect (e.g., ARM's cache coherent interconnect) among heterogeneous components. Moreover, *unconventional caching* is another major source of bugs. For instance, when data written by a core with ARM's write streaming mode is treated as conventional cacheable data by another core, memory ordering rules may be violated (ARM Cortex-A17 #832071). A similar buggy behavior can occur with Intel's non-temporal data accesses (Intel Core2 Quad AK85), and *cache maintenance instructions* (e.g., DCCMVAC in ARM) may malfunction when they are used in the presence of data races (ARM Cortex-A15 #827671).

Miscellaneous. *Microarchitectural hazards* are also common on both core and cache. For instance, a conditional load or store operation may go wrong under rare timing conditions in the presence of a concurrent floating-point operation (ARM Cortex-A7 #823274). On the cache side, an exclusive memory operation may fail to guarantee exclusiveness when the same cacheline is accessed also by a stream of stores (ARM Cortex-A73 #853676). Finally, we also noticed a couple of bugs that do not belong to any other group: an uncommon combination between a DRAM controller setting and a northbridge's operating frequency (AMD Family 10h processors #372) or a processor's mode change (ARM Cortex-A53 #845719).

5.3.2 Symptoms of Memory Subsystem Bugs

Various symptoms have been reported in product errata. Among them, a significant portion leads to unexpected system hangs (*deadlocks*). For instance, an improper handling of a snoop message might result in a deadlock (ARM Cortex-A53 #821523). Many bugs associated with data races among caches also lead to deadlocks, as well as some core-side bugs. Other bugs lead to *starvation*, so that some memory operations never make forward progress, often the result of frequent interrupts from other activities sharing the same

hardware resources (ARM Cortex-A75 #770356). Another major symptom is *data corruptions*. Some are caused by corner-cases of the cache coherence protocol (ARM Cortex-A57 #853971), others by unconventional caching (AMD Athlon 64 #81). We also noted several *memory ordering violations*: beside the read-after-read hazard in Figure 5.1, we learned of more subtle bugs, for instance, atomic operations that fail to guarantee atomicity (AMD Family 14h processors #530). Some recent processors have a feature that nullifies fence operations: unfortunately such feature may malfunction when a cache-maintenance operation is in progress (ARM Cortex-A57 #814670).

5.3.3 Memory Subsystem Bugs in Open-Source Designs

In addition to the errata bugs reported in Table 5.1, we investigated three open-source multicore designs: (1) an octa-core RISC-V RTL design with Rocket in-order cores [14], (2) a quad-core RISC-V RTL design with Boom out-of-order cores [4], and (3) an octa-core ARM out-of-order model in gem5 [56]. We conducted a bug-hunting campaign searching for memory ordering violations using open-source memory consistency validation tools [153, 120] and in-house litmus tests designed to trigger bugs like those in Table 5.1. Through this extensive process, we learned that: i) the open-source designs are protected against some errata-like bugs. For instance, gem5 includes specific provisions to detect and avoid read-after-read hazards, while Boom has a compile-time knob to select whether or not to squash instructions affected by the hazard. ii) In addition, many sophisticated performance features, often connected to product errata bugs, do not exist in open-source designs (e.g., write streaming mode, cache maintenance operations). Yet, we noted an interesting buggy behavior in gem5, illustrated in Figure 5.3 with a two-threaded test program. Core 1 performs an unaligned store (A) (addresses 63 and 64 belong to distinct 64-byte cache-lines), followed by two loads (B) and (C). Core 2 performs another unaligned store (D) to the same memory location as (A). We observed a few occurrences of two incorrect results, shown on the bottom right. This behavior was noted while evaluating if bug 3 from Table 5.3 was present in gem5. Note that those incorrect results do not actually violate cache coherence invariants [185], as cache coherence is only concerned with individual byte-long memory locations.

5.4 Detecting Bug-Prone Microarchitectural State

We propose a *bug-anticipation module* that captures a sequence of microarchitectural events likely to trigger bugs. The module strives to *identify a sequence of events leading to a bug manifestation, before the (micro-)architectural state is altered*. In developing it, we

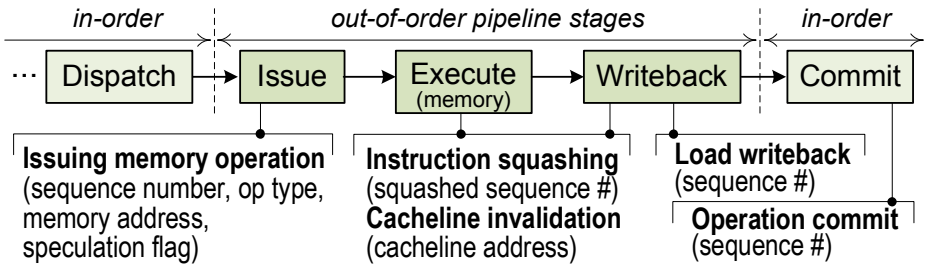


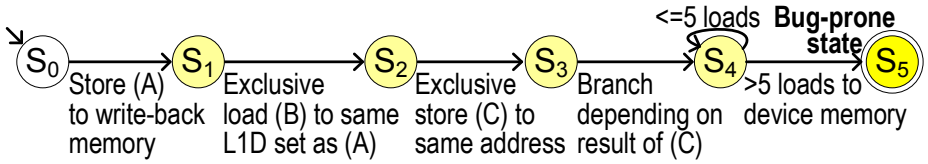
Figure 5.4: Microarchitectural events of interest. Due to practical limitations in monitoring signals, we probe only some of the events in back-end pipeline stages, in addition to cache invalidation messages.

observed that many memory subsystem bugs occur in rare conditions, frequently related to back-end execution stages in microprocessors. Thus, the bug-anticipation module continuously monitors selected back-end microarchitectural events. It also tracks *sequences* of events, because many bugs involve multiple events over time. It implements finite state machines (FSMs) to achieve its goals, because they are intuitive and easy to implement in programmable logic. For example, the bottom part of Figure 5.1 illustrates the FSM capturing the read-after-read hazard. The rest of this section discusses the design of this module.

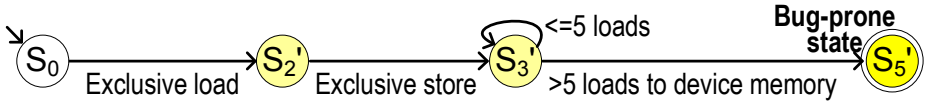
5.4.1 Collecting Microarchitectural Events

Ideally, at runtime, we would like to have full observability into every internal signal, an almost impossible feat in practice. Thus, mindful of the limited access to signals, the module focuses only on a handful of microarchitectural events that are most likely to support the detection of a majority of memory subsystem bugs; those tend to be located in the back-end stages of the pipeline, from issue to commit.

Figure 5.4 provides a schematic of the microarchitectural events that we collect at runtime. Without loss of generality, we illustrate our solution on typical pipeline stages of an out-of-order processor [183, 56]. Our event collection occurs mostly in the issue, execute and writeback stages, as shown in the figure, since those allow the out-of-order execution of micro-operations (μops). Specifically, i) when a memory μop is issued, we collect its dynamic sequence number (i.e., unique identifier). We also gather some additional information: the operation type (load, store, prefetch, etc.), the memory address, and a speculation flag, indicating whether the μop is issued in advance of preceding operations. In addition, ii) while a memory operation is in flight (i.e., execution and writeback stages), we collect instruction-squashing events that nullify any of its μops, as well as we mark μops affected by a cacheline invalidation (a.k.a. snoop) message from the L1 cache controller. iii) We



(a) FSM to catch ARM Cortex-A57 832075



(b) Approximated FSM to detect the same bug as (a)

Figure 5.5: FSM approximation. Long sequences and complex conditions of microarchitectural events in the FSM of part (a) can be simplified as shown in part (b), leading to a negligible risk of false detection.

also collect load-writeback events, so that we can implement one of our bug-elusion solutions (see Section 5.5.2) and iv) we monitor μop commit events. The internal structure of the bug-anticipation module is highly dependent on the bugs to be identified. For those that we evaluated in our experiments, the anticipation module is somewhat similar to a load-store queue, because it observes events related to memory operations and to the core-cache interface.

μMemPatch signal probes are limited and fixed at design time, thus potentially limiting its coverage to some degree. There are indeed bugs triggered by events at the fetch stage, or in the instruction cache, or in lower-level caches, none of which are monitored by our solution. For instance, a gem5 read-modify-write (RMW) deadlock bug [8] involves an instruction fetch to a cacheline locked by an RMW data access, which cannot be captured by our solution.

5.4.2 Reducing the Area Overhead of the Bug-Anticipation Module

The area footprint of the bug-anticipation module varies with the microarchitectural events to be identified, the length of the sequence, etc. As we will show in Section 5.6.4, the module's implementation may require a significant area, due to the inherent inefficiency of programmable logic, as compared to standard cells. To limit this overhead, we designed two approximation techniques:

FSM sequence simplification. While we noted that most bugs can be identified with sequences of just a few FSM states ($\sim$5), others entail relatively long sequences and very detailed conditions. Figure 5.5a shows an FSM designed to catch a deadlock condition reported in ARM Cortex-A57 #832075. In our implementation, we simplify the FSM as

shown in Figure 5.5b, that is, we trace only a pair of load and store operations ($S_0 \rightarrow S'_2 \rightarrow S'_3$) and device loads ($S'_3 \rightarrow S'_5$), dropping the address and register index comparisons. While this simplification may lead to false detection, in practice this risk is fairly negligible in real-world applications.

Memory address truncation and hashing. A large portion of the bug-anticipation module's area is occupied by address-related logic. Memory addresses are relatively wide (e.g., 64 bits in RISC-V ISA), while most other signals have very few bits. To limit area overhead, we apply address truncation, considering only the lower 20 bits of each address. We can also apply a simple address hashing, generating 8-bit hash values by XORing eight 8-bit chunks of the 64-bit addresses.

5.5 Microarchitectural Bug-Elusion

In μMemPatch, we consider and compare a number of methods to bypass memory subsystem bugs. The goal is to recover from a bug at a much finer granularity than conventional checkpointing and rollback methods [186], leading to minimal performance impact. Our bug-elusion methods entail a small design modification in the back-end stages of the microprocessor pipeline. Below, we first present two basic methods based on instruction squashing and re-execution, then propose two more advanced solutions that restrict either the speculation on memory operations or data races, and finally discuss the modifications required to the back-end stages. Figure 5.6 will guide us through the presentation by illustrating how each elusion method can be applied to overcome the read-after-read hazard discussed in Figure 5.1.

5.5.1 *ImmSq* & *DelSq*: Instruction Squash and Re-execution

We evaluate two variants of squash and re-execution: immediate and delayed squash. In the former solution, we squash the instruction as soon as the bug-anticipation module identifies it as buggy. Figure 5.6a shows the action taken when the FSM in Figure 5.1 enters the bug-prone state S_3. The second variant postpones the squash until the instruction arrives at the commit stage, as illustrated in Figure 5.6b. The benefits of a delayed squash are multiple: i) it allows more reliable timing, as the detection at the bug-anticipation module may not be readily available if it operates at a lower frequency; ii) it ensures forward progress in adversarial bug-triggering situations. Indeed, some microarchitectural implementations do not allow a squash and a commit in the same clock cycle. In those cases, an immediate squash could block commit events indefinitely.

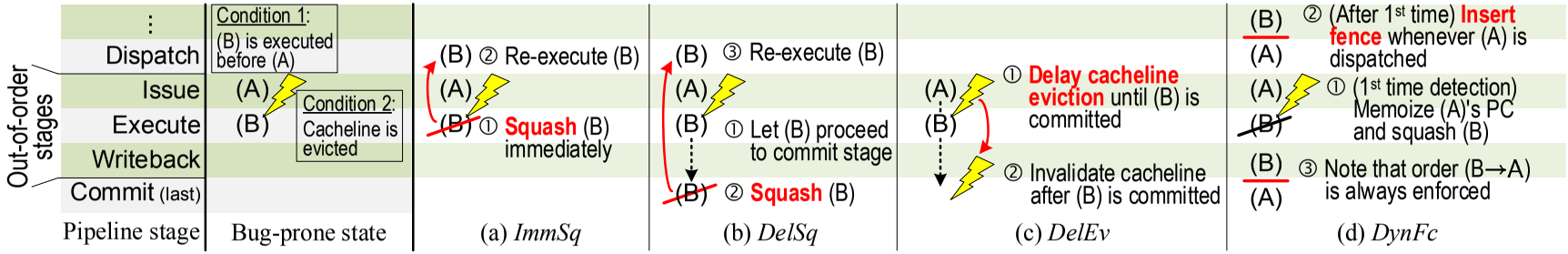


Figure 5.6: Microarchitectural bug-elusion methods. For the read-after-read hazard of Figure 5.1, with anticipation conditions illustrated on the left, we can apply four different bug-elusion methods, shown in (a) through (d). The red color highlights patching actions.

Table 5.2: Elusion methods comparison

Elusion method	Coverage	Key advantage
ImmSq – Immediate squash	All	Immediate recovery
DelSq – Delayed squash	All	Delayed detection
DelEv – Delayed cache eviction	Data race	Minimal impact
DynFc – Dynamic fence insertion	Speculation	Strengthened ordering

5.5.2 *DelEv*: Limiting Data Races by Delaying Cache Eviction

To limit data races, cache coherence mechanisms can be designed so that cache evictions can be delayed. Inspired by [172], we notice that a cacheline invalidation message can be delayed until all speculative loads to that cacheline are completed. In Figure 5.6c, the invalidation message is not processed until both loads (A) and (B) reach the commit stage. Because of this delayed invalidation, (A) retrieves the same data as (B), and no ordering violation will occur. The delayed cache eviction method can only be applied to bugs involving cache coherence interactions among cores. Note that its implementation should be carefully verified to ensure the correctness of the modified coherence interactions. While developing this method, we encountered deadlocks from delayed evictions, which could be avoided via *ImmSq*. Specifically, to prevent deadlocks, we first check if speculative loads have reached the writeback stage, in which case the eviction is no longer delayed. We also check the μop's sequence number tied to each eviction to avoid an excessive delay.

5.5.3 *DynFc*: Limiting Speculation by Dynamic Fence Insertion

Dynamic fence insertion enforces stronger ordering between memory operations. This method is very effective for bugs related to speculation, such as the read-after-read hazard. Unlike conventional fence insertion, we propose an advanced fence insertion that is readily paired with our bug-anticipation module. Specifically, we use a small lookup table to memoize the program counters (PC) of buggy instructions, previously identified during the same program execution. As shown in Figure 5.6d, this technique uses one of the squash methods to recover from the first occurrence of a bug. At that time, it also records the PC of the operation in the memoization table (PC of (A) with reference to the figure). Afterward, when the core dispatches the same instruction, it consults the memoization table for the instruction's PC: if found, it dynamically inserts a fence instruction right after the flagged instruction.

Table 5.3: Bugs modeled in our experimental evaluation

No	Bug description	Erratum ID
1	Read-after-read (RAR) hazard	ARM Cortex-A9 #761319
2	RAR on unaligned access type 1	ARM Cortex-A15 #777771
3	RAR on unaligned access type 2	(same as above)
4	RAR on multiple micro-ops type 1	ARM Cortex-A57 #826978
5	RAR on multiple micro-ops type 2	ARM Cortex-A57 #828023
6	Speculation on prefetched data	Intel 6th-gen Core SKL055
7	Snoop during exclusive access	ARM Cortex-A53 #819472
8	Snoop during cache clean	ARM Cortex-A15 #827671
9	Store crossing page boundary	ARM Cortex-A15 #813469
10	Device loads after exclusive access	ARM Cortex-A57 #832075
11	Disallowed address on cache flush	Intel 2nd-gen Core BJ21

5.5.4 Bug-Elusion Implementation

The two squashing methods require minimal modifications to the hardware: squashing and re-execution is already implemented in modern out-of-order cores [56, 4], so we simply add a trigger condition. Delayed cache eviction is implemented at the interface between the core and L1 data cache controller. Specifically, it needs to track the sequence number of μops that are associated with delayed evictions. Dynamic fence insertion needs a content-addressable memory (CAM) for PC memoization. It also requires the insertion of a fence instruction – easily done by piggybacking the effect of a fence instruction on the memory operation (e.g., asserting the fence flag at the dispatch stage). Note that these modifications can be applied at design time, and selectively activated based on the outcome of the bug-anticipation module. Table 5.2 summarizes our bug-elusion methods based on the extent of the coverage and key advantage they provide.

Note that implementing the CAM structure may be challenging due to its power overhead and slow speed. To address these challenges, a Bloom filter can be used instead of the CAM structure.

5.6 Experimental Evaluation

5.6.1 Experimental Setup

We evaluated μMemPatch in the gem5 full-system simulator [56]. We created an octa-core system with a 4×2 mesh network and a directory-based MOESI cache coherence protocol. Each core is configured with a private 32KB L1 instruction and data caches (set associativity = 8), and a shared 256KB L2 cache (set associativity = 4). We ran our

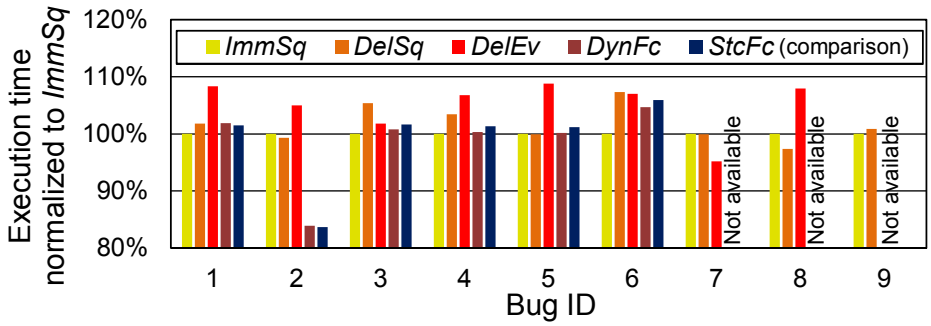


Figure 5.7: Micro-benchmark suite evaluation: All benchmarks complete correctly, even in presence of the bugs reported on the horizontal axis. The bars compare the performance of the test when using each of the elusion methods, normalized to the immediate squash (*ImmSq*). For comparison, we also provide the performance of a solution (*StcFc*) where a fence is statically inserted in the proper location of the code to avoid the bug.

simulations using the syscall emulation mode with the m5threads library.

Table 5.3 provides the list of bugs we evaluated. Each bug is modeled after the erratum listed on the third column. Bug 3 is also illustrated in Figure 5.3. Bug 2 is similar to bug 3, but with a different address layout: loads (B) and (C) are swapped and store (D) accesses only the lower byte. While we tried to faithfully model the bugs by following the descriptions provided, our bugs are unlikely to be identical to the actual bugs in commercial processors, due to lack of detailed information. In particular, we believe that our bug 6 is more conservative than the actual bug. Similarly, bug 9 is also simplified to check only the addresses of store operations while ignoring TLB-related activities. Finally, we could not simulate bugs 10 and 11 on gem5 because the simulator does not allow speculation between exclusive and uncacheable accesses (bug 10) and it does not implement the related instruction (bug 11).

5.6.2 Micro-Benchmark Results

We first evaluated the coverage and performance impact of our solution on a set of micro-benchmarks, of our own creation. The tests are written for the ARMv7 ISA, and each test is tailored to trigger one bug from Table 5.3. Our micro-benchmarks are structured similarly to litmus tests [26], with the difference that they are fine-tuned to trigger bugs much more frequently. Each test is single-threaded (bugs 6 and 9) or 2-threaded (bugs 1–5, 7 and 8). They comprise an initialization routine and a test loop, with the test loop repeated 1,024 times. Each test loop iteration is synchronized across participating threads using a spinlock variable. We ran all these micro-benchmarks in our setup; the bug-anticipation unit was always successful in predicting a bug manifestation, and used one of the four

elusion methods to bypass the bug. First, we note that, in all cases, the test completed correctly, providing the correct test results. Figure 5.7 compares the performance of the bug-elusion methods, normalized to *ImmSq*. Note that *DelEv* and *DynFc* are applicable only to some of the bugs, as discussed earlier. Comparing the two squash variants, *ImmSq* performs better than *DelSq* for most bugs, except bugs 2 and 8. *DelEv* underperforms *ImmSq* in most cases, except for bug 7. *DynFc* works especially well for bug 2 – we believe the runtime savings here originate from reduced cache conflicts.

In the plot, we also report the runtime of a solution using a static fence instrumentation (*StcFc*), that is, the micro-benchmark code is instrumented before execution by inserting one fence operation into the test loop, so to avoid the bug. This solution should be very efficient, although impractical in most applications, but very small programs, where one could afford the engineering time required to analyze the code extremely carefully. From the plot, we note that *StcFc* is no better than *DynFc*, despite the much larger software engineering investment, likely because the bug is triggered frequently in our micro-benchmark suite.

5.6.3 SPLASH-2 Results

We also evaluated SPLASH-2 benchmark workloads [204] for the x86 and ARMv7 ISAs. Note that we encountered a few unexpected errors when running some of the workloads with ARMv7 – in those cases, we considered only bugs triggered before the simulation terminated. We excluded benchmarks *fmm* and *water-spatial* from consideration, because they were both terminated prematurely within a few minutes of simulation in the x86 ISA. Table 5.4 reports μMemPatch's ability to detect and bypass memory subsystems bugs, while Figure 5.8 presents the performance impact evaluation. In presenting these results, we considered three alternative solutions: i) *fence – static all*, where a fence instruction is added after each load for the entire code. This solution conservatively limits memory re-orderings to avoid speculation-related bugs. ii) *fence – static pthread*, where fences are instrumented similarly to "static all", but only within the m5threads (pthread) library code. This solution is more reasonable since all SPLASH-2 workloads heavily rely on the pthread library for synchronization. Finally, we also implemented a iii) *fence – ideal* solution, which uses an oracle approach that cannot be deployed in practice, but it represents a lower bound on the number of fence instructions that should be executed to overcome a bug. This solution profiles the dynamic execution of a program binary and inserts a fence only in locations that would cause the bug to manifest. We only use it to provide a lower bound in the performance impact of the static-fence solutions presented.

In Table 5.4, we report all the bug/benchmark combinations tested on gem5. For each

Table 5.4: Bug patching coverage on SPLASH-2 workloads

Workload	Bug1	Bug2	Bug3	Bug4	Bug5	Bug6	Bug7	Bug8	Bug9
barnes	✓⊕	–	–	–	–	✓⊕	✓	–	–
cholesky	✓⊕	–	–	–	–	✓⊕	–	–	–
fft	✓⊕⊙	–	–	–	–	✓⊕	✓	–	–
lu (con)	✓⊕	–	–	–	–	✓⊕	–	–	–
lu (non)	✓⊕⊙	–	–	–	–	✓⊕	✓	–	–
ocean (con)	✓⊕	–	–	–	–	✓⊕	✓	–	–
ocean (non)	✓⊕	–	–	–	–	✓⊕	✓	–	–
radiosity	✓⊕	✓⊕	✓⊕	–	✓⊕	–	✓	–	✓
radix	✓⊕⊙	–	–	–	–	✓⊕	✓	–	–
raytrace	✓⊕⊙	–	–	–	–	✓⊕	✓	–	–
volrend	✓⊕	–	–	–	–	✓⊕	✓	–	–
water-nsquared	✓⊕	–	–	–	–	✓⊕	✓	–	–

✓: μMemPatch ⊕: *fence – static all* ⊙: *fence – static pthread*

of those, we indicate which of the solutions was successful in bypassing each occurrence of the bug and complete the benchmark execution correctly. Bug 1 is observed in most workloads in both x86 and ARM ISAs (except for ARM *cholesky*, which terminates due to an error). Note that bug 1 has been fixed in gem5 as described in Section 5.3.3; we reverted this fix for the sake of our evaluation. Bug 6 is observed mostly in the ARM ISA for most benchmarks as well as in x86 *cholesky*. These two bugs are triggered much more frequently than the other bugs. Bug 7 is also detected in various workloads in the ARM ISA, but its triggering frequency is very low – up to several tens of times over the entire execution (million to billion cycles). The other bugs are observed only in *radiosity* (bugs 2, 3, 5 and 9) or not observed at all (bugs 4 and 8). In the table, a check mark (✓) indicates that μMemPatch overcomes the bug, i.e., the bug-anticipation module identified the sequence of events corresponding to the bug and correctly patched the execution. Note that *fence – static all* provides good coverage, although it is likely to have a high performance impact, while *fence – static pthread* is more efficient, but presents very low coverage. *fence – ideal* is not shown since it would be impossible to predict the specific dynamic execution; if it could be done, it would cover all speculation-related bugs by construction.

Figure 5.8 presents the performance impact of μMemPatch's four elusion methods, and compares it against the fence insertion baselines we outlined. Here, we present results for bug 1 on x86 and for bug 6 on ARM, since we have significant manifestation data for those bugs. Note how μMemPatch presents minimal performance impact, and performs almost always significantly better than the fence insertion techniques including the oracle fence solution. For bug 1, all bug-elusion methods can bypass the bug with less than

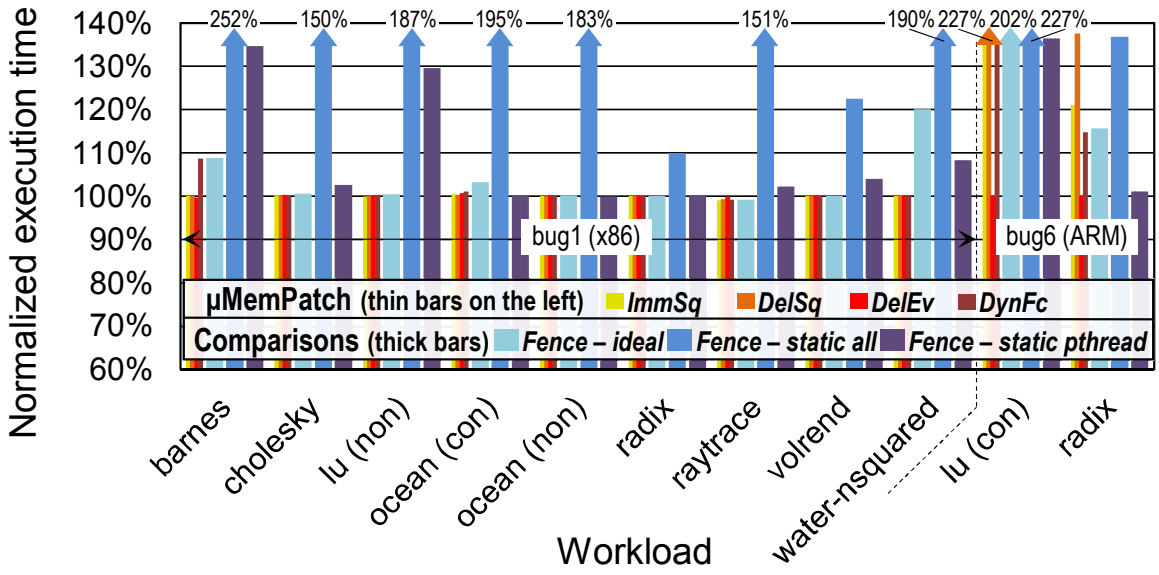


Figure 5.8: Execution time with patching for SPLASH-2, normalized to an unpatched system. μMemPatch bug-elusion methods entail minimal performance impact, $<$1% with the best elusion selection for each bug. By comparison, fence insertion solutions either entail high performance degradation (*all*), or they rarely succeed in bypassing the bug (*pthread*).

1% performance overhead over an unpatched system, in most cases, with a noticeable exception for *barnes* where *DynFc* shows a relatively large overhead (9%). For bug 6, *DelEv* appears to provide the minimal performance overhead, while *DelSq* underperforms even the conservative fence "static all" insertion.

5.6.4 Implementation Overheads

We measured the silicon area overhead of several bug-anticipation modules for the bugs listed in Table 5.3. For each bug, we designed and validated an anticipation module in SystemVerilog, almost equivalent to its gem5 counterpart. The major difference is that the SystemVerilog design uses a truncated 20-bit memory address and 8-bit sequence number. The module's inputs are the microarchitectural events of Figure 5.4, and its output is a bug-detection flag. To obtain the area footprint of each module for the FPGA target, we synthesized it using Synopsys Synplify Premier targeting Xilinx Spartan 6 XC6SLX150 at 45nm, and obtained the usage of the various cell types (e.g., flip-flops, LUTs). We then estimated the area of each cell by synthesizing a corresponding Verilog functional model with NanGate 45nm Open Cell Library, and computed the aggregated area of all the cells. For the standard-cell target, we simply synthesized the SystemVerilog module with Synopsys Design Compiler and the NanGate library.

We compared the silicon areas obtained in this process with that of an ARM Cortex-A9

Table 5.5: Area overhead of the bug-anticipation module

Bug	**Standard cell area** in um^2	**FPGA area** in um^2
1	1,562 (0.06%)	151k (6.0%)
2	1,605 (0.06%)	164k (6.6%)
3	1,406 (0.06%)	137k (5.5%)
4	1,655 (0.07%)	176k (7.0%)
5	1,435 (0.06%)	148k (5.9%)
6	3,345 (0.13%)	414k (16.6%)
7	1,173 (0.05%)	114k (4.6%)
8	618 (0.02%)	70k (2.8%)
9	19 (<0.01%)	787 (0.03%)
10	1,414 (0.06%)	209k (8.4%)
11	11 (<0.01%)	1,109 (0.04%)

Percentages are normalized based on a 2.5 mm^2 core area.

processor core (without L2 cache), which is 2.5 mm^2 [2]. Table 5.5 summarizes the area overheads for the escaped bugs we studied. On average, we find that our bug anticipation module entails approximately a 5.8% area overhead for the FPGA target. Note that, in practice, a designer is likely to develop a specialized programmable logic fabric that implements FSMs efficiently. Such a solution would likely be much smaller than our FPGA target, we provide the standard cell measurements as a lower bound for a specialized solution. The anticipation modules on the standard cell target occupy on average 0.05% of the area of the ARM core.

For bugs 1 through 5 we used a similar design, so their area overheads are also similar (5.5%–7.0%). For bug 6, we traced a forwarding from a prefetch to a load operation, which requires a large event observation window, leading to a higher area overhead (16.6%). Note that bugs 9 and 11 only require combinational observations (no sequences), so their detection modules are very simple. For bug 1, we also strove to optimize the design by hashing addresses (see Section 5.4.2), and obtained a smaller area of 117,767 um^2 for the FPGA-target and 1,119 um^2 for the standard-cell target. Finally, we also developed a bug-anticipation module that covers all 11 bugs together: this module occupies 606,611 um^2 in the FPGA fabric and 4,774 um^2 with the standard cells.

5.7 Discussions

Programmable logic size. When μMemPatch is deployed in a real-world processor, the capacity of programmable logic should be carefully determined by considering the silicon area and power consumption allowed for the programmable logic. To reduce the

programmable logic overhead, we can make an integrated bug-anticipation module combined for multiple bugs, which in turn offers opportunities to cover more bugs. To this end, one can leverage a string-matching engine (e.g., [190]) to efficiently encode a set of rules as state transition tables. In addition, as shown in Table 5.5, the FPGA fabric has significantly higher area overheads than the standard cells. Thus, one can design a hybrid bug-anticipation module that utilizes pre-built coarse-grained hardware components (e.g., finite state machines, address comparators) along with programmable logic.

Bug coverage. We conducted a coverage analysis on the bugs listed in Table 5.1. We analyzed the errata documents to infer what bugs can be discovered and recovered from by μMemPatch. We found that 59 out of 96 bugs (61%) would be covered by our solution. A majority of uncaught bugs are related to L2 and L3 caches, on-chip network, hardware prefetcher, and non-memory operations. If our bug-anticipation module is enhanced to monitor these modules, many of those bugs are likely to be caught. However, there are a few bugs that originated from complex combinations of advanced features (e.g., ARM Cortex-A15 #799271), which are unlikely to be caught even after the enhancement.

False bug detections (false positives) may lead to unnecessary performance degradation. There are a few sources of false bug detections: (1) conservative FSM modeling, (2) FSM simplification, and (3) memory address truncation and hashing. Note that (1) might be required if a bug-prone state cannot be accurately isolated due to unmonitored microarchitectural events (i.e., events not shown in Figure 5.4), while (2) and (3) are our approximation techniques in Section 5.4.2. We carried out a sensitivity study to measure the impact of (3) for bug 1 with the x86 gem5 configuration described in Section 5.6.1 and the twelve SPLASH-2 workloads in Table 5.4. We counted the bug occurrences for 64-bit whole addresses and those for 20-bit truncated addresses, and observed that the count is 1% higher with the 20-bit addresses on average (i.e., 1% false positive rate). Most workloads show much less than 1% false positives, except for *water-nsquared*, which showed approximately 10% false positives.

We could not quantify false positives originated from the FSM simplification. We applied the simplification on bug 10, which we could not simulate with gem5. In general, our simplification method takes a sound approximation on the FSM, thus, the simplified FSM captures a more conservative condition than the original condition. In addition, false positives due to context switching can be prevented by clearing the memoization table whenever the context is switched.

Bugs in μMemPatch's FSMs may lead to false positives and negatives. Small false positives are considered benign as they do not affect the correctness of program execution and minimally impact performance. Thorough root-cause analyses could completely eliminate false negatives (i.e., targeted bugs uncaught by the FSMs). In addition, FSMs should be designed to conservatively cover all likely conditions leading to targeted bugs. To reduce false positives and negatives, we can leverage formal property verification (FPV) for FSMs. These FSMs are relatively simple, thus, amenable to FPV tools.

Benefits over prior combinational-only solutions. μMemPatch strives to accurately detect bug-prone states by looking at sequences of microarchitectural events, which comes with larger area and power overheads compared to a combinational-only detection mechanism that monitors only control signals at the current cycle [177]. This combinational-only mechanism can be as effective as μMemPatch if control signals offer sufficient information on microarchitectural states. For instance, bug 11 is triggered when the CLFLUSH instruction is performed on the local xAPIC's address space. If the core has a dedicated signal to indicate whether or not the cacheline address accessed by the CLFLUSH instruction belongs to the xAPIC's address space, the combinational-only mechanism could catch this bug. However, it could not catch more involved bugs, such as bug 10, illustrated in Figure 5.5.

Using μMemPatch to mitigate side-channel security attacks. Recently, there are growing security concerns over the speculative execution in high-performance microprocessors. μMemPatch's ability to observe microarchitectural state can be re-targeted for suspicious activities from side-channel attacks, although it entails observing microarchitectural covert channels, which we have not discussed in this dissertation.

5.8 Related Work

[151] provides a comprehensive summary of various runtime verification solutions proposed in the literature. Among them, hardware-based patching solutions [150, 177] share the same goals as μMemPatch. As we discussed in Section 2.2, these works observe microarchitectural signals at runtime, creating a corresponding signature. If the signature matches any of the bug signatures, they initiate a recovery process (instruction editing, replay, checkpointing, hypervisor support). Note that the focus is on single-core bugs. [199] expands [150, 177] to multicore designs by catching incorrect cache coherence behaviors, as we also covered in Section 2.2. We share the same principles (detection and elusion) and

have some commonalities with these solutions. However, μMemPatch makes clear innovative claims over those: first, μMemPatch is tailored to capture sequences of microarchitectural events (vs. a single-cycle pattern detection with combinational logic only) leading to a much higher precision in bug identification. Second, our elusion mechanisms are specialized for memory subsystem bugs, so they entail extremely low performance overheads, compared to a generic checkpointing and rollback mechanism, which may cost 100,000 cycles [186].

Microcode patching is proven to be very effective at covering a wide range of bugs. However, its performance impact may be high. Moreover, microcode translation is not readily available for some processor products, such as ARM Cortex processors, which use hard-coded micro-operations. In a similar domain, dynamic binary translation can alter instruction sequences to bypass bugs. Unlike these hardware-based solutions, one can rely on software recompilation to avoid bug-prone instruction patterns. However, the analysis work for multi-threaded software can be challenging and the software source code is often unavailable to chipmakers.

Cache coherence and memory consistency are essential features in multicore systems [185]. Verifying and validating these features is a very demanding and time-pressured task (see Section 2.2 for prior verification efforts focused on these features). Thus, many errata bugs are (partially) related to these features (Table 5.1). Many prior proposals (e.g., [193, 153, 120]) have focused on the verification and validation of these features. Here, we consider the information obtained by these solutions as input to μMemPatch's bug-anticipation module.

5.9 Summary

This chapter discussed μMemPatch, a microarchitectural patching solution for memory subsystem bugs in multicore systems on chip. Unlike prior global patching solutions, μMemPatch is a distributed patching solution, where a small patch module is attached to each core and its L1 data cache. Each patch module observes microarchitectural events happening at the core and L1 data cache that are connected to it. It also observes cache eviction messages arriving at the cache, which offer essential information about microarchitectural state in the other chip components. Observing these messages allows μMemPatch to catch many system-wide bugs at the level of other global patching solutions.

Each distributed patch module consists of two components: a bug-anticipation module and a bug-elusion module. The bug-anticipation module is implemented in a programmable logic fabric located at each core, employing finite state machines to catch sequences of mi-

croarchitectural events that may lead to bugs at runtime. Its area overhead is approximately 6% compared to an ARM Cortex-A9 core, on average, over the eleven bugs that we evaluated in our experiments. However, our distributed approach comes at the cost of reduced bug coverage. Our current implementation may not be able to detect about 39% of the memory bugs that we collected from published errata documents.

In addition, we propose a few low-overhead bug-elusion methods, which can be easily paired with the bug-anticipation module. With these bug-elusion methods, μMemPatch incurs less than 1% performance degradation on SPLASH-2 workloads.

We now move on to address errors that may occur in the on-chip networks of complex SoC designs. Specifically, the next chapter discusses routing in these networks, and how to efficiently recover when a component failure hits this critical, single-point of failure subsystem of the SoC.

CHAPTER 6

Quick Routing Reconfiguration for On-Chip Networks

On-chip networks, or networks-on-chip (NoCs), are a promising communication infrastructure in system-on-chip (SoC) designs, because they provide high-bandwidth, concurrent and power-efficient communication among chip components. NoCs are often the only communication medium among chip components [97, 16], and hence a malfunctioning NoC possibly disables the entire chip in which it is deployed. For instance, if a packet is stuck at a router in the network, it prevents all other packets from moving through that router. Such a small failure may cause a domino effect in the network; the entire network may be stuck, resulting in a complete communication failure across all chip components. Thus, it is of paramount importance to protect NoCs from permanent faults.

Routing reconfiguration has been investigated as a cost-effective solution to tolerate permanent faults in NoCs. In particular, global routing-reconfiguration solutions (e.g., [22, 78]) can tolerate many faults impacting random components of the NoC. Such high fault tolerance is a key requirement in nano-scale semiconductor technology nodes where permanent faults could occur more frequently. These global solutions deploy topology-agnostic routing algorithms (e.g., [178, 145]) to guarantee correct delivery, maximal connectivity, and deadlock-freedom. Unfortunately, they tend to freeze the entire on-chip network during reconfiguration. For instance in [22], reconfiguration takes thousands of clock cycles for an 8×8 mesh even with dedicated reconfiguration hardware. When reconfiguration is conducted in software, it takes substantially longer [85, 60]. During reconfiguration, regular network activity is suspended; in-flight packets should be stalled at their current locations. Otherwise, new (post-fault) routes may conflict with old (pre-fault) routes, possibly triggering a deadlock situation. In contrast, many digital systems rely on uninterrupted operability from their communication infrastructure, therefore, these global solutions are not viable for them.

To address this limitation, in this chapter, we focus on providing routing reconfiguration capabilities without suspending global network activity. Specifically, we propose a local-

ized routing reconfiguration solution so that we can minimize the impact only to the small region affected by the fault. To this end, we *decompose an entire network into multiple segments*. We then capture the topology of the segmented network with routing metadata. Upon fault occurrences, we perform partial reconfiguration and update the routing metadata embedded in the network, then use it to compute emergency routes, instead of routing tables, via router-to-router communication occurring only in the network segment affected by the fault and a few adjacent segments. Note that this *partial reconfiguration corresponds to the reassembly process* in our decomposition-based approach; this reconfiguration is done via router-to-router communication for a small region of the network.

Based on this decomposition method, we present our novel routing framework, called BLINC, **B**risk and **L**imited-**I**mpact **N**oC routing re-**C**onfiguration. BLINC enables fast and low-impact reconfiguration, while providing valid routes under most fault occurrences, as in prior global routing reconfiguration solutions. However, occasionally, it may fail to cover rare fault patterns, where multiple faults occur within a segment, a limitation inherent to our decomposition method.

In addition, we demonstrate an important application of BLINC; it can be utilized to enable transparent reliability for NoCs, based on aggressive online testing and failure prevention. In our solution, individual components are taken offline and tested to evaluate if they are expected to fail soon, in which case they are disabled. BLINC allows us to quickly move components offline for testing, and then back online. Thus, it provides a first-response routing solution when a failure is deemed imminent. This capability allows our framework to operate uninterrupted and without data loss through testing and reconfiguration.

6.1 Background and Motivation

6.1.1 Avoiding Deadlock by Removing Cyclic Resource Dependencies

As briefly mentioned in Section 2.3, deadlock situations can happen when packets wait for each other in a cyclic manner. Such situations can be avoided by breaking cyclic resource dependencies [72], disabling at least one of the turns that contribute to the cycle [91]. For instance, Figure 6.1a forbids the turn 1-5-4 so that packets' routes do not include the sequence 1→5→4 or 4→5→1. This turn restriction breaks the cycle along nodes 0, 1, 5 and 4.

A class of routing algorithms [85, 178, 22, 176, 169], some of which are detailed in Section 2.3, does not rely on topology-specific characteristics (e.g., X or Y direction) so that they can compute routes in any random topology. For instance, the up*/down* routing

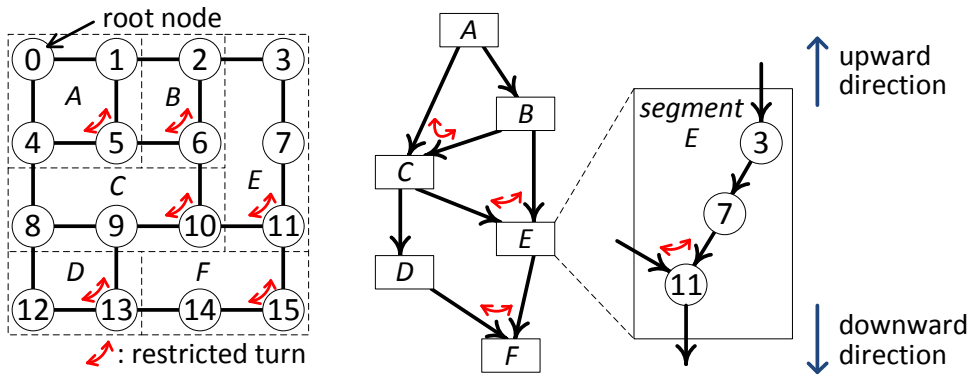


Figure 6.1: Network segmentation example. (a) The segmentation process begins by identifying a segment that starts and ends at the root node (node 0). Additional segments stem from nodes already part of other segments (e.g., nodes 2 and 6). (b) The corresponding segment-to-segment tree structure reflects this construction from root to leaves. (c) Nodes within each segment are organized in intra-segment trees. Note how each segment includes two connections to a parent segment.

algorithm [178, 176, 22] builds a spanning tree of links. Links towards the root node are marked as *up* links, while links towards the leaves are marked as *down* links [157]. The routing algorithm then disallows down→up turns; once a packet takes a down link, it is not allowed to turn onto an up link. While this restriction rule can be applied to any up*/down* tree in general, the routing algorithm usually spans the tree in a fixed manner (e.g., breadth-first [178, 22] or depth-first [176]). Consequently, these topology-agnostic approaches often provide limited options in the placement of turn restrictions.

6.1.2 Alternative Route Generation Cost Analysis

Most regular on-chip network topologies, including meshes and tori, often have many alternative routes available for each source-destination pair. For example, in a mesh, it is possible to choose between XY or YX routing, among other minimal options. Like in the regular topologies, irregular topologies which have broken links between nodes as in Figure 6.1a also have alternative routes in the network, though their numbers are limited compared to the regular topologies. In the figure, for instance, node 1 can reach node 4 through node 5 or through node 0. Here, we consider any fault can disable a link or a router, while a router disabling is modeled as disabling all links connected by the router.

Turn model. To avoid cyclic channel dependencies [73], and thus guarantee deadlock-freedom, some of these possible routes should be prohibited. For instance, the turn model

[91] disallows routes passing through a particular set of turns. A turn is defined as a connection between two links through a router. For instance, in Figure 6.1a, the turn 1-5-4 is prohibited so that no packet can traverse the path 1→5→4 or 4→5→1. Upon a fault affecting a link (or a portion of a router compromising link operability) the disabled turns must be recomputed to allow packets to go through alternative healthy routes. This effort entails a global routing reconfiguration [78], and it does not guarantee deadlock-freedom.

Up*/down* routing. Tree-based routing algorithms, such as up*/down* routing [178], can be applied to route packets in any topologies. The up*/down* algorithm assigns a total order to each node in the tree, from root to leaves, and it guarantees deadlock-freedom, as long as packets are not routed between two lower-order nodes (i.e., lower-depth nodes) via a high-depth node (no down→up turn). However, upon a topology change, the up*/down* algorithm suffers from long reconfiguration latencies both for on-chip regular topologies and off-chip network topologies. For instance, [22] reported a reconfiguration latency proportional to the square of the number of nodes in mesh topologies. Plus, [60] showed more than 20k cycles for 64-node off-chip networks. These long latencies originate from global reconfiguration [22] or from complex reconfiguration procedures [60].

Segment-based routing. Note that it is possible to design alternative routes based on local information if we use segment-based routing [145]. In segment-based routing, the entire network is partitioned into segments. The segmentation process starts by selecting a root node, and then identifying a segment as a sequence of nodes and links that starts and ends at the root node. Each subsequent segment is identified by building a sequence of nodes that starts and ends at nodes already included in the segmented portion of the network. Figure 6.1a shows an example of segmentation. Recent work [144] has shown that it is preferable to segment a network so that each segment has exactly two links connecting it to the already segmented portion of the network. We follow this advice in our experimental evaluation. We propose here to augment this algorithm and maintain metadata at each node so that, upon a fault, we quickly modify the routing configuration locally.

6.2 BLINC Overview

The main objective of our technique is to promptly find alternative routes for packets affected by a network topology change, due to a fault or other event which triggers reconfiguration. Specifically, our goal is to be fast and minimalistic in the number of routing modifications so that the network traffic is minimally perturbed.

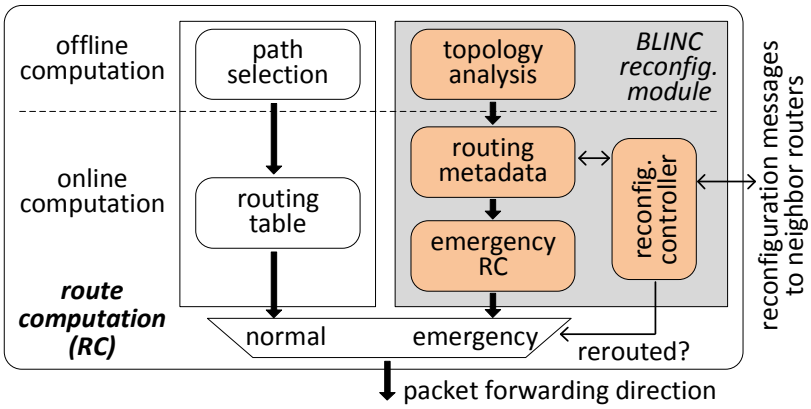


Figure 6.2: BLINC route computation. BLINC deploys two route computation components: routing tables (white) and reconfiguration modules (gray). On a fault occurrence, invalid routes are immediately replaced by emergency routes, while a software procedure in the background recomputes new optimal routes. When they are ready, they replace the emergency routes.

In distributed routing, forwarding directions are selected locally at each router. Thus, rerouting entails recomputing the routing tables for all routers in the network [166, 78, 22]. We observe that the recomputation effort can be reduced by utilizing *pre-computed routing metadata* so as to quickly pinpoint the affected routes. The immediate rerouting response from BLINC is fast and deadlock-free, but not necessarily minimal. However, traffic remains uninterrupted while a new minimal routing configuration is generated offline. Concurrently, the reconfiguration process also initiates a complete routing reconfiguration in software to generate new optimal routing paths in light of the new topology. Once this latter process is completed, the reconfiguration is transferred to the nodes and it replaces the emergency routing approach.

Figure 6.2 shows the components required for our BLINC algorithm. The blocks with white background color are part of the baseline router; routing tables are generated offline [117] and ready when the network becomes operative. Upon a topology change, the gray-background reconfiguration module quickly calculates valid alternative (*emergency*) routes so that packets affected by the change can be safely sent through them. Note that the majority of packets still utilize the original optimal routes. Thus, the deployment of the emergency routes enables fast rerouting at the expense of route optimality. In the following subsections, the reconfiguration module is further described.

6.3 Route Computation

6.3.1 Our Enhanced Segmentation Infrastructure

Our offline routing solution augments segment-based routing with an additional high-level tree structure, showing the connectivity between segments. The higher-level tree (Figure 6.1b) is built by traversing the network segment-by-segment, following adjacency, starting from the root segment. Any two adjacent segments have a parent-child relationship if the segment under consideration (child segment) has links connecting it to one or more segments already in the tree (parent segments). For example, once segment A and B have been considered, segment C is found to be a child of both. Nodes in the tree are ordered from root to leaves based on the order in which they are included in the tree (to improve readability, Figure 6.1b shows the segment order by using letters instead of numerals). Moreover, we also introduce an ordering of the nodes within each segment by building *"intra-segment trees"* (see Figure 6.1c). Once all nodes are ordered, deadlock-free routing can be enforced by forbidding turns around the highest order node in each segment. Note how the forbidden turns indicated in Figure 6.1a follow the approach described; for instance in segment E, the highest order node is 11, according to Figure 6.1c. Thus, the turn 10-11-7 is disabled.

To provide an intuitive understanding of our approach, we leverage the fact that each segment is connected to segments closer to the root through two links. Thus, upon a link failure within a segment, it is possible to use the two links to reach the two disconnected portions of the segment. The tree structure remains unchanged, and it is sufficient to find a different route through one or more segments. As an example, in Figure 6.1a, assume that packets going from node 4 to node 13 traverse nodes 8 and 9. If the link 8-9 fails, it is possible to find an alternate route via 5, 6, 10 and 9. Note that the alternate route may be non-minimal. However, as shown in Section 6.4, routes generated using our algorithm are only slightly longer than minimal.

6.3.2 Routing Metadata

To quickly find emergency routes upon a failure, the BLINC reconfiguration module leverages routing metadata that is computed while segmenting the network. Routing metadata is embedded at each router, and it includes three types of information:

- **Port type**: A router port can be connecting the router to a lower-order node (parent port), or to a higher-order node in the same segment (intra-segment port), or to a

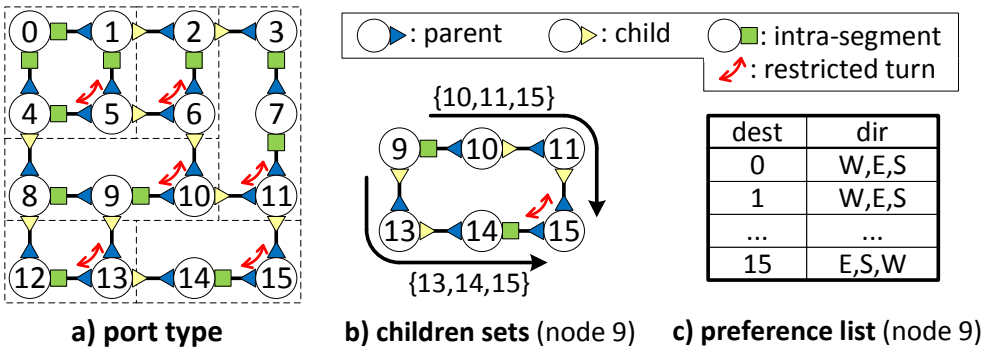


Figure 6.3: Routing metadata. (a) Port types are assigned based on our hierarchical segmentation process. (b) Each port in a node has an associated children set through child and intra-segment ports. (c) Preference direction lists associated with each node are optional.

higher-order node in a different segment (child port). Figure 6.3a indicates the type of all ports for our example network.

- **Children set**: the set of reachable nodes along downward routes for each port. Figure 6.3b shows the children sets for node 9.
- **Preference list** (optional): An ordered list indicating the preferred output directions. If available, this list is used to improve the quality of the emergency routes generated. The list can be user-provided or generated automatically. In our evaluation we prioritize based on distance to destination. Figure 6.3c shows an example of preference list.

Note that if an output port includes the destination node in its children set, the node has at least one valid route to destination, since downward traversal is always allowed. To store the routing metadata at each router we need: 2 bits to encode the port type, a bit array to indicate the children set for each port, and 6 bits per destination to encode the preference list (at most 3 directions, 2 bits to encode each direction). Thus, for an 8×8 mesh, we need at least 264 bits per router (384 additional bits if the preference list is provided).

6.3.3 Reconfiguration Process

Upon a link failure, BLINC leverages the metadata described above to quickly generate alternative routes for the affected packets. Figure 6.4 illustrates the process with a high-level schematic. Each segment can be represented as a chain of nodes, and thus the segment affected by the failure will find itself disconnected. The localized reconfiguration process will re-establish connectivity for all nodes by exploiting the additional routing paths that

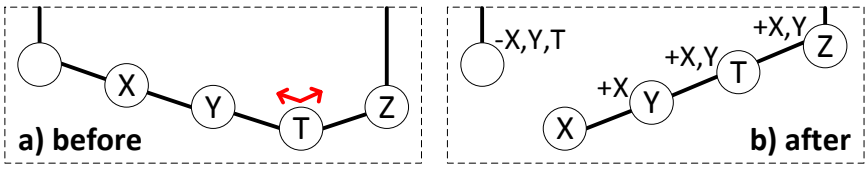


Figure 6.4: Children set update on reconfiguration.

had earlier been disabled to avoid deadlock. Indeed, because of the construction described in Section 6.3.1, each segment contains exactly one disabled turn which, at this point, will be re-enabled. Then all the children sets within the segment must be updated so that each node is reachable from the segment boundary. This goal entails adding children to some ports' children sets and removing children from other ports. In the example of Figure 6.4, Y was originally reachable via X only, but after the failure, it becomes reachable via Z instead. Finally, the additions and subtractions to the children sets are propagated outside the segment, until a common ancestor is reached. The reason to keep the children sets updated is that, during a topology change, packets whose routing is affected by the change will use the children sets to determine their new paths. This decision guarantees that even packets undergoing a detour will reach their destination in a minimum number of hops possible in the new topology. Moreover, the elements added or removed from the children sets are maintained in a separate bit array so that it is possible for a packet to determine when it is undertaking an "emergency route". Note that this emergency route is also deadlock-free because the rest of the network maintains the same turn restrictions as before. Note also that a new optimal routing configuration is being generated in the background in software; once computed, it overwrites all emergency routes.

Algorithm 6.1 provides an outline of the reconfiguration algorithm. We describe each step in detail below and illustrate the process with an example in Figure 6.5. In the most general case, the faulty link is not adjacent to the node with the disabled turn for the segment that suffered the fault. In this case, all steps must be executed. In the case where the faulty link is adjacent to the node with the disabled turn, only steps 1 and 5 are required.

Step 1: Disabling the link. Nodes adjacent to the faulty link stop sending packets through it (line 1 in the pseudo-code).

Step 2: Re-enabling the turn. Before the fault, the node with the disabled turn (T in Figure 6.4) was the leaf in the intra-segment tree. After the fault, the portion of the segment between the turn-disabled node and the faulty link becomes isolated (portion between X and T in Figure 6.4). To reconnect it, we need to i) re-enable the turn at T, ii) swap the port

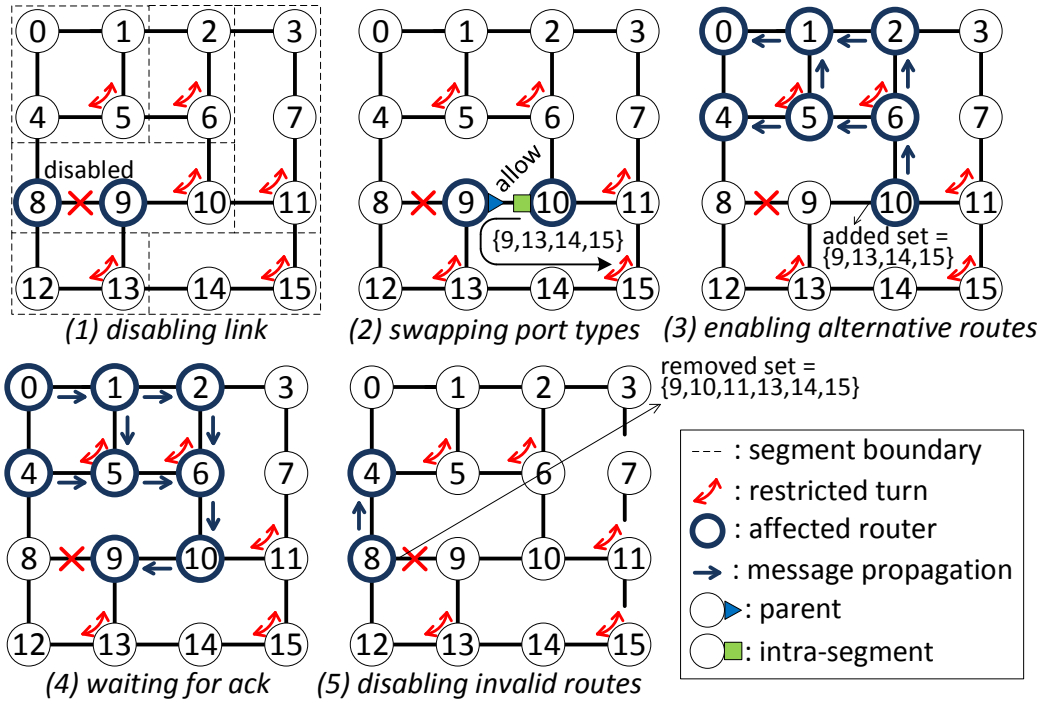


Figure 6.5: Reconfiguration example. The turn restriction in the segment with a fault is eliminated and the new intra-segment leaf is the node next to the faulty link (node 9). An added-children set is created to indicate the new routes to reach the downward nodes after the fault, and it is propagated towards the root node. Once this step completes, acknowledgment messages are propagated back down. A similar process is followed to propagate removed-children sets starting from the other end of the fault (node 8).

types for each link in the isolated portion (Figure 6.5–(2)), and then iii) create an *"added-children set"* for each port in the isolated portion, which includes all the nodes downstream towards X (lines 3-5 in the pseudo-code). The added-children set is used to detect when a packet has to take a detour because of the fault: if the destination is in the original children set, the packet does not experience a detour, but if the destination is in the added-children set the detour is necessary.

Step 3: Enabling alternative routes. Once the added-children set for the turn-restricted node (T in Figure 6.4) is computed, the set is propagated toward the root node, across segment boundaries, to instruct every node of the new route to reach the destinations next to the faulty link. For each node towards the root, the current children set of this node is compared against the incoming added-children set, then the latter is reduced to include only nodes not already present in the children sets of the node under consideration. Indeed, if a destination was already in the children set of a node, no routing change should be applied at this node. The process stops when the added-children set becomes empty. In

Figure 6.5–(3), the added-children set is propagated from node 10 all the way to the root node and node 4 (lines 6-9).

Step 4: Waiting for acknowledgment. The last recipients of the added-children set generate an acknowledgment message that is propagated all the way back to the node adjacent to the link failure (line 10).

Step 5: Disabling invalid routes. Finally, the other portion of the segment (the one connected to the parent port side of the faulty link) generates a *"removed-children set"* to indicate that it cannot reach the other end of the link. The removed-children set is propagated towards the root in a similar fashion to the added-children set; the only difference is that nodes are eliminated from the removed-children set when they already exist in the union of the children sets of all ports other than the port receiving the remove-children message (lines 11-15). Each router tracks its children's modification through a bit vector.

Algorithm 6.1 BLINC reconfiguration procedure

```
 1: (parent_node, child_node) = disabled_link
 2: if(Not HasTurnRestriction(child_node))
 3:   turn_node = FindTurnRestrictedNode(child_node)
 4:   ReversePortType(from child_node to turn_node)
 5:   added_set = GetNewChildren(turn_node)
 6:   curr_node = Parent(turn_node)
 7:   while(Not IsEmpty(added_set))
 8:     AddChildrenSet(curr_node, added_set)
 9:     Update(added_set), curr_node = Parent(curr_node)
10:   WaitForAck(child_node)
11: removed_set = GetUnreachableChildren(parent_node)
12: curr_node = Parent(parent_node)
13: while(Not IsEmpty(removed_set))
14:   RemoveChildrenSet(curr_node, removed_set)
15:   Update(removed_set), curr_node = Parent(curr_node)
```

Messages carrying the port swapping and the added- or removed-children sets can be transmitted on the same network using a dedicated virtual channel. Packets in reconfiguring routers are stalled until the reconfiguration process is completed. Then packets are routed as before, simply following the turn restriction rules. However, when a packet is forwarded through a port that includes that packet's destination in its added- or removed-children set, the packet has entered "emergency routing", and a corresponding flag is set in its header flit. Once a packet is in emergency routing, it routes using a minimal-hop tree-routing

Table 6.1: Network parameters

Parameter	Value
Router	Wormhole, 3-stage pipelined
Flit buffer	8 flits/input port, 64 bits/flit
Network topology	Mesh 6×6, 8×8, 10×10
Packet size	10 flits/packet
Traffic	Uniform random, 0.05 flits/cycle/router
Number of faults	0%, 1%, 5%, 10%
Processing delay	5 cycles for *add* and *remove*, 1 cycle for *ack*

algorithm by always selecting the port containing the destination in its children set. When multiple options are available, the preference list is used to choose the best one. When no choices are available, the packet is routed towards the upward direction, in the worst case, to the root node.

Note that our online reconfiguration process may fail to support rare fault occurrences where multiple faults impact the same segment. Such fault occurrences break the tree structure of segments (Figure 6.1b). Thus, the network may not be able to deliver packets due to the lack of valid routes for some source and destination pairs, although the network still maintains a physical connection between the pairs. In case of these cases, our process triggers a re-segmentation process to change the segment structure.

6.4 Experimental Evaluation

In this section, we first evaluated the performance characteristics of our solution in Section 6.4.1, and then considered its value in the context of a fault detection and reconfiguration application for uninterrupted availability in Section 6.5. We evaluated our BLINC algorithm with a cycle-accurate NoC simulator, Booksim [73]. Table 6.1 shows detailed parameters used in this experiment.

Fault model. We assume that the faults' spatial distribution is uniform over the component's area [22]. Thus, we derive area values for each major module (input buffers, crossbar, route computation unit, etc.) of the router in [73], using Synopsys DC and the Nangate 45nm target library. The faults' impact noted in this synthesized logic model is then mapped to the link-failure model of the network: i) faults in the router's control logic (9.4%) disable the entire router, affecting all surrounding links; ii) faults in the router datapath (90.6%) disable only one link, including the two I/O ports connected at its ends. Our evaluation considers 4 baseline systems, with the following fault's nominal rates: 0%, 1%,

Table 6.2: Average number of affected routers and reconfiguration latency

Method	Initial faults	Impact of next fault: # affected routers			Impact of next fault: Reconf. latency (cycles)		
		6×6	8×8	10×10	6×6	8×8	10×10
BLINC	0%	7.0	9.0	9.9	21.0	26.0	30.1
	1%	6.8	9.3	10.3	21.0	28.1	31.1
	5%	7.1	9.1	10.0	23.9	28.7	30.6
	10%	6.6	8.9	10.0	24.9	30.0	34.4
ARIADNE [22]	-	All routers			1.3k	4.1k	10k
OSR-Lite [187]	-	All routers			-	∼569*	-
Wachter et al. [197]	-	All routers			-	-	0.2k–208k

5% and 10%. The rate corresponds to a number of injected faults equal to the fraction of links in the topology (an 8×8 mesh has 112 links, thus the 25% rate would be 28 faults). Note that some faults may affect several links.

To evaluate BLINC, we create 10 distinct faulty topologies, using different random seeds for each fault rate, and then generate one more failure at 10 random different sites. In total, each fault rate is evaluated with 100 distinct fault situations (10 baseline topologies × 10 failure locations).

6.4.1 Characterization

Number of affected routers. The left part of Table 6.2 reports the average number of routers affected by a reconfiguration event over a range of fault densities and network sizes. The affected number of routers increases slowly with network size, showing that BLINC localizes the fault manifestation to a small region. Compared to existing methods [22, 187], BLINC achieves more than 80% reduction in the number of affected routers, across a wide range of fault densities.

Reconfiguration latency. Reconfiguration latency was computed assuming that each node takes 5 cycles to process an add/remove children-set message and 1 cycle to propagate the acknowledgement messages. Our findings are reported on the right part of Table 6.2. While reconfiguration latency is minimally sensitive to network size, it does show a steady increase with growing fault density. We believe this is due to the naturally occurring longer segments in faulty networks, which in turn, impose more hops in the transmission of reconfiguration messages. Overall, BLINC's reconfiguration latency is 98% shorter than previous hardware-based techniques [22]. Note that [187, 197] have the disadvantage of performing reconfiguration in software, thus competing with applications for CPU-time.

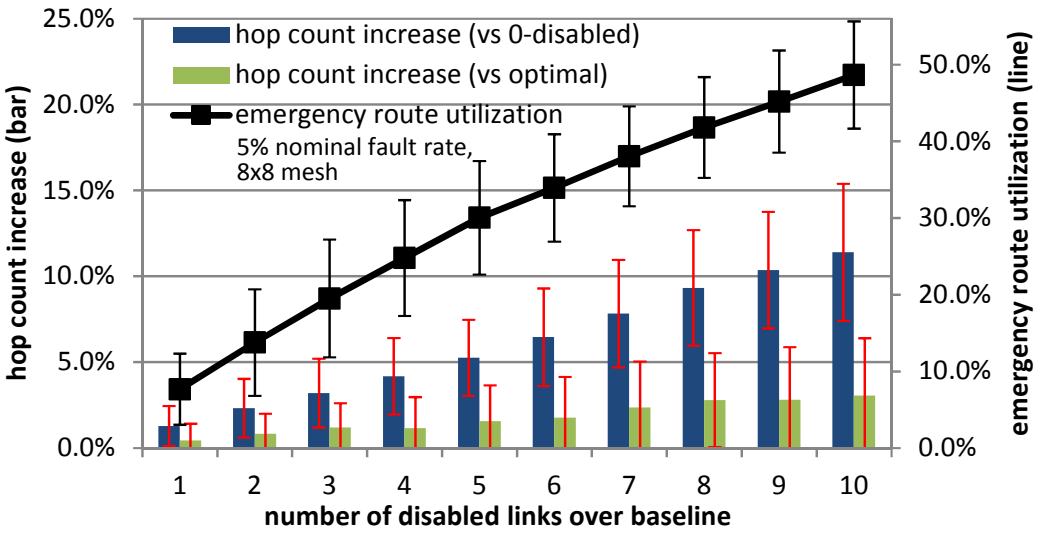


Figure 6.6: Effect of multiple disabled links. The network performance degrades gracefully as additional links are disabled. The emergency routes shows only slightly worse hop counts than optimal ones.

Quality of emergency routes. Figure 6.6 shows the hop count increase and the utilization of emergency routes when the number of disabled links over the baseline topology varies from 1 to 10. The average hop count increases as the number of disabled links increases, up to 11.4%, compared to the topology without the disabled links. In addition, the plot indicates only a 3.0% increase in hop count compared to optimal routes with the disabled links. Further, emergency routing is applied to 7.7% of the packets at 1 fault, and up to 48.7% of the packets at 10 faults. These results suggest that emergency routing provides near-optimal routes.

6.4.2 Discussion

BLINC reconfiguration is capable of tolerating a single link failure per segment. Once a segment is damaged by a fault, the next fault in the same segment may not be recovered by BLINC. We deploy a background rerouting algorithm to handle this. The background rerouting computes an optimal routing function for the surviving network topology. Subsequently, the new metadata can be overwritten safely when the network is idle. Although this rerouting process usually takes significantly longer than our fast reconfiguration [85], it is still much shorter than the plausible time to the next failure. Indeed, BLINC supplements existing solutions by providing a fast emergency response against the fault.

BLINC's metadata adds to the router's area footprint, so it can be affected by faults. However, since the metadata consists mostly of storage, it can be easily protected by error-

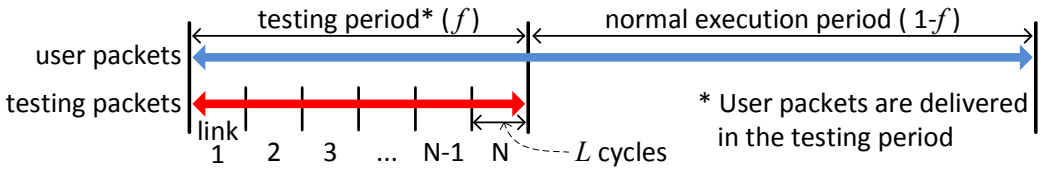


Figure 6.7: Online testing flow. The network should remain completely available even during testing so that an aggressive testing frequency does not degrade network performance.

correcting codes.

6.5 Application: Online Fault Detection

To showcase the value of BLINC's approach, we present its deployment in a fault detection and reconfiguration solution that provides uninterrupted availability. The methodology tests network resources aggressively to detect early signs of an upcoming fault. Each link, in turn, is taken offline for testing, which is performed through transmission of testing packets generated by a test pattern generator, such as the one in [93]. This approach can detect a majority of router faults [78]. For this methodology to be valuable, i) the network should be available and connected while a link is being tested and ii) the testing approach should be capable of detecting early signs of link failure (e.g., increased delay) so that the network can reconfigure around the upcoming fault with no loss of packets. The first requirement can be accommodated by our BLINC algorithm; as shown in the previous section, it can provide emergency routes with minimal overhead. The latter requirement has been solved in the context of microprocessor designs [184, 206] but, to date, no solution of this kind has been developed for NoCs. Because of BLINC's fast and localized reconfiguration, it is possible to select each link in turn, take it offline, test it in a harsh operating environment to mimic the circuit aging, (such as lowered supply voltage [184]), and then bring it back online. BLINC can simply reconfigure the NoC to avoid the target link before the testing phase, and then reactivate the original routing after the test completes. If a link is found at risk of experiencing failure, emergency routing is maintained until a new segment-based routing solution is computed in the background.

The testing flow is characterized by two parameters: the length of the test duration for each link (L) and the network testing rate (f), as illustrated in Figure 6.7. One complete testing cycle entails testing each link in turn. Note that the network should remain completely available even throughout testing.

Figure 6.8 reports our findings, showing average packet latency under a range of testing rates and test durations. Viable testing rates and durations were derived from [125, 93,

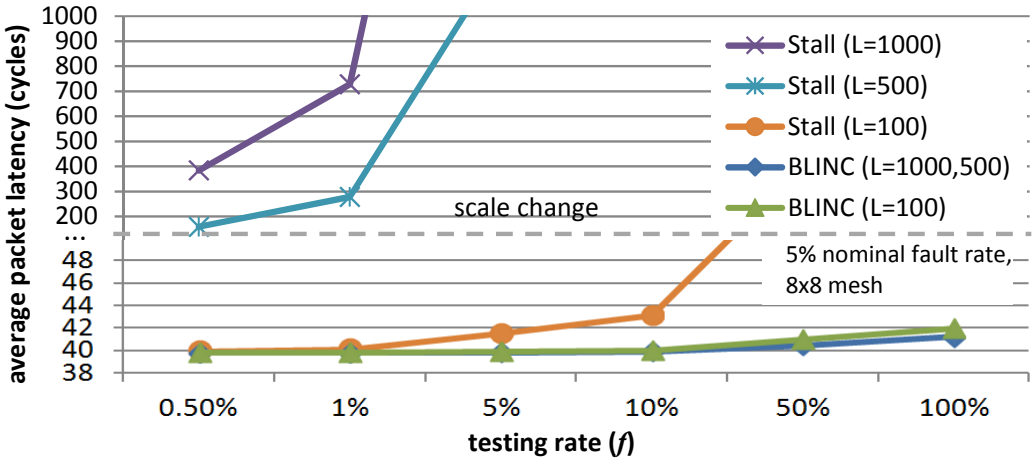


Figure 6.8: Average packet latency under online testing. BLINC reconfigures routing quickly, supporting online testing with only negligible performance degradation.

206]. For instance, the rightmost data point in our plot corresponds to one complete test period every 112,000 cycles. We compare our measurements against a routing solution (called *Stall*) that does not benefit from our BLINC solution. In Stall, packets are simply stalled in their buffers whenever they are trying to use a link undergoing testing. We only compared against Stall, because other solutions [166, 78, 22, 197] have reconfiguration latencies longer than the test durations we are evaluating. From Figure 6.8, we observe that with BLINC, average packet latency is minimally affected (6% in the worst case) by the ongoing testing and reconfiguration process, regardless of test durations. Moreover, Stall cannot provide uninterrupted availability beyond a test duration of 500 cycles, even at very low testing rates (the latency is increased by 17 times for L = 1,000 and f = 1%).

Figure 6.9 shows the accepted flit rate during testing. We evaluated this under a randomly chosen 8×8 mesh topology and 1,000 random traffic patterns with $L = 500$. BLINC provides a steady packet delivery capability. The accepted flit rate shows only little fluctuations as shown in the figure. On the contrary, Stall shows a decreased delivery capability with a periodic increase when the link under test changes.

6.6 Related Work

Fast routing reconfiguration has been investigated mostly for off-chip interconnection networks, such as local area networks (LANs). [60] proposes a dynamic, progressive reconfiguration procedure based on the up*/down* routing algorithm [178], which uses graph manipulation operations. However, it only discusses reconfiguration principles without investigating the details and hardware required to support the reconfiguration procedure.

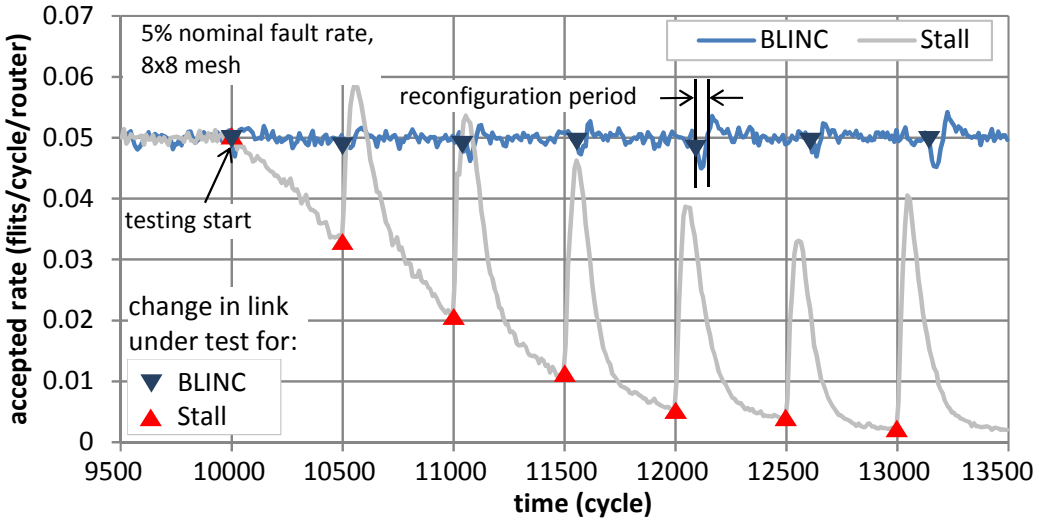


Figure 6.9: Accepted flit rate during testing. BLINC shows a steady rate with little fluctuations during reconfiguration (near 12,100 cycle and 13,150 cycle). Stall shows a constantly decreasing rate with periodic peaks.

Table 6.3: Comparison with existing routing reconfiguration techniques

Method	Context	Computation	Impact	Speed	Tolerance
BLINC	on-chip	hardware	**local**	**very fast**	**high**
ARIADNE [22]	on-chip	hardware	global	fast	**high**
MD [80]	on-chip	hardware	**local**	**very fast**	low
Sem-Jacobsen et al. [179]	off-chip	software	**local**	slow	**high**
OSR-Lite [187]	on-chip	software	global	moderate	moderate

Similarly, [179] proposes a fast dynamic reconfiguration procedure based on partial channel lists. Although both solutions claim fast reconfiguration and uninterrupted operation, they do not provide evidence supporting their claims.

In the on-chip networks domain, OSR-Lite [187] proposes a fast reconfiguration solution utilizing resources to support two routing-computation logic sets based on [171], with only one of them active at a time. Upon a fault occurrence, a central routing manager calculates the replacement routes, while the old ones are still in use, and then they are swapped when the calculation ends. While OSR-Lite is reported to be faster than hardware solutions, the dedicated central manager can be a single-point of failure. In addition, it suffers from low fault tolerance due to the baseline routing algorithm [171] as discussed earlier.

Similarly to these fast reconfiguration solutions, BLINC also strives to reduce reconfiguration latency, but it also provides the high fault tolerance that is typical of the global, topology-agnostic routing solutions [171, 22, 157], discussed in Section 2.3. To summarize how BLINC fits in the related research landscape, Table 6.3 presents a comparison of

relevant routing reconfiguration techniques.

6.7 Summary

In this chapter, we presented BLINC, a brisk and local-impact NoC routing reconfiguration algorithm. BLINC utilizes a network segmentation process, which partitions an entire network into multiple segments. This decomposition allows us to compute just a small amount of information, called routing metadata, in computing alternative routes around a faulty location. The metadata is distributed across the network, embedded in each router. Upon a fault occurrence, it is exchanged across the routers located within a small fault-impacted region of the network. The computation leads to a partial reconfiguration, which is the key trait of BLINC's quick routing reconfiguration.

Our evaluation shows an 80% reduction in the number of routers affected by reconfiguration, and a 98% reduction in reconfiguration latency, compared to state-of-the-art global routing reconfiguration solutions. While achieving these significant reductions, BLINC's partial reconfiguration may not handle multiple faults occurring at the same segment. We also discussed how BLINC enables uninterrupted availability for networks-on-chip, by allowing individual network links to be taken offline for testing at high frequency. BLINC maintains stable network performance with only a 6% increase in latency during testing, in contrast with a 17-fold latency increase for a baseline approach that stalls one link segment at a time.

In the next chapter, we continue the discussion of routing in on-chip networks by presenting a solution that can efficiently recompute routing paths in the face of failures, while also taking into account network congestion.

CHAPTER 7

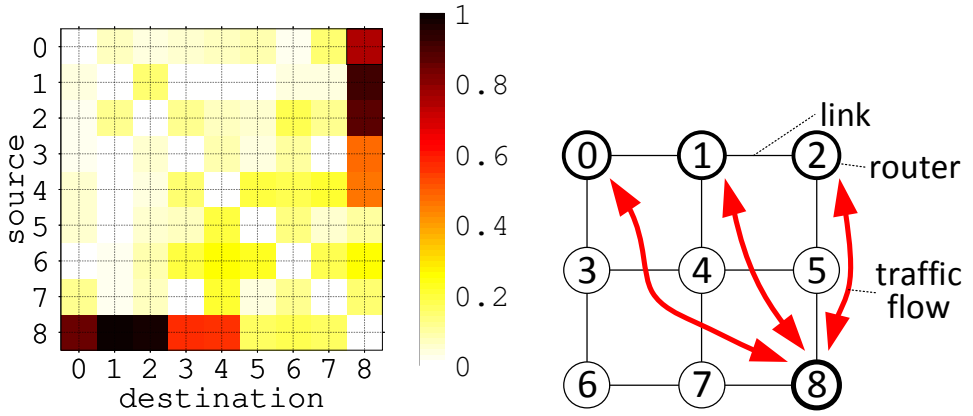
Application-Aware Routing for Faulty On-Chip Networks

In continuing to address the problem of interconnect routing in the presence of component failures, this chapter presents a novel routing algorithm that, unlike the solution of the prior chapter, seek a new set of optimal routes across the entire system after each new failure. The algorithm improves over state-of-the-art solutions in that it minimizes performance degradation even upon several fault occurrences. In prior works, finding application-optimized routes involved heavy computation to evaluate various route options available in the network. To reduce the amount of computation, the solution discussed in this chapter proposes to utilize localized routing rules that can quickly prune suboptimal route options.

As explained in the previous chapter, prior fault-tolerant routing solutions [22, 78, 157, 110, 169, 208, 80] re-compute routes upon each fault occurrence to utilize only fault-free paths. Nonetheless, they often lead to severe performance degradation after only a few faults, making the continued deployment of the chip impractical. For example, in [157], the throughput of an 8×8 mesh network drops by over 20% after only 10 faults. This steep loss is mainly due to increased traffic congestion on the remaining healthy paths.

Application-aware routing solutions [155, 69, 59] minimize performance degradation by leveraging the application's traffic patterns to compute application-optimized routes. One of the major drawbacks of these solutions is that they entail almost always very complex and resource-demanding computations. This is because there are a large number of deadlock-free route options possible in fault-ridden, irregular topologies, and each of these options needs to be evaluated to obtain optimal routes.

To reduce the computation requirements, we can leverage the fact that most packets flow among only a subset of the network-on-chip (NoC) nodes [204, 55]. In other words, we can prioritize a handful of high-traffic communication paths when exploring route options. More importantly, we noticed that we can exploit localized routing rules, called turn-enabling rules, that *decompose a network-wide route decision into a localized one*. The turn-enabling rules capture the interactions among turns – when a turn is restricted



(a) The amount of traffic with *lu_con*

(b) Traffic flow on a 3×3 mesh network

Figure 7.1: Application's traffic pattern. (a) The normalized traffic pattern for a portion of the SPLASH-2 application *lu_con* [204] shows that the largest fraction of traffic is between nodes 0-2 and 8. (b) When *lu_con* runs on a 3×3 mesh network, there is a significant amount of traffic flow between the upper-left region and the lower-right region.

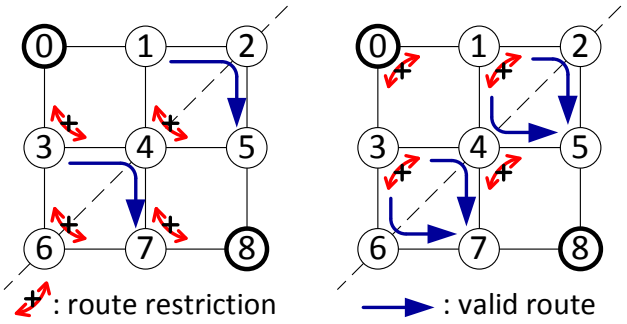
for the sake of deadlock-freedom, some other nearby turns must be enabled to guarantee maximal network connectivity. Here, we propose a few different such rules applied to 2D mesh networks.

With turn-enabling rules, we propose to decide turn restrictions iteratively; in each iteration, we determine the location of one turn restriction then enable adjacent turns by applying the rules. This iterative procedure completes when we finally find deadlock-free, maximally-connected, application-optimized routes. Note that *this iterative procedure corresponds to the reassembly process* in our decomposition approach. We call the routing solution proposed in this chapter FATE, from **F**ault- and **A**pplication-aware **T**urn model **E**xtension.

FATE attains efficiency from leveraging its iterative procedure that aggressively prunes the search space of route options in faulty network topologies. FATE is designed to find routes that maximize the effective path diversity that an application experiences at runtime, thus, minimizing network congestions. While the iterative procedure significantly reduces route-search efforts, it may not be able to find an optimum (i.e., the best set of routes), which can be overlooked due to the pruning process.

7.1 Motivation

Applications running on network-on-chip interconnects exhibit unique communication demands among processing units. To illustrate this point, we profiled the amount of traffic



(a) Routing function 1 (b) Routing function 2

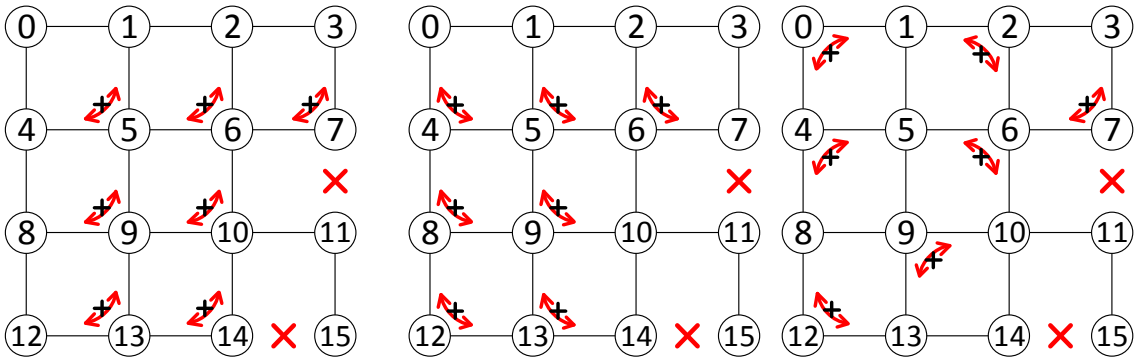
Figure 7.2: Traffic-aware routing restrictions. (a) When selecting the routing function 1, which forbids the north-east turns at node 3, 4, 6 and 7, there are only two allowed turns to cross the mid-diagonal line. (b) In contrast, the placement of routing restrictions as in the routing function 2 allows four distinct turns across the mid-diagonal.

from a portion of the *lu_con* benchmark from the SPLASH-2 suite [204], as summarized in the normalized heat map in Figure 7.1a. In the heat map, each row corresponds to a source node, and each column corresponds to a destination node. The heat map clearly shows that there is a great amount of traffic to and from node 8. In particular, the node 8 communicates more frequently with nodes 0, 1, and 2. These three traffic routes (nodes $0 \leftrightarrow 8$, $1 \leftrightarrow 8$, and $2 \leftrightarrow 8$) are depicted in Figure 7.1b, showing that there is much more traffic flow for the paths between the upper-left region and the lower-right region than the other paths (e.g., between the upper-right region and the lower-left region).

As introduced in Section 6.1, routing restrictions are necessary in order to prevent deadlock situations among packets. Using the route restrictions shown in Figure 7.2a, there are only two turns that allow transferring traffic from the upper-left region of the network to its lower-right region. In contrast, the route restrictions shown in Figure 7.2b allow four distinct turns. As a result, the routing solution of Figure 7.2b is less likely to cause congestion. Application-aware solutions of this type have been investigated in [155, 69, 116, 182, 59], but they often entail high routing computation overheads.

To generalize the impacts of turn restrictions with respect to traffic flow, Figure 7.3 shows three different turn-restriction patterns that are deadlock-free on a faulty 4×4 mesh network. Note that there are many other deadlock-free turn-restriction placements, which are not shown in the figure. Any turn restriction prohibits the route that takes the turn, reducing path diversity in the network. Some of the paths are likely to be utilized more frequently than the other paths, depending on the application's network traffic flow. Thus, it is important that we choose turn restrictions that have minimal traffic blockages.

To illustrate this point further, the turn restrictions in Figure 7.3a prevent all west-to-



(a) North-west turn restrictions (b) North-east turn restrictions (c) Irregular turn restrictions

Figure 7.3: Various deadlock-free turn-restriction patterns. In the mesh network topology, there are many possible patterns of turn restrictions. This figure shows three turn-restriction patterns in a 4×4 mesh network with two faulty links. The faulty links are marked as red X marks.

north or north-to-west turns, while all east-to-south or south-to-east turns are allowed. Specifically, there is only one route between node 3 and node 12 (3↔2↔1↔0↔4↔8↔12), while there are 10 distinct, minimal (i.e., 5-hop) routes between node 0 and node 14 (e.g., 0↔1↔5↔9↔13↔14). Figure 7.3b has a very different path-diversity characteristic; only one path between node 0 and node 14, but more than 10 paths between node 3 and node 12.

Figure 7.3c shows an irregular turn-restriction pattern. Without detailed analysis, it is clear that this pattern has multiple routes between node 3 and node 12, as well as multiple routes between node 0 and node 14. In other words, this pattern has more balanced path diversity across different source-destination pairs.

Application-specific routing solutions aim at offering full flexibility in the turn-restriction placement [155, 69, 59]. They analyze an application's communication paths, and selectively remove routes in the network until no cyclic resource dependency exists. As discussed in [69], however, this type of approaches (e.g., [155]) scales poorly as the network size increases. [69] tries to overcome this scalability issue in two steps. It first minimizes the number of cyclic dependencies, and then uses VCs to break the remaining cycles. However, using VCs to avoid deadlock is a costly option in resource-constrained NoCs. In contrast, our approach does not require VCs for deadlock-freedom, and VC resources can be entirely dedicated to prioritize traffic or reduce congestion.

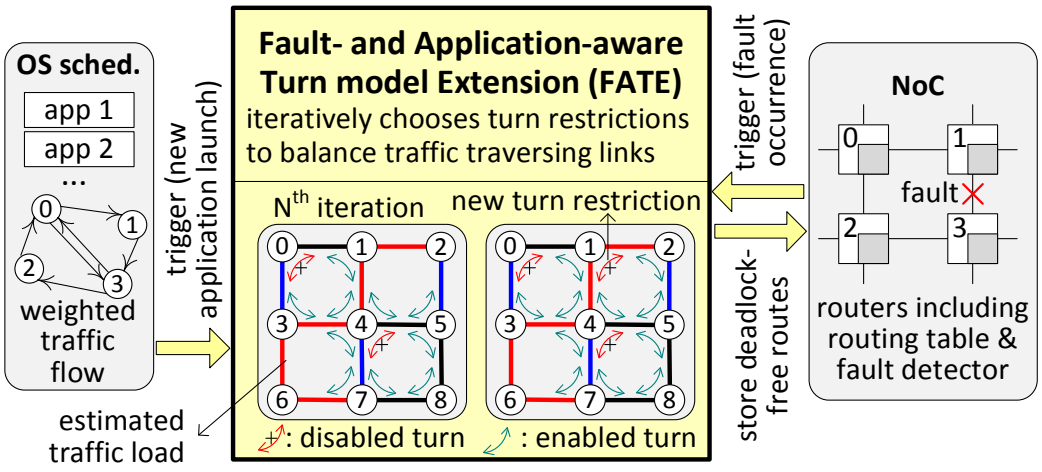


Figure 7.4: FATE overview. FATE is triggered by either a new application launch or a new fault occurrence. It leverages a traffic load estimator to compute a deadlock-free routing function that minimizes congestion. The new routing function is then stored at the routers.

7.2 FATE Overview

FATE is a software-based offline routing algorithm that generates a deadlock-free routing function, while maximizing the number of distinct paths available between nodes with high communication requirements, for applications running on faulty 2D mesh networks. Figure 7.4 shows how FATE operates: it is triggered by a new fault occurrence or an application launch. It then uses the SoC's idle cores to compute a new routing function. If no idle core is available, the start of the application may be delayed to compute the routing function, or a prior, possibly non-optimal, function may be used, until a core becomes available. While finding optimal, deadlock-free routes is an intractable problem [116, 182, 69], FATE's efficient heuristic quickly discovers a near-optimal routing function: it first computes the minimal number of turn restrictions that must be placed in the network. This value can be easily derived from the total number of cycles in the topology. The construction is then based on an iterative exploration, where turn restrictions are placed one at a time. After placing each turn restriction, FATE deploys its *turn-enabling rules* (Section 7.3) to enable turns that must be active in order to maintain both maximal connectivity (i.e., no disconnection) and deadlock-freedom in light of the most recent turn-restriction choice. Moreover, in choosing the location of each new turn restriction, FATE leverages a novel traffic load estimation using the communication patterns extracted via application analysis (Section 7.4). If a deadlocked or disconnected routing configuration is encountered during the exploration, FATE uses backtracking to broaden the search until it finds a satisfiable solution. Finally, the network's routers are re-programmed using the new routing function.

FATE assumes that information about the application's communication patterns is available: indeed, these can be observed in SoC applications using runtime profiling, and then modeled through Markov chains [54, 51]. As in many fault-tolerant routing solutions, we also assume that the NoC is equipped with a fault diagnosis solution (e.g., the online network testing presented in Chapter 6). In addition, we assume that our solution is deployed in systems where the OS is notified of new fault detections and can launch FATE's routing computation software, updating routing tables accordingly. This assumption may not hold in some systems. For those systems, FATE can be adopted by deploying a dedicated network manager. In this setup, FATE calculates in advance optimized routing functions for representative communication patterns, and stores them in memory. At runtime, the network manager evaluates current traffic patterns, and chooses the best routing function by comparing against the patterns of the stored ones.

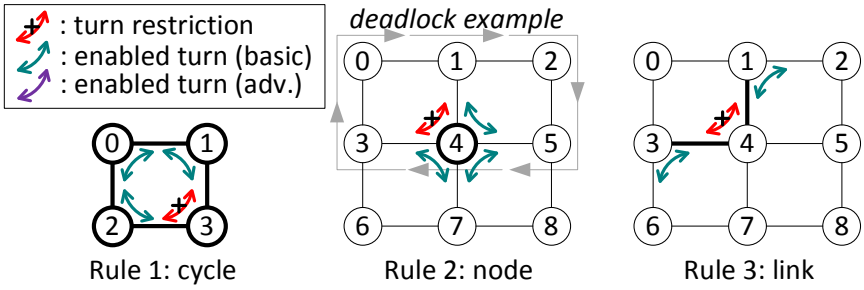
7.3 Turn-Enabling Rules

This section discusses the rules that determine which turns must be enabled as a consequence of another turn being disabled. These rules, grouped into *basic* and *advanced*, can be applied in any order, and are illustrated in Figure 7.5. We first describe how the rules operate in regular meshes, and then extend them to faulty topologies. Note that our approach minimizes the number of turn restrictions, and thus maximize cumulative bandwidth to all destinations, but it does not necessarily enable minimal-length routes.

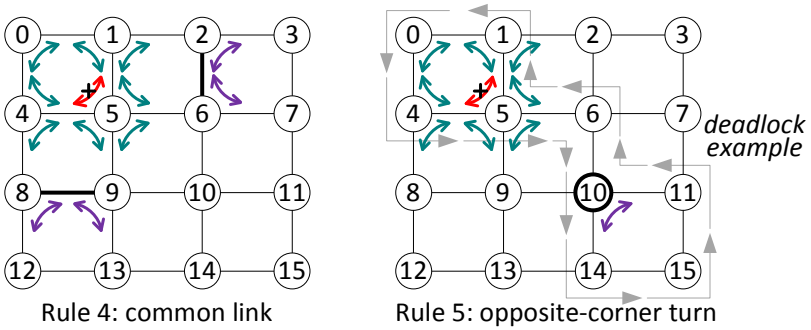
1) Basic rules identify which turns must be enabled because of a turn restriction on the surrounding cycle, node and links, and they are illustrated in Figure 7.5a.

- **Rule 1** (**cycle**) — *Once a turn in a cycle is disabled, all other turns in the same cycle should be enabled* so that all nodes in the cycle can still communicate.
- **Rule 2** (**node**) — *Once a turn in a node (router) is disabled, all other turns insisting on the same node should be enabled.*
- **Rule 3** (**link**) — *Once a turn adjacent to a link is disabled, the turn to the same link, on the opposite side with respect to this turn and not insisting on the same router should be enabled.*

In a fault-free mesh network, there is only one turn that should be enabled for each link affected by a disabled turn as a result of Rule 3. Note that Rules 2 and 3 are not necessary to guarantee the connectivity of the network, but any violation of these rules would lead to a superfluous number of turn disabling. For instance, if both the turns 1-4-3



(a) Basic rules



(b) Advanced rules

Figure 7.5: FATE's turn-enabling rules. Whenever FATE selects a new turn restriction, it can infer a number of other turns that should be enabled at other locations to obtain an optimal routing function (minimal number of disabled turns) faster. Basic rules (a) enable turns in adjacent locations, while advanced rules (b) impact remote turn locations. In the diagram for Rules 2 and 5, we show a deadlock cycle that we would obtain if we did not enable the turns indicated.

and 1-4-5 were disabled in the middle network of Figure 7.5a, then the network would still allow a dependency along the path 0→1→2→5→4→3→0 as shown in the figure (gray arrows), and we would need to include an additional turn restriction to break it. Similarly, with reference to the right network in Figure 7.5a, if turn 4-1-2 were disabled, a cyclic dependency would exist along the path 0→1→2→5→4→3→0.

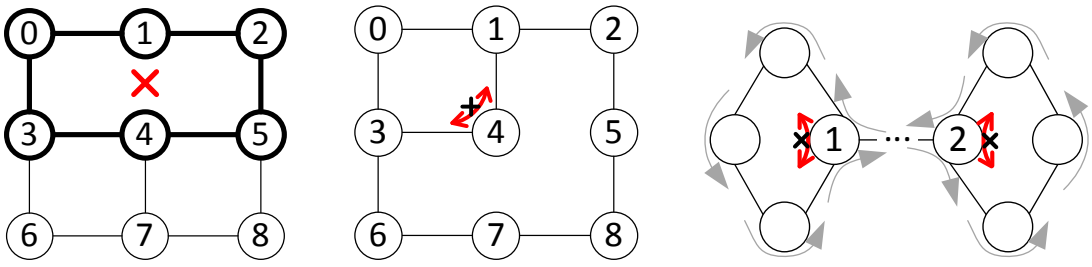
2) Advanced rules force FATE to enable turns that are remote with respect to the restricted turn. They allow to aggressively prune the search space towards a solution with a minimal number of disabled turns. Figure 7.5b illustrates the two rules below; we marked in red the restricted turn, in teal the turns enabled by the basic rules, and in purple the turns enabled by the advanced rule being illustrated.

- **Rule 4** (**common link**) — *If a cycle has only two undecided turns that share a common link, then all turns that are adjacent to that link and lie outside the cycle, should be enabled.*

This rule can be inferred from Rules 1 and 3. For a cycle with two undecided turns, by Rule 1, one of the two should be disabled. If these two turns are adjacent (i.e., sharing a link), disabling either of the two turns will always involve the shared link. We apply Rule 3 to this link so that we do not allow another cycle to also place its turn restriction on this link. An example is shown on the left side of Figure 7.5b, where the cycle 1-2-6-5 has two undecided turns after placing the turn restriction at 1-5-4: the turns 1-2-6 and 2-6-5, sharing the link 2-6. This shared link is also adjacent to two other turns: 3-2-6 and 2-6-7 (both belong to the cycle 2-3-7-6), which should be enabled by Rule 4. Note that Rule 4 can be applied to enable turns both on the horizontal and vertical directions. Also, it can be applied iteratively, to further reduce the disabled-turn-placement search space.

- **Rule 5** (**opposite-corner turn**) — *If two turns are located on opposite nodes in a cycle, and the two turns are not adjacent to any of the cycle's links, then only one of them can be disabled.*

We say that such two turns are in opposite-corner locations. The reasoning behind this rule can be understood by considering the example on the right part of Figure 7.5b: if we had disabled both the opposite-corner turns 1-5-4 and 11-10-14, then we could no longer avoid a deadlock. Indeed, by Rule 2, only two turns would remain undecided in the central cycle (5-6-10-9): 5-6-10 and 5-9-10, and by Rule 1, we would have to disable one of them. However, disabling either of these turns creates a cyclic dependency that could lead to deadlock. As an example, the figure shows the cyclic dependency we obtain if we disable turn 5-9-10, as well as the two opposite-corner turns. Note that it is possible to apply Rule 5



(a) Cycle recomputation (b) Mutual turn (c) Finding deadlocks

Figure 7.6: Extending turn-enabling rules to faulty topologies. The rules in Figure 7.5 can be applied in the presence of faults with a few modifications. For instance, (a) cycles must be recomputed in faulty regions, and (b) a turn shared by convex- and concave-shaped cycles should be used to break only one of the cycles. (c) A deadlock may occur when there is a path between two nodes belonging to different cycles and both nodes are the one with the disabled turn for its cycle.

repeatedly by considering increasingly larger cycles. For instance, the rule could be applied to the cycle 5-7-15-13, and force the south-east turn on node 15 to be enabled (assuming that we had a larger network where that turn existed).

7.3.1 Turn-Enabling Rules in Faulty Topologies

Among the basic rules, Rule 1 is the one that is affected the most by faults: cycles may become merged because of faults, as shown in the example of Figure 7.6a (link 1-4 is faulty). In the figure, the two cycles 0-1-4-3 and 1-2-5-4 no longer exist, and they are merged in the cycle 0-1-2-5-4-3. Moreover, faulty networks may have both concave and convex cycles (unlike fault-free mesh networks, which have only convex cycles) and Rule 1 must be appropriately applied in the case of concave cycles. When a turn is common to two cycles (e.g., turn 3-4-1 in Figure 7.6b), then disabling that turn can only be counted towards breaking one of the two cycles, not both.

Rules 2 and 3 remain unchanged for faulty topologies. Consider link 3-4-5 in Figure 7.6a as an example: if we were to disable the turn 2-5-4, Rule 3 would require the turn 5-4-7 to be enabled.

In addition, we limit the application of Rule 4 to links contributing to cycles that have not been affected by faults. For instance, with reference to the left network in Figure 7.5b, if the link 5-6 were to be faulty, we would not apply Rule 4 to link 2-6, because it contributes to a cycle that has been opened by a fault; but we would still apply the rule to link 8-9. The reason we limit the application of Rule 4 is that it could become complex to identify which turns should be enabled when a link spans multiple routers.

Finally, we simply apply Rule 5 as we described for regular meshes. Applying this rule in faulty networks allows us to aggressively prune the search for an optimal disabled-turn placement. However, some turns enabled are not necessarily causing deadlock in faulty networks, and thus they should not be enabled. If the enabled turns should have been disabled, our backtracking step (Section 7.4.3) would correct the situation. To avoid the backtracking, it is also possible to pre-emptively check whether a turn enabled via Rule 5 could lead to a deadlock configuration. Figure 7.6c shows how to check deadlock for this purpose: when there are two distinct cycles connected through a path starting at node 1 and ending at node 2, we cannot disable both the turns indicated in the figure, because this placement would enable the deadlock cycle shown in gray.

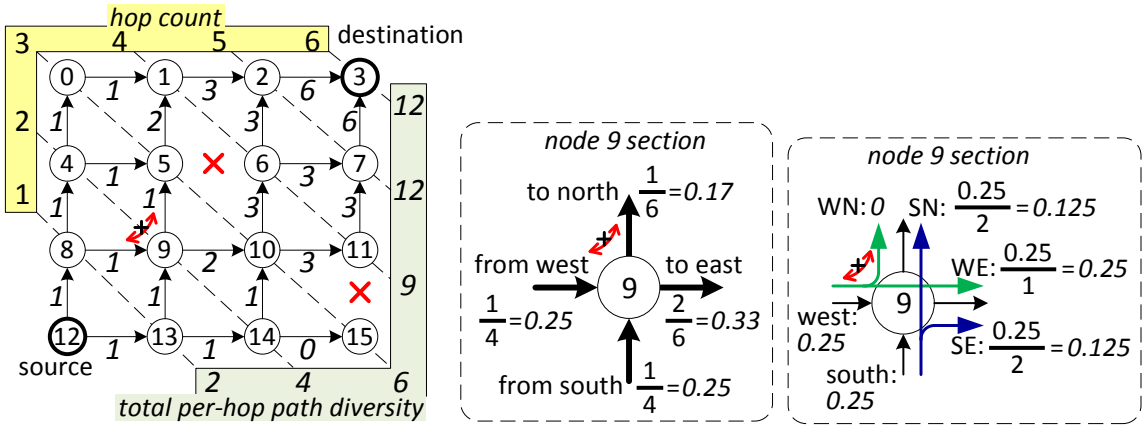
7.4 Route Computation

In this section, we propose a heuristic algorithm to identify deadlock-free routes with minimal routing restrictions. Also the routes identified by our heuristic algorithm do not entail any routing-induced disconnection (i.e., the algorithm provides valid routes for every connected source-destination pair). FATE relies on the information it receives about the application's communication patterns to strive to place turn restrictions on low-traffic links. When the FATE algorithm begins, all turns are undecided. Turn restrictions are then placed one at a time, starting from the regions transferring the most traffic. Upon placing each restriction, the turn-enabling rules are applied to enable the related set of turns. This process is repeated until each turn is either enabled or disabled.

To identify which turn to disable next, we first estimate the traffic load on each link, turn and cycle. We then disable the turn that minimally worsens the heaviest load-transferring link, choosing among turns on the heaviest load-transferring cycle. The intuition behind this choice is that we want to resolve the routing function first around the regions (i.e., cycles) transferring the most traffic so that we have plenty of flexibility in our choices. Within each region, we want to disable the turn that minimally affects the congestion in the hotspot.

7.4.1 Traffic Load Estimation

To estimate the load on each network's link, turn and cycle, we consider one source-destination pair provided by the application at a time. For each pair, we compute all the possible paths that packets can take from source to destination, and we then derive the fraction of traffic that would go through each link. The computation of all the loads proceeds with the four steps below (see Figure 7.7).



(a) Path diversity (b) Link load (c) Turn load

Figure 7.7: Link-load and turn-load estimation example. We compute traffic load of links and turns using path diversity. (a) The path diversity is calculated at each link considering fault locations and turn restrictions. (b) The link load is computed by dividing its path diversity by the per-hop path diversity. (c) The turn load is derived from the load of its input link and the number of permissible outputs.

Step 1: Compute path diversity. We calculate the number of different routes (i.e., path diversity) to reach each link from the source node. For instance, in Figure 7.7a, the east link of node 9 can be used by two different routes from node 12: 12→8→9 and 12→13→9, while only one route can use the north link of node 9: 12→13→9. In this process, we only allow minimal-length routes within the limits of the turn restrictions that are already in place.

Step 2: Compute link-load estimates. We now use the results of Step 1 to estimate the load on each link based on the path diversity available. We calculate the total path diversity at each hop from the source (as illustrated in Figure 7.7a), and divide the link's path diversity by the total diversity. Figure 7.7b shows the computation for all the links associated with node 9. For instance, the load of the north link is 0.17 (= 1 / 6).

Step 3: Compute turn-load estimates. To estimate the load at each turn, we divide the input load from the source direction of the turn by the number of output links allowed for that source. Figure 7.7c shows the computation for all the turns at node 9.

Step 4: Compute cycle-load estimates. For each cycle, the load is computed by simply summing the loads on all the turns in the cycle. For instance, the load for the south-east (SE) turn is 0.125 (= 0.25 / 2).

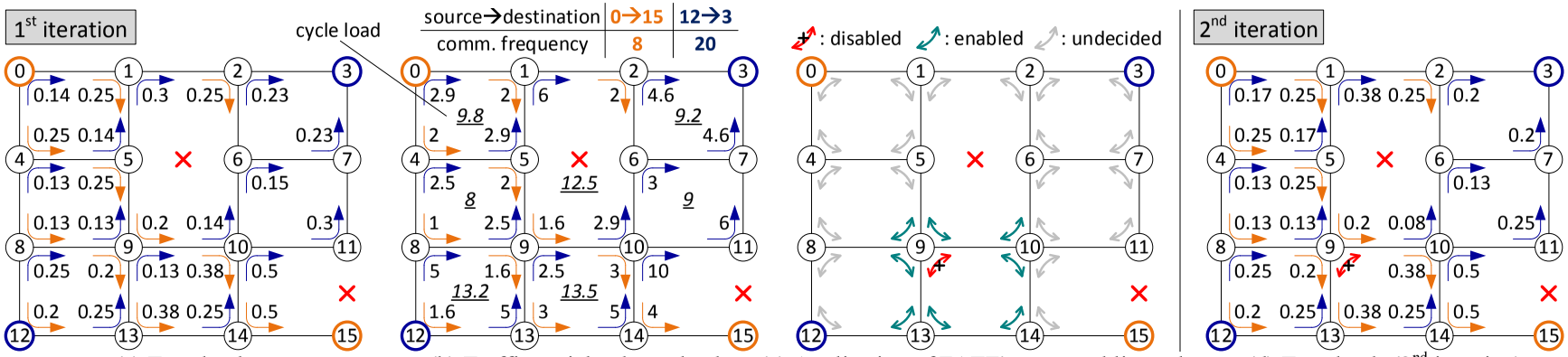


Figure 7.8: FATE's routing algorithm example. We show the first two iterations for a network with two failed links and an application with two communicating pairs. (a) Computation of turn loads. (b) Communication weights and cycle loads indicate that 10-9-13 is the most promising turn-disabling location. (c) Eight turns are then enabled by our turn-enabling rules. (d) Loads computation for the second iteration.

7.4.2 Iterative Turn-Restriction Decision

Once all load estimates have been computed, we can apply the FATE routing algorithm, as illustrated in Algorithm 7.1. Note that we weigh the load estimates by multiplying each estimate by the traffic weight associated to its source-destination pair. The algorithm starts by selecting a turn to disable, choosing the turn with the lightest impact on the heaviest link load, among those in the highest-load cycle (lines 2-4). Once the turn to be disabled is selected, we apply the turn-enabling rules to enable as many other turns as possible (line 5). If the set of turns enabled/disabled leads to a deadlock or a disconnected network (line 6), we backtrack, and update the list of conflicting selections (line 8). Our backtracking algorithm is discussed in more detail in the next subsection.

Algorithm 7.1 FATE routing algorithm

```
1: repeat until there is no undecided turn
2:   compute_link,turn,cycle_loads()
3:   cycle = cycle_with_heaviest_load()
4:   disabled_turn = turn_with_smallest_link_load_increase(cycle)
5:   enabled_turns = apply_turn_enabling_rules(disabled_turn)
6:   if ( not (check_deadlock() or check_disconnected()) )
7:     disable(disabled_turn), enable(enabled_turns)
8:   else update_conflict_history(), backtrack()
```

In designing our routing algorithm, we evaluated a few other strategies to select the next turn to be disabled: beside the one just described, we also tried (1) the turn with the absolute lowest load in the network, (2) the turn with the lowest load among those in the highest-load cycle, and (3) the turn connected with the highest-load link in the highest-load cycle. Experimentally, we found that those strategies performed slightly worse than the one we described.

Figure 7.8 illustrates the algorithm with an example. The application provided two communication pairs: 0→15 (shown in orange), with a weight of 8, and 12→3 (shown in blue), with a weight of 20. We first compute link and turn loads, as shown in Figure 7.8a. Then we apply the weights and compute cycle loads in Figure 7.8b. Cycle 9-10-14-13 is the one with the highest load. By analyzing each turn, one at a time, we find that the one that entails the smallest link-load increase is 10-9-13, so we disable it. Figure 7.8c shows the network after the application of the FATE's turn-enabling rules. At this point, 21 turns are left undecided, thus we start a second iteration by updating the link, turn and cycle load estimates as shown in Figure 7.8d.

7.4.3 Avoiding Illegal Routing Function (Backtracking)

As mentioned earlier, the FATE routing algorithm may require backtracking if the set of turn restrictions in place allows for deadlock (usually along a complex cycle) or disconnects the network. These issues may arise because the FATE's rules do not take into account all the implications of a turn restriction, but only the simpler ones so that their application is computationally cheap. When these situations occurs, we record the location of all restrictions and we add the current set of turn-disabling placements to a *conflict history*, which we use to avoid repeating the same configurations. While the majority of topologies and communication patterns in our experiments have successfully completed with only a small amount of backtracking, we found a few cases (less than 1%) that result in more than 1,000 iterations to finish. We believe these situations occurred because of our simplistic backtracking model that rolls back to the most recent decision, instead of a more intricate model that selects a promising roll-back point. This limitation is partially contained by a random restart technique that we implemented, similar to that in some SAT solvers. Upon a restart trigger, previous decisions are forgotten, and a new initial turn location is selected. We set the restart threshold to 1,000 backtracking events in our experiments.

7.5 Experimental Evaluation

We evaluated FATE with a cycle-accurate NoC simulator [112] modeling an 8×8 2D mesh NoC. The network's 3-stage routers are capable of look-ahead routing and speculative switch allocation (i.e., switch allocation is concurrent with VC allocation). Each input port within a router contains 2 VCs per protocol class, and 5 flits per VC, unless otherwise noted. We utilized 3 protocol classes to support the MESI cache coherence protocol when running SPLASH-2 benchmarks, and a single protocol class when evaluating synthetic traffic patterns. We apply the FATE algorithm to place turn restrictions, then deploy a minimal adaptive routing approach where packets choose at each router which direction to take among those enabled. In addition, we deploy a local congestion monitoring scheme based on the credit count so that output channels towards non-congested input buffers are favored.

We analyzed FATE's performance both on fault-free and faulty 2D mesh networks, injecting a varying number of link faults in the latter scenario. Specifically, we experimented with configurations containing 1 (1%), 3 (3%), 6 (5%), 11 (10%) and 17 (15%) faulty links, and averaged the results over 10 different random sets of fault placements for each data point. Note that we do not consider configurations that lead to disconnected nodes, as this situation would require thread migration support when running parallel benchmarks.

Table 7.1: gem5 configuration for SPLASH-2 traces

Core	64 cores, x86 ISA, 2GHz, out-of-order, 8-wide issue
Cache coherence	MESI, CMP directory, 64-byte block
Network	8×8 mesh, 3 classes/port, 2 VCs/class, 5 flits/VC
L1 cache	Private, 16kB inst. + 16kB data, 4-way set, 3-cycle lat.
L2 cache	Shared, distributed, 256kB/node, 16-way set, 15-cycle lat.
Memory	1GB/controller; 1 controller at each mesh corner

This requirement, in turn, would make it difficult to reason about performance scaling due to a different number of active cores in different configurations. Even though we model only link failures, FATE is equally effective in tackling routers' logic failures by mapping such failures to one or more links connected to the failed routers [157]. Please note that we evaluated our solution under various corner-case topologies to take into account unpredictability of fault locations. These random topologies include, for instance, nodes with only a single surviving link, cycles consisting of more than 10 nodes, etc.

We evaluated our testbed with both synthetic traffic [112] and traces from the SPLASH-2 benchmark suite [204]. We used 5 different synthetic patterns: bit complement (*bitcomp*), bit reversal (*bitrev*), shuffle (*shuffle*), transpose (*transpose*) and uniform random (*uniform*). Our synthetic traffic consists of a mix of equal amounts of 1- and 5-flit packets. The 11 SPLASH-2 traces we experimented with, on the other hand, were obtained from full-system simulation using gem5 [56] in the *syscall* emulation mode. Our gem5 model was configured as shown in Table 7.1. The traces were collected for 10 million cycles after spawning threads and initializing caches. The traffic weights were calculated by counting the number of flits per each source-destination pair. The latency values were capped at 1,000 cycles during the trace-based simulation to avoid extremely large latencies due to network saturation. We did not evaluate FATE with the PARSEC benchmark suite, because PARSEC is known to underutilize the network, and its traffic is more evenly-distributed than SPLASH-2. We expect PARSEC to yield results similar to those of *uniform*.

7.5.1 Performance on Faulty Networks

We compared our solution against fault-tolerant, application-oblivious routing solutions that are based on the construction of spanning trees: up*/down* with breadth-first search (BFS) [178, 22], and up*/down* with depth-first search (DFS) [176]. For BFS, we experimented with 4 different configurations: each configuration placed the root node at a different corner of the mesh. Then we calculated the geometric mean of the results obtained over all four configurations. For DFS, we implemented the two heuristics from [176]: *Ht1*

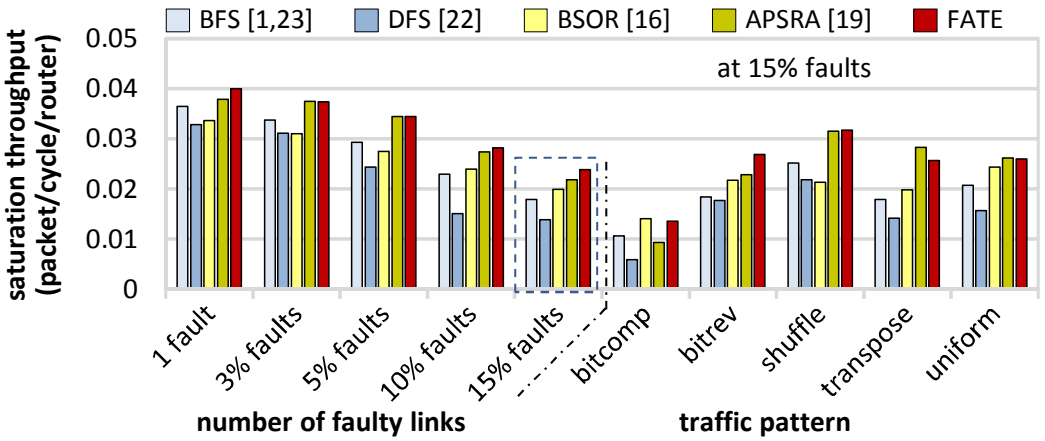


Figure 7.9: Saturation throughput for synthetic traffic patterns over various fault rates and traffic patterns. FATE provides 10%-33% better saturation throughput over BFS, and up to 9% better throughput over APSRA.

for selecting the root node, and *Fv* for choosing a tree-spanning direction.

We also compared our solution against two existing application-aware routing solutions: Application-Specific Routing Algorithm (APSRA) [155] and Bandwidth-Sensitive Oblivious Routing (BSOR) [116]. Both algorithms are modified for distributed routing. For APSRA, we applied APSRA's turn cost calculation instead of our load estimation method in Section 7.4.1. For BSOR, we first placed turn restrictions according to the negative-first turn model, then applied the Dijkstra's shortest path algorithm. We tried four different turn restrictions, each with four rotations. The algorithm is then applied iteratively by reducing the link capacity to aggressively optimize for congested links, as shown in [116].

Figure 7.9 reports the average saturation throughput (i.e., when latency reaches 3 times the zero-load latency) across various fault rates and traffic patterns. In the left half of the figure, we observe that the performance of our scheme degrades more gracefully than both application-oblivious spanning-tree solutions (BFS and DFS) as faults increase. FATE achieves a 10% improvement over BFS when there is only one faulty link. Although the network at this low fault rate maintains an almost-regular topology, FATE still offers a better throughput than BFS by leveraging distinct traffic patterns. FATE's improvement over BFS increases to 33% when 15% of the links are faulty. At this high fault rate, the spanning-tree solutions often fail to ensure minimal routing restrictions, leading to high performance loss.

FATE also achieves consistently higher throughput at all fault rates than both APSRA and BSOR. While it shows comparable throughput with APSRA at low fault rates, FATE provides a 9% higher throughput at the 15% faulty-link rate. We believe that this is because APSRA's traffic estimation becomes inaccurate at high fault rates, as the estimation relies

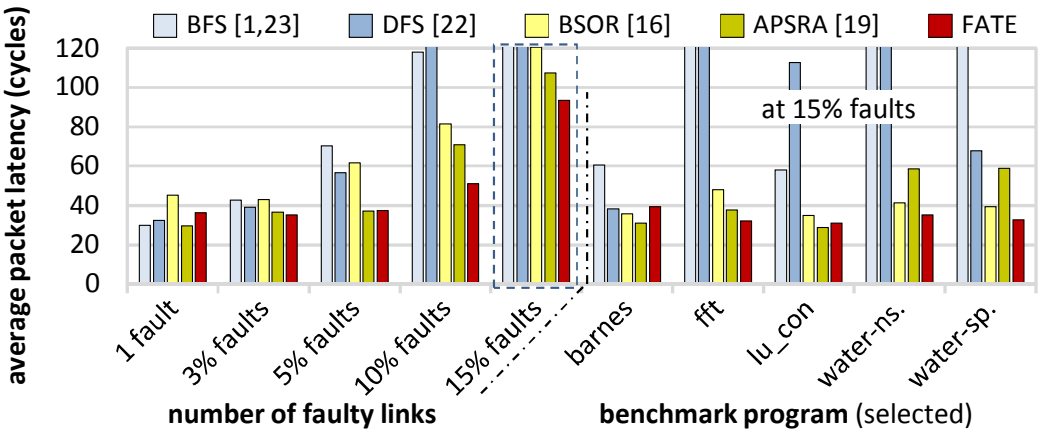


Figure 7.10: Packet latency for SPLASH-2 traces over various fault rates and benchmarks. Except for the 1-fault case, FATE shows 18%-59% improvements in packet latency over BFS, and up to a 13% improvement over APSRA.

on the path diversity between sources and destinations using source routing. In contrast, FATE estimates traffic load by considering hop-by-hop routing-decision using distributed routing. Finally, BSOR provides lower throughput than both APSRA and FATE, most probably because it lacks a dynamic approach to congestion management.

FATE also performs better than BFS across various traffic patterns at the 15% faulty-link rates, as shown in the right half of Figure 7.9. We observe that traffic patterns where packets utilize a few turns more frequently (e.g., *transpose* and *bitrev*) are those that benefit the most from FATE. Our solution provides at least at-par performance in the patterns where turns are used evenly (e.g., *uniform*).

Figure 7.10 reports the average packet latency from our trace-driven SPLASH-2 simulations across various fault rates and benchmarks. As shown in the left half of the figure, FATE experiences negligible latency increase up to the 5% faulty-link rate. Beyond that point, the latency increase is more significant. However, note that how the latency increase begins at lower fault rates: it is much steeper in BFS and DFS routing. This dramatic increase comes from the earlier saturation effect in resource-constrained faulty networks. Note also that in networks with only one faulty link, FATE performs worse than other solutions. This result is due to the high-impact contribution to the average by *ocean_con*, which exhibits phases with very high injection rates for short time periods. These phases are not representative of the entire benchmark execution, and hence our solution is unable to optimize for them. In addition, FATE attains lower latency than APSRA at all fault rates except one faulty link. The two solutions show comparable latency until the 5% faulty-link rate. At the 15% rate, however, FATE shows 13% lower packet latency than APSRA. FATE also outperforms BSOR at all fault rates.

Finally, we show results for five selected benchmarks at the 15% faulty-link rate on the right side of Figure 7.10. FATE consistently provides much lower latency than BFS and DFS. For instance, when running *fft*, each node accesses frequently memory nodes located at the corners of the mesh, and communicates with a few other nodes. As a result, FATE enables more routes among these frequently communicating nodes. On the other hand, both BFS and DFS are application-oblivious, so their disabled-turn placements are not so favorable to those nodes.

7.5.2 Performance on Fault-Free Networks

We compared FATE's fault-free operation and congestion management capabilities against 3 fully-adaptive routing techniques: DyXY [127], NoP [49] and RCA1D [92]. For those solutions, we implemented deadlock detection based on timeout, and reserved one VC for deadlock recovery. We also considered prior fault-tolerant solutions and application-aware solutions for fault-free networks.

Figure 7.11 shows the average saturation throughput of FATE against all the solutions above across 3 different VC settings (3, 4 and 6 VCs), and synthetic traffic patterns. As shown in the left part of the figure, FATE outperforms DOR: FATE achieves a 4.5% improvement when using 3 VCs, and up to 23% with 6 VCs. FATE's advantage grows with the number of VCs since it manages resources more efficiently than oblivious routing techniques.

However, the reverse trend holds when FATE is compared against fully-adaptive solutions, i.e., DyXY, NoP and RCA1D. FATE's improvement over the fully-adaptive solutions diminishes as the number of VCs increases, because the cost of reserved VCs in the fully-adaptive solutions is amortized when more VCs per class are available. For networks with only 3 VCs, FATE shows a 33% improvement over DyXY and a 21% improvement over RCA1D. This advantage is lost at 4 VCs, while at 6 VCs FATE provides 13% lower throughput than RCA1D. In the area- and power-constrained on-chip environment, NoCs with fewer VCs are more prominent, and hence our solution is widely applicable.

We further analyze FATE's fault-free performance by considering various synthetic traffic patterns. The right part of Figure 7.11 shows the average saturation throughput for networks with 3 VCs under various traffic patterns. Our solution outperforms the fully-adaptive solutions for most patterns in this setting: the exceptions are *bitcomp* and *uniform*, where DOR is the best performer. Considering the benefits of FATE for faulty networks (Section 7.5.1), we believe that a slight performance loss on fault-free networks for rare adverse traffic patterns is acceptable.

Note that in our experiments, FATE leverages local congestion information to adap-

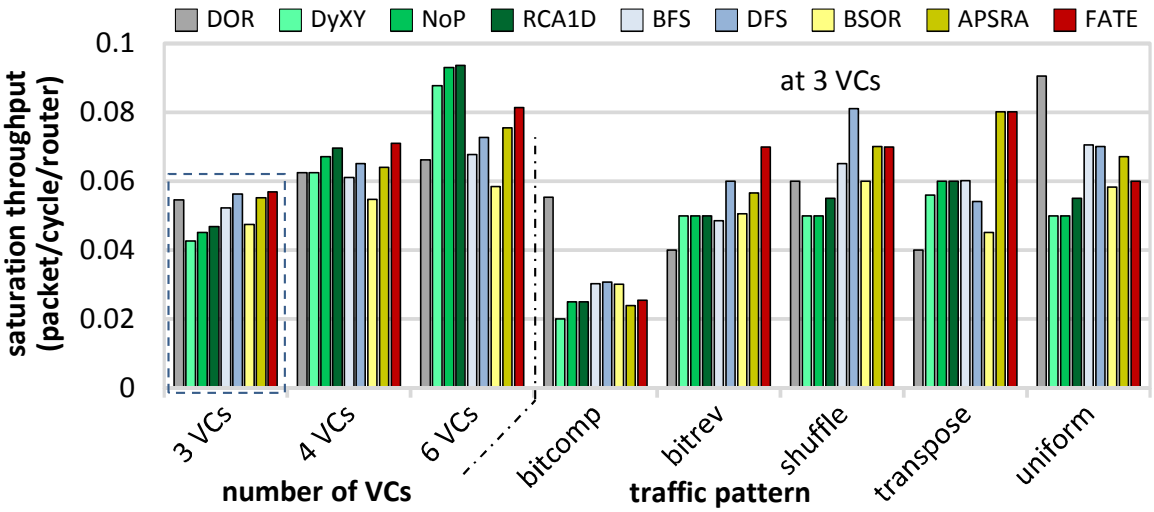


Figure 7.11: Saturation throughput for synthetic traffic patterns over a varying number of VCs and traffic patterns.

tively choose better routes at runtime. However, it has been shown in prior research [49, 92] that global network-level congestion monitoring schemes deliver superior performance. In future work, we plan to evaluate the benefits of applying some of these schemes to FATE.

7.5.3 Overheads

Routing-function computation overhead. We evaluated the overhead of computing the routing function, and compared our findings against APSRA [155]. Table 7.2 reports the average computation time to derive a routing function, for both FATE and APSRA. In evaluating computation time, we only included the total time spent in estimating traffic load (Section 7.4.1 for FATE), since that is by far the major contributor to the algorithm's computation time, for both ASPRA and FATE. The other activities contributing to the routing function computation (e.g., backtracking, deadlock checking) are minor contributors and identical for both solutions. Execution times were measured by averaging over 5 executions for each routing function computation on an Intel Xeon E5520. Overall, it can be noted that FATE is a significantly faster solution than APSRA.

In the right portion of Table 7.2, we also compare the number of attempts of turn-disabling placement, averaged over 10 different faulty topologies and 16 traffic weights (5 synthetic traffic patterns and 11 SPLASH-2 benchmark traces) at each fault rate, in order to gain insights on the gap between the computation of FATE and APSRA. This value is the cumulative sum of each turn-disabling placement attempt, including all the placements that had to be removed because they led to a conflict or deadlock. Note that FATE's number of attempts is minuscule compared to APSRA: this result comes from

Table 7.2: Computation overhead for FATE and APSRA

	Average time (sec)		**Average number of placements attempted**		**% runs reaching 200k cap**	
	APSRA	FATE	APSRA	FATE	APSRA	FATE
Fault-free	733	3.61	71,321	117	19%	0%
1 fault	973	3.27	94,639	107	31%	0%
3% faults	1,044	3.29	107,956	107	41%	0%
5% faults	1,149	3.21	120,877	105	48%	0%
10% faults	1,257	3.62	151,802	118	69%	0%
15% faults	1,223	2.93	159,667	96	74%	0%

our turn-enabling rules, which greatly prune the search space for an optimal set of turn-disabling placements. In many situations, APSRA's routing algorithm made extremely large turn-disabling placement attempts before reaching a stable solution. We capped those algorithm's runs at 200,000 placement attempts and we report the fraction of occurrences where we reached the cap value. Note that FATE never had a case that required 200k placement attempts, while APSRA had a noticeable fraction of algorithm's runs that went over the limit.

Although FATE requires much less computation than existing application-aware routing, it may still not be sufficiently fast because of its software-based computation. In those situations where recovery performance is of essence, alternative hardware-only solutions (e.g. retransmission [147] and BFS-based routing [22]) can be deployed concurrently with FATE, while it executes in software. Upon FATE's completion, its solution can replace the interim recovery solution.

Area and power overheads. FATE requires a reconfigurable routing infrastructure (e.g., routing tables) to recompute the routing function at runtime. We deploy routing tables as shown in [22, 116], one for each router. Each table contains N entries where N is the number of nodes, and each entry contains four directional 2-bit fields (8 bits per entry). The 2-bit fields are used to prioritize valid output directions by using the number of hops to the destination. In addition, we utilize four routing-restriction bits (similar to [59]) to avoid making invalid decisions at each router. Thus, to store the computed routing function in memory, we require $N\times(N\times8+4)$ bits per application of the FATE algorithm. Moreover, we use the number of used credits as our congestion metric: this is often already available in routers and comes at no extra cost.

We evaluated the area overhead of routing tables and route-computation logic, targeting the Nangate 45nm library using Synopsys DC. The microarchitecture of a baseline router is configured as specified in Table 7.1 with an operating frequency of 400MHz. With this

configuration, our routing computation adds approximately 6% area overhead, mostly for the routing table. Note that the other fault-tolerant, adaptive routing solutions we compared against (BFS, DFS and APSRA) entail similar overhead, as they utilize similar routing intrastructures as FATE.

While we have not evaluated FATE's power overhead, we provide here a qualitative comparison. Our solution entails significantly less computation than APSRA while, at the same time, producing routing functions that lead to lower packet latency. Thus we believe FATE would consume less power than APSRA. When comparing to BSOR, BFS and DFS, the two trends are in opposition: we do attain lower packet latency, at the cost of higher computation time for the routing function. Thus we only provide an overall power gain if we can absorb the additional power cost of the computation over the benefits in packet latency.

7.6 Related Work

Fault-tolerant routing. As discussed in Section 2.3, fault-tolerant, deadlock-free routing solutions have been extensively investigated in the past. Among them, FATE shares most traits with Ariadne [22], uDIREC [157], CBCG [169], and Hermes [110], as all of them put no constraints on the number and location of faults. In contrast, these prior solutions do not take traffic flow into consideration to optimize for throughput.

Similar to the above on-chip solutions, in the off-chip network domain, topology-agnostic routing algorithms [85, 178, 176] focus mostly on placing routing restrictions using topological characteristics, while ignoring traffic flow during the placement. On the other hand, we exploit communication patterns to find better routing restrictions.

Application-aware routing. Application-aware routing solutions tune the routing function to traffic flow. APSRA [155] strives to meet the bandwidth requirement of an application, while achieving deadlock-freedom by breaking cyclic dependencies. However, deploying this solution to identify optimal routes in faulty networks has an extremely high computational cost. Addressing APSRA's limitation, ACES [69] investigates a quick application-aware routing heuristic in irregular topologies by partitioning VCs to break cycles. LBDRx [59] develops a resource-mapping tool and a routing algorithm for SoC applications, while reducing routing overheads by leveraging logic-based routing instead of routing tables. Note that all of the above solutions use adaptive routing to mitigate congestion. In contrast, BSOR [116] selects load-balanced, oblivious paths via either mixed integer linear programming (MILP) or Dijkstra's shortest path algorithm to approximate

optimally balanced paths. Similarly, ETM [182] reduces the computation requirements of the MILP problem with the aid of a genetic algorithm. These latter two oblivious routing solutions, however, suffer from low performance as they do not manage runtime congestions well. Although all of these application-aware solutions can be deployed in faulty networks, they often require either intractable computation or extra hardware.

Adaptive routing. Adaptive routing techniques [127, 49, 92] can be used to minimize the performance impact of faulty networks. They mitigate interference among packets by routing some of them through underutilized network links. Application-aware routing techniques can manage congestion more efficiently when they employ runtime congestion monitoring as in these adaptive routing techniques. Various congestion monitoring schemes have been explored in 2D mesh NoCs. For example, DyXY [127] observes buffer occupancy in adjacent routers, and then favors forwarding packets to routers with more vacancies. NoP [49] and RCA [92] extend this idea by monitoring congestions in routers farther away. All such solutions are fully-adaptive, and thus, require dedicated VCs to guarantee deadlock-freedom, a costly endeavour in area- and power- constrained NoCs.

7.7 Summary

This chapter proposed FATE, a lightweight route-computation algorithm that calculates high-performing, adaptive routes for faulty on-chip networks. The algorithm takes into account the knowledge of an application's communication patterns to find application-optimized routes, which offer high throughput and low latency. FATE's lightweight computation is enabled by turn-enabling rules that decompose a network-wide route-search process into localized turn-restriction decisions, avoiding route-search efforts that are unnecessary, e.g., routes with either a deadlock or a disconnection. On top of the turn-enabling rules, we leverage an iterative route-search process where each iteration determines one turn restriction. In choosing the most promising turn-restriction locations, we strive to balance traffic loads evenly among healthy network resources, using our novel load-estimation metric.

Experimental results show more than two orders-of-magnitude speedup, compared to a state-of-the-art application-aware routing solution. The results also show improvements in throughput and latency for synthetic traffic patterns and SPLASH-2 benchmark traces, over state-of-the-art fault-tolerant and application-aware routing solutions. In addition, FATE keeps a low silicon area overhead profile, at only 6%. Finally, we demonstrated that FATE can be used even for fault-free networks, showing saturation throughput comparable

to several mesh-based adaptive routing solutions, topology-agnostic routing solutions, and application-aware routing solutions.

We have now come to discussing the last critical type of components in SoCs: accelerator modules. In this regard, the next chapter addresses the challenge of verifying complex interactions among accelerators in a system on chip. This verification is one of the last and most complex phases of SoC verification at design time, and may have a high impact on the product development timeline.

CHAPTER 8

Test Generation for Verifying Accelerator Interactions

This chapter investigates evasive errors in accelerators, when they are integrated into system-on-chip (SoC) designs. In a typical SoC design flow, accelerators are individually verified before being integrated into the system. These individually-verified accelerators are then brought together to form a system; this integration process strives to check if the entire system operates as expected in as many use-case scenarios as possible. This chapter focuses on one of the key verification challenges in this integration process: the verification of accelerator interactions.

Accelerator interactions are becoming more diverse, as SoC designs integrate increasingly many specialized accelerators that can efficiently execute certain domains of applications. While there are numerous accelerator interactions that an SoC design may exhibit, we observe that there are relatively fewer interactions that the SoC design *is likely to exhibit* during its lifetime. To identify such probable interactions, we leverage a high-level model of the SoC design, available in an early design phase, and we collect and analyze execution traces (e.g., memory-access sequences) from the model. Then, to capture essential interactions among accelerators, we propose to *dissect long accelerator interactions* observed in the collected execution traces, into *basic interaction elements*. These basic interaction elements represent elementary relationships among accelerator executions (e.g., concurrent or sequential execution). These elements are then *reassembled to create synthetic tests*, which will be run on the hardware under verification (e.g., RTL models, gate-level netlists). Note that the hardware under verification is usually much more detailed and complex than its high-level model.

The decomposition and reassembly of accelerator interactions enable a novel accelerator-interaction verification framework, called AGARSoC, **A**utomated Test and Coverage-Model **G**eneration for Verification of **A**ccelerator-**R**ich **S**ystems **o**n **C**hip. AGARSoC is a methodology and a set of tools that guide the verification of accelerator-rich SoCs toward high-priority accelerator interaction scenarios. From a high-level model of the SoC design,

AGARSoC extracts and generates coverage models and high-quality test programs that hit the coverage goals. The generated programs are much more compact than the original software suite they are derived from, and hence require fewer simulation cycles. In addition, AGARSoC is highly versatile for analyzing test runs, adapting verification goals, and generating test programs with only minimal input from engineering.

8.1 Background and Motivation

Accelerators represent a variety of processing units and are often used in different meanings in the literature. At one end of the spectrum, they are often used to indicate coarse-grained processing units other than microprocessor cores such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs). Each of these processing units are capable of computing many different applications. Thus, they are, to some degree, general-purpose processing units. However, they are usually very efficient at processing certain domains of applications. For instance, GPUs are specialized to process data-parallel applications. FPGAs are efficient at performing complex operations on data streams.

At the other end of the spectrum, accelerators indicate small, fine-grained computation units, such as data compression engines and cryptographic engines [99]. This type of accelerators are also known as coprocessors or intellectual property (IP) blocks. Each of these computation units is designed to perform only a specific computation. For instance, the IBM POWER9 processor includes a couple of AES engines that support various encryption modes and key lengths. These AES engines are connected to the on-chip interconnect. Here, we focus on accelerators similar to the latter category (i.e., fine-grained accelerators).

Accelerator execution model. We assume that the executions of accelerators are governed by microprocessor cores. Specifically, an execution of an accelerator is initiated by a microprocessor core. The accelerator then reads the input data from the memory, performs the computation, then writes the computation results back to the memory. Afterward, the accelerator notifies the microprocessor core when it finishes the computation.

The microprocessor core configures the accelerator before launching a task. The configuration generally includes (1) where the accelerator must retrieve input data, (2) where the accelerator must store its computation results, (3) the mode of computation, etc. Usually, accelerators support multiple modes of computation, and these modes are specific to accelerator implementations.

Verifying accelerator interactions. It is common that tens or hundreds of accelerators are integrated in an SoC design. Each of these accelerators is designed and verified individually; this design and verification process is often agnostic to other accelerators integrated in the SoC design. This individual verification process often assumes an ideal scenario with respect to the remainder of the system, i.e., no interaction or interference from other accelerators. This verification method is fairly efficient in finding bugs in the early SoC integration phase where accelerators are integrated into the system for the first time. However, this method often misses many corner-cases where multiple accelerators interfere with each other. The missing corner-cases can cause errors that manifest themselves only in very rare circumstances.

Later in the SoC integration phase, accelerators are more thoroughly verified by leveraging randomized verification techniques. For instance, accelerators can be connected to an on-chip network that simulates random interferences in the system, such as varying packet latency or packet loss. While randomization techniques have been very successful in detecting corner-case errors, they might not be the most efficient verification method. Due to the random nature, it is hard to guarantee the 100% verification coverage. Thus, they rely on an extensive amount of executions, many of which have been already verified in the past. Here, we advocate a directed verification approach; we strive to find corner-cases that the system *may experience*. In designing our directed verification solution, we consider two types of accelerator interactions:

1. Direct interactions – for instance, an accelerator takes as input the outcome of another accelerator. This kind of interactions are frequently found in image and video accelerators, or in multithreaded applications.

2. Indirect interactions – for instance, an accelerators is run while other accelerators are running independently. Such concurrent executions may create contention to shared system components (e.g., memory subsystems, on-chip networks). This kind of interactions are common in multitasking environments.

8.2 AGARSoC Overview

We recognize a need for tools to manage the complexity of verifying the large space of accelerator interactions in accelerator-rich SoCs. We observe that even though the space of possible interactions in an accelerator-rich SoC is large, the actual interactions that are exercised by software are much more limited. It is of paramount importance to get these likely-exercised interactions correct as they affect customers the most. By analyzing soft-

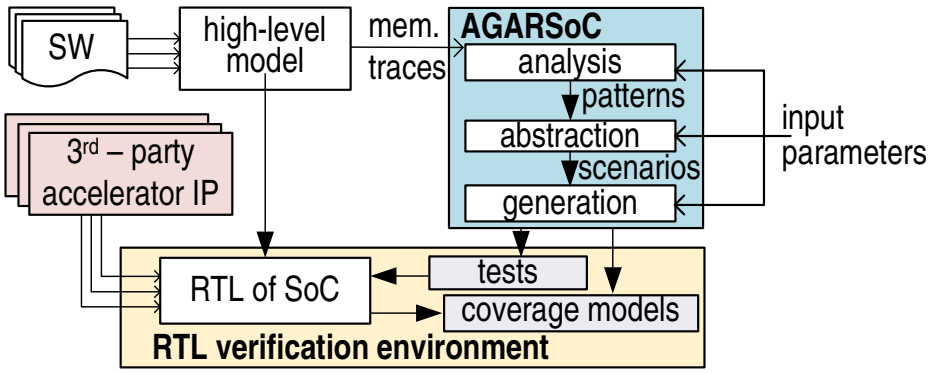


Figure 8.1: AGARSoC overview. We analyze the execution of software on a high-level model of an accelerator-rich SoC to identify high-priority accelerator interaction scenarios for verification. We then generate corresponding coverage models and compact test programs, which are used to guide the verification of the RTL for the SoC. AGARSoC is mostly automated and can be flexibly controlled by parameters supplied at runtime.

ware execution traces from high-level models, we propose to identify the accelerator interaction scenarios that are likely to be executed by software and thus are of highest priority for verification. We then generate coverage models that capture these scenarios and test programs that exercise these scenarios. Specifically, we identify basic elements of high-priority interactions, where the basic elements capture concurrent execution scenarios and sequential execution scenarios.

Our proposed solution, illustrated in Figure 8.1, leverages HW/SW co-verification environments where software to be shipped with the SoC is developed and tested on high-level simulation models, concurrently to RTL development. By using our generated test programs, the integration team can significantly cut down the amount of effort and simulation cycles spent on RTL verification. In addition, our automated tools simplify adjusting verification goals as new software becomes available during the verification lifetime. Our solution is designed to guide the verification of accelerator interactions and hence is complementary to approaches that target other goals.

8.3 Capturing Software Behavior

Accelerator-rich SoC architectures integrate several accelerators, general-purpose cores, and peripheral components as illustrated in Figure 8.2. Each accelerator in the system is designed to execute a specialized task quickly and efficiently. Accelerators for cryptographic hashing, video encoding/decoding, and image processing are already prevalent in current SoCs. Software executing on the general-purpose cores launch tasks on the accelerators by writing task parameters to the memory-mapped accelerator configuration registers. A

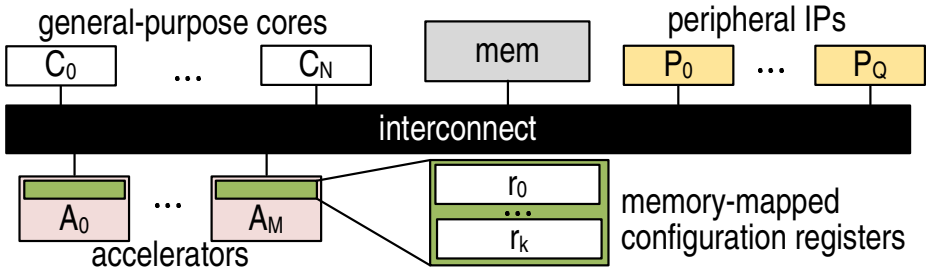


Figure 8.2: Accelerator-rich SoC architecture. Multiple cores, accelerators, memories, and peripheral components are integrated in a chip through an interconnect. These components interact via memory operations, which are routed by the interconnect to the proper destinations. Memory-mapped registers in the accelerators are used to configure accelerators for execution.

launched accelerator performs its task, possibly operating on a chunk of data in memory. Upon completion of its task, an accelerator notifies the software, often by writing its status to a memory location specified during configuration.

To capture software behavior, AGARSoC analyzes memory operation traces collected from software running on a high-level model. These traces include regular read and write accesses to memory from processor cores and accelerators, write operations to accelerator configuration registers, and special atomic memory operations for synchronization. Each access in a memory trace includes the cycle, memory address, type, size, and data for the access. The individuality of each accelerator and the existence of multiple memory transactions for configuring and executing an accelerator are features unique to accelerator-rich SoCs. Previous works on analyzing software behavior focus on capturing performance and target processor cores alone [89, 95].

8.3.1 Analyzing Memory Traces

We scan the memory traces from the processor cores to detect patterns of memory accesses to accelerator configuration registers, which we refer to as *accelerator configuration patterns*. We also extract *accelerator access patterns* from accelerator memory traces and match them with their triggering configuration patterns to get *accelerator execution patterns*. We group these accelerator execution patterns into *execution classes* based on their configured modes of operation.

AGARSoC's analysis is a mostly automated procedure with minimal engineer input. Engineers specify the memory mappings for the different components, which is obtained from the SoC configuration. We expect accelerator designers to provide descriptions of start and end conditions for accelerator configuration and access patterns, as well as specify

which configuration writes define a mode of operation for their accelerator. Integration engineers can then encapsulate these conditions as Python functions and easily pass them as arguments to AGARSoC's analysis tools. Below are examples of designer-specified conditions for the multi-writers, multi-readers (MWMR) protocol in the Soclib framework [196], encapsulated in the Python functions shown in Figure 8.3:

- *"An accelerator signals task completion by writing a non-zero value to a memory location written to configuration register 0x4c".*
- *"Writes to configuration registers 0x44, 0x48, 0x60, 0x50, and 0x5c define mode of operation."*

```
def MWMREndDetector(configPat, accOp):
    endLoc = configPat.accesses[0x4c].data
    if accOp.addr != endLoc or not accOp.isWrite():
        return False
    return accOp.data != 0

def MWMRConfigHash(configPat):
    mode = [configPat.target]
    for reg,acc in configPat.accesses.iteritems():
        if reg in [0x44, 0x48, 0x60, 0x50, 0x5c]:
            mode.append(acc.data)
    return tuple(mode)
```

Figure 8.3: User-defined functions. `MWMREndDetector` returns `True` if a particular accelerator memory operation is signaling the end of a task. `MWMRConfigHash` returns a hashable tuple that contains the mode of execution. The `configPat` (accelerator configuration pattern) and `accOp` (accelerator memory operation) data structures are provided by AGARSoC.

8.3.2 Identifying Interaction Scenarios

We identified two types of interaction that are likely to lead to scenarios that were not observed during independent verification. Firstly, concurrently executing accelerators interact indirectly through shared resources. Any conflicting assumptions about shared resource usage manifest during concurrent executions. Secondly, the state of the system after one accelerator execution affects subsequent accelerator executions. We develop heuristics that identify concurrent and sequential accelerator interaction scenarios by analyzing the observed execution patterns. We detect and retain only execution classes in our scenarios, instead of the actual execution instances. Thus, our heuristics avoid capturing redundant scenarios that do not belong to a diverse set of interactions.

Concurrent accelerator interaction scenarios gather accelerator execution classes whose execution instances have been observed to overlap. The overlap is detected by comparing the start and end times of the accelerators' execution patterns. While start and end times observed from a high-level model may not be exactly identical to those observed in RTL, they represent **the absence of ordering constraints** and therefore indicate a possibility of concurrent execution on the SoC. Concurrently executing accelerators interact indirectly through shared resources (interconnect, memory, etc.) leading to behaviors that may not have been observed during the independent verification of the accelerators.

Sequential accelerator interaction scenarios are either intended by the programmer (program ordered, and synchronized) or coincidentally observed in a particular execution. We infer three types of sequential accelerator executions:

1. Program-ordered – Non-overlapping accelerator execution classes invoked in program order from the same thread, detected by comparing start and end times of accelerator executions invoked from the same thread.

2. Synchronized – Accelerator execution classes whose instances are invoked from multiple threads, but are synchronized by a synchronization mechanism. We detect these by first identifying synchronization scopes (critical sections) from core traces and grouping accelerator invocations that belong in scopes that are protected by the same synchronization primitive. For lock-based synchronization, for instance, we scan the memory traces to identify accelerator invocations that lie between lock-acquire and lock-release operations. We group the accelerator execution classes for these invocations by the lock variables used, giving us groups of synchronized classes.

3. Observed pairs – Pairs of accelerator execution classes whose execution instances were observed not to overlap. Unlike program-ordered scenarios, which can be of any length, observed pairs are only between two execution classes.

8.3.3 Abstract Representation

Following the analysis and scenario detection steps, AGARSoC creates a representation of software execution that abstracts away execution-specific information irrelevant to our goal, such as specific memory locations and actual contents of data manipulated by the accelerators. Our representation contains three major categories: accelerator execution classes, concurrent scenarios, and happens-before (sequential execution) scenarios. Each entry in these categories contains a unique hash for the entry and a count of how many times

it was observed in the analyzed execution. The entries for accelerator execution classes and concurrent scenarios are directly obtained from the analysis. The happens-before category contains combined entries for all unique observed pairs and program-ordered sequences. In addition, we generate all possible non-deterministic interleavings of synchronized accelerator executions. The entries in the happens-before sections have counts of how many times they were observed in the execution traces. For any generated ordering that has not been observed, we store a count of 0.

Figure 8.4 illustrates an example of how a software execution is turned into an abstract representation. The SoC in this example has 3 cores and 3 accelerators. The software threads running on these cores trigger multiple accelerator executions, labeled 1 – 6 in the figure. A circle, a diamond, and a triangle are used to represent invocations of A_0, A_1, and A_2, respectively. The different colors indicate different modes of operation. AGARSoC will identify 5 different classes of accelerator execution corresponding to the different accelerators and modes observed during execution. Executions 1 and 4, 3 and 5, and 3 and 6 overlap, resulting in the three concurrent scenarios in the abstract representation. Similarly, AGARSoC identifies two sequences of program-ordered execution classes. Lock-acquire and lock-release operations using the same lock variable, indicated by locked and unlocked padlocks, respectively, mark one group of synchronized accelerator executions. AGARSoC generates all 6 possible interleavings of sequence length 2 for the accelerator execution classes in this group. The unobserved interleavings are marked with a count of 0. The immediate observed pairs extracted from the execution are not shown in the Figure because they have all been included in the program-ordered and synchronized interleavings groups shown. Note that the observed pair resulting from executions 4 and 2 is also a possible interleaving of 2 and 5.

Abstract representations for different software executions can be combined to create a union of their scenarios by simply summing up the total number of occurrences for each unique scenario observed across the multiple executions. Through the process of combining, redundant accelerator interactions that are exhibited across multiple programs are abstracted into a compact set of interaction scenarios. This is one of the key features that allows AGARSoC to generate compact test programs that exhibit the interaction scenarios observed across several real programs.

8.4 Coverage Model Generation and Analysis

AGARSoC generates a coverage model that guides the RTL verification effort toward high-priority accelerator interactions as identified from software execution analysis. The

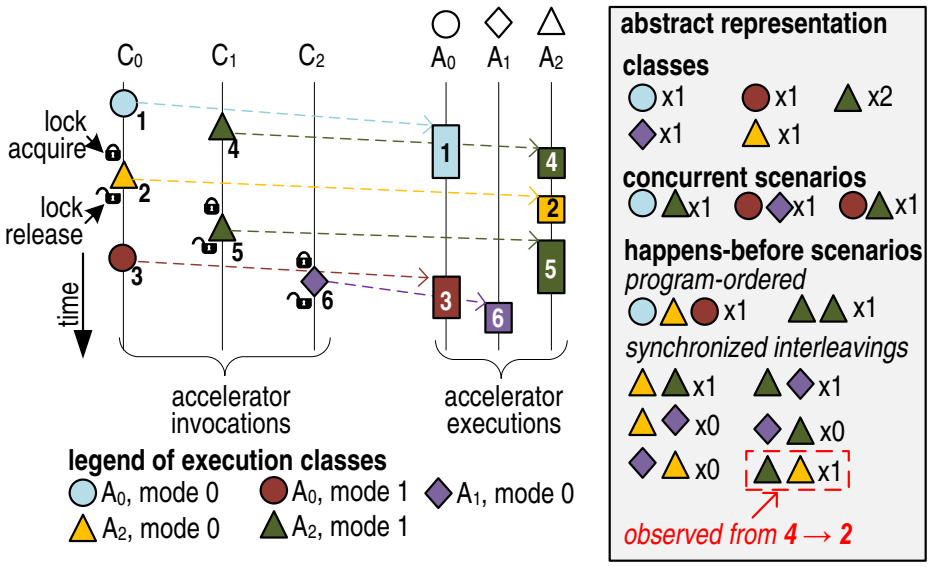


Figure 8.4: Abstracting an execution. Accelerator tasks are launched from the general-purpose cores. We group these task executions into unique execution classes, represented by shape and color. We represent concurrent execution classes and sequential execution classes. The ordering among classes is obtained from program-order, generated interleavings of synchronized invocations, and observed immediate ordered pairs.

coverage model generation algorithm performs a direct translation of the abstract representation extracted from the analysis: scenarios are translated into coverage events and scenario categories are translated into event groups. The coverage events in each event group can be sorted by the count associated with each scenario entry in the abstract representation. A higher count indicates a more likely scenario.

In addition to coverage model generation, AGARSoC's coverage engine is capable of evaluating and reporting the functional coverage data from RTL simulation. To take advantage of this capability, the RTL verification environment should output traces of memory operations. We believe this is a feature that most real-world RTL verification environments have, since memory traces are valuable during RTL debugging. AGARSoC's coverage engine is designed to be very flexible to adapt to changing verification requirements. Coverage models that AGARSoC generates can be merged without the need to analyze executions again. In addition, coverage output files are designed such that a coverage analysis output for a particular execution can be used as a coverage model for another execution. These features give engineers the flexibility to incorporate new software behaviors at any stage of verification, to compare the similarity between different test programs, and to incrementally refine verification goals.

Figure 8.5 shows a snippet of a coverage report generated by AGARSoC. This report shows the coverage analysis for an AGARSoC generated program versus a coverage model

Classes

Name	Count	Goal	Seen	% of goal
A0.1	7	29		24.1
A1.1	6	24		25.0
A2.1	6	25		24.0
A2.2	5	18		27.8
A1.2	4	4		100.0
A2.2	4	4		100.0
A0.1	3	4		75.0
A1.1	3	7		42.9

Concurrent

Name	Count	Goal	Seen	% of goal
A0.1 A1.1 A2.1	2	2		100.0
A0.1 A1.1 A2.2	2	6		33.3

Happens-Before

Name	Count	Goal	Seen	% of goal
A1.2 A2.2	2	2		100.0
A1.1 A1.2	1	3		33.3
A0.1 A0.1	1	4		25.0
A2.1 A0.1	0	0		0.0

Figure 8.5: Sample coverage analysis output

extracted from a software suite. The *Goal* column shows the counts of each coverage event as observed in the original software suite. AGARSoC currently uses these goals only as indicators of how frequently scenarios are observed, which can guide the choice of scenarios to include for test generation. The *Count* column shows how many times the events were observed in the generated test program. The *Seen* column indicates the presence of an event occurrence with a green color and its absence with red. The *% of goal* column indicates how many times the synthetic events were observed compared to the events in the original software suite. This column is ignored when all that is required is at least one hit of a coverage event. However, it becomes relevant when comparing the similarity between different test programs.

8.5 Test Generation

In addition to coverage models, AGARSoC also generates test programs that exhibit the accelerator interaction scenarios captured in its abstract representation. The test generation engine is highly flexible, allowing for the generation of test programs that target chosen scenarios. The number of threads in the generated test program, the length of ordered sequences to reproduce, and the range of scenarios to generate (from a given scenario group, based on frequency of occurrence) are some of the configurable parameters that control AGARSoC's creation of a multi-threaded schedule of accelerator executions to be executed by the generated test program.

8.5.1 Generated Test Program Structure

AGARSoC generates multi-threaded C programs that invoke accelerator executions through calls to library functions. Using barriers, the executions of these threads are divided into phases where the accelerator executions invoked in each phase can execute concurrently and accelerator executions in multiple phases are guaranteed to be ordered. For each execution, AGARSoC generates the definition of accelerator execution classes, a data layout, and queues of accelerator execution classes for each thread to execute. At the beginning of each phase, each thread in the phase-based execution mechanism pops an entry from its queue and invokes an accelerator, based on the generated data layout and execution class definitions.

8.5.2 Schedule Generation Algorithm

AGARSoC uses multiple heuristics to generate a compact schedule of accelerator executions that exhibits the desired accelerator scenarios. Our first step adapts the concurrent scenarios to an N-threaded environment, where N is specified by the engineer. If there are any concurrent scenarios with more than N concurrent executions, we enumerate all possible N-long sets of concurrent executions. The outcome of the first step is a list of unique scenarios with N or less concurrently executing accelerator execution classes. Our second step creates a list of ordered unique, execution class groups of size M or less. M is also specified by the engineer.

Our third step uses a heuristic that creates a multi-phase schedule of the concurrent scenarios that also satisfies the most number of sequential scenarios. Note that all invocations of a concurrent scenario are placed in a single phase, to be triggered by multiple threads. All accelerator executions invoked in a phase must complete before the threads proceed to the next phase. Therefore all accelerator executions in the earlier phase are ordered before all the accelerator executions in the subsequent phase, satisfying multiple sequential scenarios. Our heuristic creates partial schedules of length M until all concurrent scenarios are scheduled. At each step, it searches the length M schedule that satisfies the largest number of previously unsatisfied sequential scenarios. We perform this schedule search multiple times, starting from a different concurrent scenario each time. We finally pick the complete schedule that satisfies the most number of sequential scenarios.

Our last step constructs a single sequence of program-ordered executions from the remaining sequential scenarios. Using partial pattern matching, we ensure that the sequence we create has no repetitions and has the shortest length possible.

Table 8.1: Test software suites

Suite group	# of progs	Description
Soclib 1	9	Sequential accesses with locks
Soclib 2	9	Sequential accesses without locks
Soclib 3	9	Concurrent accesses with locks
Soclib 4	9	Concurrent accesses without locks
Soclib 5	18	Combinations of the four above
μBlaze 1	7	No synchronization
μBlaze 2	8	Lock, single-accelerator invocation
μBlaze 3	5	Lock, multiple-accelerator invocation
μBlaze 4	7	Barrier, redundant computations
μBlaze 5	13	Semaphore

8.6 Experimental Evaluation

We tested AGARSoC's capabilities on two different SoC platforms. We used Soclib [196] to create a Soclib ARM platform in SystemC that has 3 ARMv6k cores with caches disabled, 3 accelerators, 3 memory modules, and 1 peripheral, all connected via a bus. The three accelerators are a quadrature phase shift keying (QPSK) modulator, a QPSK demodulator, and a finite impulse response (FIR) filter. All accelerators communicate with software running on the cores using a common communication middleware. Multi-threaded synchronization is implemented using locks, which are acquired by code patterns that utilize ARM's atomic pair of load-locked and store-conditional instructions (LDREX and STREX). Using Xilinx Vivado® Design Suite [17], we created a MicroBlaze platform in RTL that comprises three MicroBlaze processors with caches and local memories, six accelerator IP blocks, one shared memory module, and peripheral devices connected via AXI interconnects. The six IP blocks are a FIR filter, a cascaded integrator-comb (CIC) filter, a CORDIC module, a convolutional encoder, a fast Fourier transfer (FFT) module, and a complex multiplier. Synchronization primitives (i.e., lock, barrier, semaphore) are implemented using a mutex IP.

For each platform, we created software suites that contain several patterns of accelerator execution, as summarized in Table 8.1. We designed our software suites to have a diverse set of accelerator usage patterns. After analyzing all of the Soclib software suites, AGARSoC identified 264 accelerator invocations, 130 observed concurrent scenarios, 315 observed happens-before scenarios. Similarly for MicroBlaze, it identified a total 358 accelerator invocations, 592 observed concurrent scenarios, and 502 observed happens-before scenarios. Figure 8.6 reports average compaction rates from AGARSoC's abstraction process of these observed scenarios. Better compaction rates are achieved for test suites that

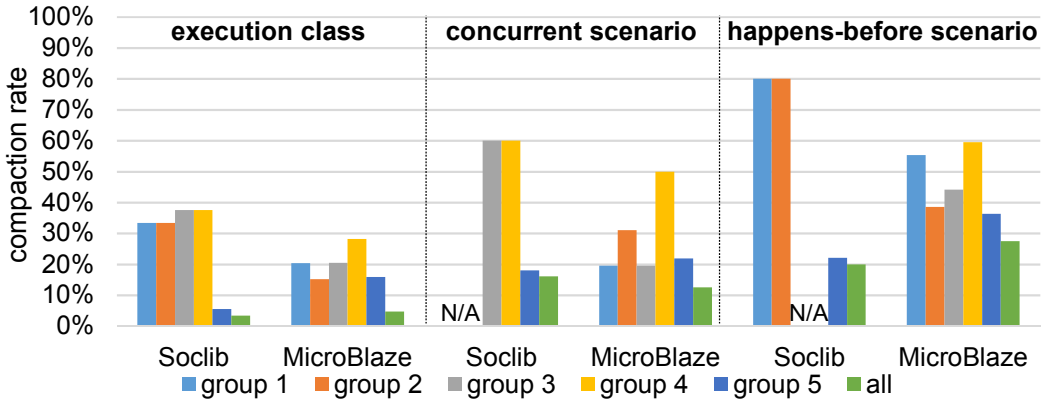


Figure 8.6: Compaction rates from abstract representation. The compaction of accelerator executions into unique instances of accelerator execution classes, concurrent scenarios, and happens-before scenarios for our two software suites. The compaction rate is measured as a ratio of the number of unique instances versus the total number of instances.

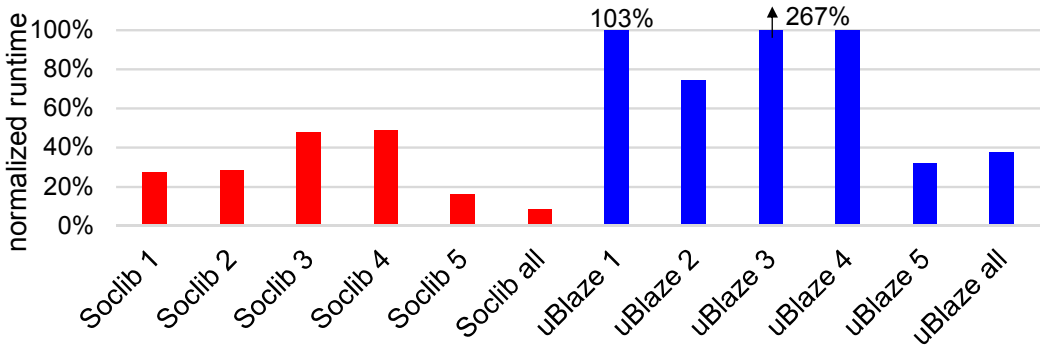


Figure 8.7: Normalized simulation runtime for each test group. While generated tests are usually shorter than the original test suites, MicroBlaze suites 1 and 3 show longer runtime, because of the inability of generating minimal schedules. However, the test program generated from all MicroBlaze suites does not exhibit this property.

exhibit redundant interactions. We report the compaction rates for each suite group and the three categories in the abstract representation.

Note that by automatically identifying important scenarios, AGARSoC focuses verification to a small, albeit important, portion of the large space of possible interactions. For instance, for the 17 unique classes identified from the MicroBlaze suites, there can potentially be 680 concurrent scenarios. AGARSoC identified only 74 cases. The potential number of unique classes is much larger than 17 and it is only due to AGARSoC's analysis that we were able to narrow it down to 17. If we conservatively assume that the SoC allows 100 unique classes for instance, 161,700 concurrent scenarios are possible.

We observed that the test programs generated from the compact abstract representation exercise a 100% of the scenarios captured in the abstract representations. Figure 8.7 sum-

marizes the reduction in simulation cycles achieved by running these tests instead of the original software suites. The AGARSoC generated tests simulate in only 8.6% of the time taken by the complete software suite for the Soclib platform and 38% for the MicroBlaze platform. However, the tests generated independently from the MicroBlaze software suites 1, 3, and 4 offer no execution time reduction, mainly due to the inability of the test generator to create minimal schedules to satisfy all scenarios. Suite 3 contains a program that is intentionally designed to exhibit accelerator patterns that push AGARSoC's schedule generator to the limit. Also, generated tests potentially exercise more happens-before scenarios than the original suites, due to all unobserved interleavings generated for synchronized accesses. Note that we are able to generate a more efficient schedule when the scenarios observed in all test suites are combined with other suites because we can get more compaction from scenarios that repeat across multiple programs. Compared to the MicroBlaze suites, the Soclib suites execute more routines that are not related to accelerator execution and hence are not reproduced in the generated tests.

AGARSoC detects synchronized accelerator executions and generates all possible interleavings as discussed in Section 8.3.3. During execution, only a handful of these possible interleavings are observed. We can instruct AGARSoC's test generator to generate tests to exercise the unobserved interleavings. We evaluated this capability by generating separate tests for the unobserved scenarios in the MicroBlaze suite groups 2 and 3. Our generated tests successfully exhibited all of the interleavings that were yet to be observed.

8.7 Discussions

Flexibility. AGARSoC provides a flexible set of tools that can support the demands of different design efforts. Capturing design-specific knowledge, as discussed in Section 8.3.1, is simple and easily portable to any verification effort. We were able to adapt AGARSoC to two quite different SoC platforms with minimal effort. In addition, AGARSoC can analyze and incorporate new software into the coverage model as it is developed, and generate tests that hit different coverage goals.

Enhancements. Even though we have not fully implemented them yet, we have several enhancements planned. Firstly, in addition to capturing concurrency and ordering among accelerator executions, we can also track their data sharing patterns to identify producer-consumer or data race scenarios. These scenarios can be easily incorporated in the abstract representation as extra scenario groups and can be specified as constraints for AGARSoC's test generation algorithm. These data sharing scenarios, together with a straight-forward

enhancement to detect cases where accelerators can invoke other accelerators, can enable AGARSoC to support several different accelerator-rich SoC design approaches. For instance, we can support scenarios where a task is executed by a pipeline of multiple accelerators, without the involvement of general-purpose cores in between. Secondly, we can randomize some parameters of the high-level simulation model to model non-deterministic execution of software. This allows us to generate multiple, legal memory traces per program increasing the richness of our software suite and further refining the accuracy of our analysis. Thirdly, we can implement mechanisms to detect other multi-threaded, synchronized accelerator executions, in addition to the lock-based and mutex-IP-based mechanism we currently support. Finally, AGARSoC's test generator can be enhanced, with minimal effort, to fully utilize the parallelism available in the SoC-under-verification and also in the verification environment. The schedules we generate to target happens-before scenarios can be multi-threaded. In addition, we can launch multiple concurrent accelerator executions from a single thread. We can also generate multiple, execution-time balanced tests that hit distinct coverage goals to be simulated in parallel.

Accuracy. AGARSoC relies on several heuristics to abstract executions into scenarios. These heuristics rely on timing information from a high-level simulation model, which does not necessarily reflect the timing that will be observed during RTL simulation. However, note that we utilize the timing information from the high-level model only to infer accelerator executions that *are likely to run concurrently or are likely to be ordered*. As such, we believe that the accuracy of our heuristics are sufficient enough to guide verification engineers to the high-priority scenarios. In addition, AGARSoC uses scheduling heuristics when generating test programs that may result in sub-optimal simulation cycles. Even though the resulting schedule may result in large test programs for unfavorable cases, it is mostly simulated significantly faster than the original software suites.

Bug detection. We only focus on identifying high-priority scenarios, defining coverage, and generating tests that hit these coverage goals. We do not investigate the ability to check for and find bugs.

8.8 Related Work

Using high-level models for hardware verification. Hardware and software co-design and co-verification has been investigated extensively as a means to shorten time-to-market in designing SoCs [52, 75, 88]. Here, virtual prototyping platforms allow software en-

gineers to develop and verify their software while hardware is still in development. As discussed in Section 2.4, the verification of individual accelerators can benefit from high-level models of SoC designs; these models are usually provided as part of the virtual platform. AGARSoC extends beyond the current usage of the virtual platforms to enable early analysis of software with the purpose of guiding hardware verification.

Analyzing execution traces. Simpoints [95] have been widely used to find representative sections of single-threaded software for use in architectural studies. Unlike AGARSoC, Simpoints are neither designed for analyzing concurrent software in accelerator-rich SoCs nor for guiding the verification process. Ganesan et al. [89] propose a mechanism for generating proxy programs to represent multi-threaded benchmarks. Unlike AGARSoC, their solution does not address accelerator interaction scenarios.

Coverage model and test generation. In contrast to previous work on automatic coverage-directed test generation [109], AGARSoC presents a solution to automatically extract coverage models by analyzing software. While a few solutions have been proposed for generating coverage models from dynamic simulation traces [138], they focus on low-level interactions obtained from RTL simulations. AGARSoC's coverage models focus on high-level accelerator interactions and are ready much earlier in the verification process.

8.9 Summary

We presented AGARSoC, a verification methodology and a set of tools that can guide the verification of accelerator interactions in accelerator-rich SoCs. AGARSoC is based on the decomposition of accelerator interactions into basic interaction elements, which can be reassembled to create new synthetic tests that target interesting interaction scenarios. To do so, AGARSoC analyzes the executions of software test suites using a high-level model of the SoC, which we use to identify high-priority accelerator interactions. Using this analysis, our tools generate coverage models for these high-priority interactions to be used during RTL verification. In addition, we can also generate compact test programs that can hit these coverage goals. We built and successfully demonstrated the capabilities of our tools for two different SoC platforms.

CHAPTER 9

Conclusion and Future Directions

Systems on chip (SoCs) integrate heterogeneous system components into a single chip. They are widely deployed in various computing devices from tiny internet-of-thing devices to data centers. Although the SoC design paradigm has been very successful in improving system performance and energy efficiency, two challenges may limit viability of the SoC design paradigm in the coming years: functional verification and reliability challenges.

The functional correctness of SoCs is challenged by their growing complexity. Design verification, ideally, must check every functional behavior that SoCs can exhibit, thus, it may entail massive software computation for simulating and checking a vast number of features and corner-cases. Unfortunately, some rarely-occurring buggy behaviors go unnoticed during verification and validation, and slip through SoC products. To avoid such disastrous situations, it is important to not only detect design bugs at design time but also promptly patch escaped bugs at runtime, even if bug-patching often requires non-trivial hardware resources.

At the same time, the reliability of SoCs is challenged by diminishing transistor sizes and increasing transistor densities. Nano-scale transistors are more likely to fail within their lifespan, possibly making the entire chip dysfunctional if transistor failures occur in critical system components. Unfortunately, it is extremely difficult to accurately predict transistor failures. There are a myriad of transistors integrated in the chip, so that predicting the degradation of each transistor is almost impossible within the short period of time allowed for design and verification. As a result, SoCs are more and more likely to experience design bugs and permanent faults after deployment, negatively affecting their viability and market growth. For the SoC design paradigm to flourish in the future, evasive errors must be addressed with solutions that are pervasive, efficient, and practical.

This dissertation investigates a *decompose-and-conquer* approach to reduce the complexity of software computation and hardware resources required to address these evasive

errors. We apply this approach to each major SoC component: microprocessor cores, memory subsystems, on-chip networks, and accelerators. By applying our approach, we observe that overall complexity is greatly reduced. For instance, our solutions for bug-masking analysis and application-aware routing reduce the overall computation by up to three orders of magnitude. However, reduced complexity comes with losses in error coverage. For instance, our bug-masking analysis is unable to predict 23% bug-masking incidents, and our bug patching cannot fix up to 39% of the memory bugs among those considered in our study, as we summarize in more detail in the following section.

In the rest of this chapter, we first summarize the contributions of the solutions proposed in this dissertation, and then discuss future research directions in these domains.

9.1 Dissertation Summary

This dissertation presents decomposition-based methods for each major type of SoC component, as discussed below, striving to improve the efficiencies of detecting and recovering from design bugs and permanent faults.

Microprocessor cores. We investigate bug-masking issues in post-silicon microprocessor functional verification. Bug-masking incidents may happen due to limited capabilities to trace and analyze detailed results generated from test execution in post-silicon validation platforms. In this context, we propose BugMAPI, which strives to quickly evaluate the likelihood of bug-masking incidents in instruction tests. It decomposes bug-masking analysis from the program level to the instruction level, then reassembles instruction-level results to obtain the program-level bug-masking probability. Thanks to this decomposition method, BugMAPI reduces the software computation involved in bug-masking analysis by three orders of magnitude. This reduction, however, comes with a loss of accuracy in the analysis results; indeed, BugMAPI fails to predict 23% bug-masking incidents on average. We also demonstrate an application of the proposed bug-masking analysis, which instruments test-cases to reduce bug-masking incidents.

Memory subsystems. We study subtle design bugs in memory subsystems. Among these bugs, we first focus on memory consistency bugs, which manifest themselves as incorrect memory-access interleavings that do not abide by the memory consistency specification for the system. Indeed, SoCs may exhibit numerous memory-access interleaving patterns, each of which must be verified if they conform to the memory consistency specification. This verification process involves heavy result-checking computation, which reasons about

the order among memory operations in multithreaded tests, a major bottleneck in the entire memory consistency verification process. MTraceCheck strives to minimize this result-checking computation. It leverages the similarity among test executions to develop a novel incremental-checking scheme on memory ordering patterns. As a result, we achieve a 5× speedup in topological sorting computation. Due to this decomposition, however, the proposed result-checking scheme may not be able to detect very subtle memory ordering violations whose detection requires inferences across multiple decomposed groups.

Furthermore, we investigate hardware patching solutions to recover from general memory subsystem bugs: not only memory consistency bugs, but also deadlock, livelock, and data-corruption bugs. To accurately identify and recover from these bugs, prior solutions deploy global hardware structures, which in turn entail significant overheads. This dissertation investigates an alternative solution that significantly reduces hardware overheads, with acceptable degradation in bug-patching capability. For this purpose, we propose μMemPatch, which is a distributed patch solution leveraging a small monitoring logic attached to each core node. This monitoring logic observes speculative executions in the core's back-end pipeline stages, as well as data races at the core-cache interface. When a bug-prone state is detected at runtime, μMemPatch triggers fine-grained microarchitectural recovery actions entailing low performance overheads. μMemPatch's area overhead is acceptable, at 5.7% on average; however, due to its limited monitoring capability, it does not cover 39% of the bugs we studied.

On-chip networks. We propose two routing reconfiguration solutions to address reliability challenges in on-chip networks. Routing reconfiguration is a cost-effective fault-tolerant solution, compared to other solutions (e.g., spare resources), to bypass faults on 2D mesh networks, because routing reconfiguration leverages path diversity inherently existing on mesh network topologies. Ideally, routing reconfiguration should incur no network downtime, while network performance (e.g., throughput, latency) should not degrade due to faults. However, prior available solutions entail high reconfiguration impact and performance degradation after reconfiguration.

BLINC aims to minimize reconfiguration impacts, especially, reconfiguration latency. To achieve this goal, it localizes routing reconfiguration by decomposing the whole network into segments. It then computes a set of routing metadata that can quickly calculate valid routes. This routing metadata is embedded in each router, and updated by simple Boolean operations during routing reconfiguration. Our experimental evaluation shows a significant reduction in reconfiguration latency: three orders of magnitude for 8×8 mesh networks. However, BLINC's online reconfiguration is unable to handle subtle fault patterns, i.e.,

multiple faults impacting the same segment.

We then strive to address performance degradation, using application-aware routing. In deploying application-aware routing, we observe that significant software computation is required to find optimal routes for the application's traffic patterns. To reduce such computation, we proposed FATE, which leverages localized routing rules that expedite the exploration of new routes that avoid the fault. The proposed routing rules prune seemingly unfruitful routes in advance, so to quickly converge to a quasi-optimal set of routes. FATE significantly reduces the routing computation by more than two orders of magnitude, compared to a state-of-the-art application-aware routing for on-chip networks. However, this solution provides no guarantee on the optimality of computed routes for the given network topology and the application's traffic patterns.

Accelerators. This dissertation focuses on the verification of accelerator interactions, a critical activity during SoC integration. SoCs consist of tens or hundreds of heterogeneous accelerators, and are expected to include many more in the future. Thus, the number of accelerator interaction scenarios is increasing, while verification strategies evolve slowly. Usually, massive simulations and testings are conducted to trigger buggy accelerator interactions that rarely occur.

AGARSoC presents a novel verification methodology to expedite coverage closure in the verification of accelerator interactions. The key technical contribution we offer is a decomposition of accelerator interactions into basic interaction elements. Specifically, AGARSoC obtains accelerator interaction patterns from high-level simulations, extracts basic interaction elements from them, reassembles the basic elements to create synthetic test-cases, and runs the synthetic test-cases on the design under verification. Our evaluation shows that synthetic test-cases can hit the same coverage goals in a fraction of the time spent running original test suites. However, AGARSoC focuses only on simple interaction patterns, both concurrent and sequential, which may limit its bug-finding capabilities.

9.2 Future Directions

We recognize several research directions that may extend the contributions of the research proposed in this dissertation.

Rethinking trade-offs. We have shown that the proposed decomposition methods sacrifice coverage, accuracy, or performance to some degree. To be specific, BugMAPI was not able to identify some bug-masking incidents that are predicted by dynamic bug-injection

analysis. MTraceCheck may be unable to detect intricate memory ordering violations that can be caught by sophisticated inferences of memory orderings across decomposed groups. μMemPatch is unable to accurately detect bugs related to rare microarchitectural conditions across multiple memory subsystem components. BLINC can handle at most one fault per segment at runtime; more faults must be recovered by a conventional global reconfiguration (i.e., network re-segmentation). FATE may not produce the optimal set of routes. AGARSoC may be inefficient in capturing sophisticated interaction scenarios.

In this dissertation research, we have not evaluated the implications of these limitations in terms of coverage, accuracy, and performance, thus it would be viable to assess the practicality of the solutions proposed with respect to those metrics.

Heterogeneous decompositions. Our decomposition-based approach splits the complex, unit- or system-level problem into a smaller, homogeneous problems. For instance, BugMAPI decomposes the whole program into individual instructions. However, such homogeneous decompositions may lead to missing some factors of the solution. In BugMAPI, some sections of the program may be more prone to bug masking than others, thus we could apply a more accurate analysis to critical sections, while we could leverage a faster, but less accurate approach, for the others, by devising decomposition at adjustable granularities. Similarly, we could explore a range of distinct decompositions for the other solutions as well, to best address the quality vs. cost trade-offs.

Using formal methods for verifying accelerator interactions. Formal methods could be used to systematically explore accelerator interactions. In particular, formal methods could replace the current scheduling algorithm used in AGARSoC, which may produce a sub-optimal test schedule, and produce an optimal test schedule for all scenarios. To this end, accelerator interactions can be captured as constraints specified in a formal specification language.

Realistic evaluations for functional verification solutions. Throughout our research endeavor on functional verification, we have performed several bug-injection experiments on simulators. Bug-injection experiments, however, may not faithfully model the effect of real-world bugs. In bug injection, bugs are often modeled after gate- or transistor-level faults, which might be vastly different from real-world bugs. However, we believe that future research on functional verification can become more impactful if the evaluation used realistic bug datasets.

Moreover, future research can take advantage of closed-loop functional verification en-

vironments. Many measurements are conducted in this dissertation in open-loop environments, measuring only the specific portions we are interested in. For instance, we measured the time spent on topological sorting in MTraceCheck; however, there are some important portions of time that we did not consider, e.g., test generation, test instrumentation, graph reconstruction. These portions may be significant in real-world verification scenarios.

In developing AGARSoC, we strived to collect realistic test suites for SoCs. Unfortunately, the SoC designs we used in our evaluation are much smaller than cutting-edge ones, and accordingly, the test suites for these SoC designs are fairly simple. Future research could evaluate real test suites for cutting-edge SoCs.

Automated diagnoses of memory subsystem bugs. μMemPatch assumes manual diagnoses to pinpoint the root causes of bugs, which may be very time-consuming for bugs that are related to very complex sequences of microarchitectural events. Automated diagnoses can reduce human efforts and hence be much faster. A possible approach in this direction would 1) model both the microarchitecture of a given memory subsystem and a given bug symptom in a formal specification language, then 2) combine the two specifications to automatically explore microarchitectural events in the memory subsystem that lead to the bug. This formal approach can be modeled as a planning problem, which has been well studied in the artificial intelligence field.

Leveraging hardware-assisted debugging features in SoCs. Modern SoCs deploy several debugging features available in silicon, e.g., Intel's Processor Trace (PT) and Branch Trace Store (BTS), ARM's Embedded Trace Macrocell (ETM). These features can collect dynamic information from program execution, e.g., control flow and memory-access results. Intel PT can also compress collected data, which can greatly reduce overheads involved in tracing. Thus, post-silicon and runtime functional verification solutions can take advantage of these debugging features to alleviate observability issues.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Apple iPhone Xs Max teardown. http://www.techinsights.com/about-techinsights/overview/blog/apple-iphone-xs-teardown/.

[2] ARM's Cortex M: Even smaller and lower power CPU cores. https://www.anandtech.com/show/8400/arms-cortex-m-even-smaller-and-lower-power-cpu-cores.

[3] Bathtub curve. https://en.wikipedia.org/wiki/Bathtub_curve.

[4] Berkeley out-of-order machine. https://github.com/ucb-bar/riscv-boom.

[5] Big trouble at 3nm. https://semiengineering.com/big-trouble-at-3nm/.

[6] The Coq proof assistant. https://coq.inria.fr.

[7] Das U-Boot – the universal boot loader. http://www.denx.de/wiki/U-Boot.

[8] Deadlock with RMW operations. https://groups.google.com/forum/#!topic/gem5-gpu-dev/RQv4SxIKv7g.

[9] gem5 mercurial repository host. http://repo.gem5.org.

[10] GNU coreutils version 8.25, January. http://ftp.gnu.org/gnu/coreutils/.

[11] Intel redesigns flawed atom cpus to stave off premature chip death. https://www.theregister.co.uk/2017/04/27/intel_redesigns_atom_line/.

[12] k-medoids algorithm. https://en.wikipedia.org/wiki/K-medoids.

[13] Qualcomm unveils Snapdragon 850, further empowering the always-connected Windows 10 PC. https://www.forbes.com/sites/davealtavilla/2018/06/05/qualcomm-unveils-snapdragon-850-further-empowering-the-always-connected-windows-10-pc.

[14] Rocket chip generator. https://github.com/freechipsproject/rocket-chip.

[15] Semi-critical Intel Atom C2000 SoC flaw discovered, hardware fix required. http://www.anandtech.com/show/11110/semi-critical-intel-atom-c2000-flaw-discovered.

[16] TILE64 processor overview. http://www.tilera.com/products/processors/TILE64.

[17] Xilinx Vivado design suite. http://www.xilinx.com/products/design-tools/vivado.html.

[18] R. Abdel-Khalek, R. Parikh, A. DeOrio, and V. Bertacco. Functional correctness for CMP interconnects. In *Proc. ICCD*, 2011.

[19] 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 *Proc. DAC*, 2011.

[20] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12), 1996.

[21] S. V. Adve and M. D. Hill. Weak ordering–a new definition. In *Proc. ISCA*, 1990.

[22] K. Aisopos, A. DeOrio, L.-S. Peh, and V. Bertacco. Ariadne: Agnostic reconfiguration in a disconnected network environment. In *Proc. PACT*, 2011.

[23] J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2), 2012.

[24] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.

[25] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *Proc. TACAS*, 2011.

[26] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2), 2014.

[27] AMD. *Revision Guide for AMD Athlon64 and AMD Opteron Processors*, 2009.

[28] AMD. *Revision Guide for AMD Family 11h Processors*, 2011.

[29] AMD. *Revision Guide for AMD Family 10h Processors*, 2012.

[30] AMD. *Revision Guide for AMD Family 12h Processors*, 2012.

[31] AMD. *Revision Guide for AMD Family 14h Models 00h-0Fh Processors*, 2013.

[32] AMD. *Revision Guide for AMD Family 16h Models 00h-0Fh Processors*, 2013.

[33] AMD. *Revision Guide for AMD Family 15h Models 00h-0Fh Processors*, 2014.

[34] ARM. *Barrier Litmus Tests and Cookbook*, 2009.

[35] ARM. *Cortex-A9 MPCore Programmer Advice Notice Read-after-Read Hazards, ARM Reference 761319*, 2011.

[36] ARM. *ARM Architecture Reference Manual – ARMv7-A and ARMv7-R edition*, 2012.

[37] ARM. *Cortex-A5 MPCore Software Developers Errata Notice*, 2013.

[38] ARM. *Cortex-A17 MPCore Software Developers Errata Notice*, 2015.

[39] ARM. *Cortex-A9 Processors Software Developers Errata Notice*, 2015.

[40] ARM. *Cortex-A15 MPCore - NEON Software Developers Errata Notice*, 2016.

[41] ARM. *Cortex-A7 MPCore Software Developers Errata Notice*, 2016.

[42] ARM. *Cortex-A53 MPCore Software Developers Errata Notice*, 2017.

[43] ARM. *Cortex-A55 Processor MP070 Software Developer Errata Notice*, 2017.

[44] ARM. *Cortex-A57 MPCore Software Developer Errata Notice*, 2017.

[45] ARM. *Cortex-A72 MPCore Software Developer Errata Notice*, 2017.

[46] ARM. *Cortex-A73 MPCore Software Developer Errata Notice*, 2017.

[47] ARM. *Cortex-A75 (MP058) Software Developer Errata Notice*, 2017.

[48] Arvind and J.-W. Maessen. Memory model = instruction reordering + store atomicity. In *Proc. ISCA*, 2006.

[49] G. Ascia, V. Catania, M. Palesi, and D. Patti. Implementation and analysis of a new selection strategy for adaptive routing in networks-on-chip. *IEEE Trans. Computers*, 57(6), 2008.

[50] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *Proc. MICRO*, 1999.

[51] M. Badr and N. Jerger. SynFull: Synthetic traffic models capturing cache coherent behavior. In *Proc. ISCA*, 2014.

[52] B. Bailey and G. Martin. *ESL Models and Their Application: Electronic System Level Design and Verification in Practice*. Springer Publishing Company, Incorporated, 2009.

[53] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. MICRO*, 1996.

[54] N. Barrow-Williams, C. Fensch, and S. Moore. A communication characterisation of Splash-2 and Parsec. In *Proc. IISWC*, 2009.

[55] C. Bienia, S. Kumar, J. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. PACT*, 2008.

[56] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[57] H. W. Cain, M. H. Lipasti, and R. Nair. Constraint graph analysis of multithreaded programs. In *Proc. PACT*, 2003.

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

[59] J. Cano, J. Flich, A. Roca, J. Duato, M. Coppola, and R. Locatelli. Efficient routing in heterogeneous SoC designs with small implementation overhead. *IEEE Trans. Computers*, 63(2), 2014.

[60] R. Casado, A. Bermudez, F. Quiles, J. Sanchez, and J. Duato. Performance evaluation of dynamic reconfiguration in high-speed local area networks. In *Proc. HPCA*, 2000.

[61] Y. C. Chang, C. T. Chiu, S. Y. Lin, and C. K. Liu. On the design and analysis of fault tolerant noc architecture using spare routers. In *Proc. ASPDAC*, 2011.

[62] D. Chatterjee, A. Koyfman, R. Morad, A. Ziv, and V. Bertacco. Checking architectural outputs instruction-by-instruction on acceleration platforms. In *Proc. DAC*, 2012.

[63] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *Proc. HPCA*, 2008.

[64] Y. Chen, L. Li, T. Chen, L. Li, L. Wang, X. Feng, and W. Hu. Program regularization in memory consistency verification. *IEEE Trans. Parallel and Distributed Systems*, 23(11), 2012.

[65] Y. Chen, Y. Lv, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan. Fast complete memory consistency verification. In *Proc. HPCA*, 2009.

[66] Y.-K. Chen and S. Y. Kung. Trend and challenge on system-on-a-chip designs. *Journal of Signal Processing Systems*, 53(1), Nov 2008.

[67] 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 *Proc. DAC*, 2016.

[68] J. H. Collet, P. Zajac, M. Psarakis, and D. Gizopoulos. Chip self-organization and fault tolerance in massively defective multicore arrays. *IEEE Trans. Dependable and Secure Computing*, 8(2), 2011.

[69] J. Cong, C. Liu, and G. Reinman. ACES: Application-specific cycle elimination and splitting for deadlock-free routing on irregular network-on-chip. In *Proc. DAC*, 2010.

[70] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects mechanisms, architectural support, and evaluation. In *Proc. MICRO*, 2007.

[71] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[72] W. Dally and C. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Computers*, C-36(5), 1987.

[73] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

[74] J. Darringer, E. Davidson, D. J. Hathaway, B. Koenemann, M. Lavin, J. K. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G. Tellez, and L. Trevillyan. EDA in IBM: past, present, and future. *IEEE Trans. CAD*, 19(12), 2000.

[75] G. De Micheli, R. Ernst, and W. Wolf, editors. *Readings in Hardware/Software Co-design*. Kluwer Academic Publishers, 2002.

[76] F. M. De Paula, M. Gort, A. J. Hu, S. J. E. Wilton, and J. Yang. Backspace: Formal analysis for post-silicon debug. In *Proc. FMCAD*, 2008.

[77] D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.

[78] A. DeOrio, D. Fick, V. Bertacco, D. Sylvester, D. Blaauw, J. Hu, and G. Chen. A reliable routing architecture and algorithm for NoCs. *IEEE Trans. CAD*, 31(5), 2012.

[79] D. L. Dill. The Murphi verification system. In *Proc. CAV*, 1996.

[80] M. Ebrahimi, M. Daneshtalab, J. Plosila, and F. Mehdipour. MD: Minimal path-based fault-tolerant routing in on-chip networks. In *Proc. ASPDAC*, 2013.

[81] M. Elver and V. Nagarajan. McVerSi: A test generation framework for fast memory consistency verification in simulation. In *Proc. HPCA*, 2016.

[82] F. Fallah, S. Devadas, and K. Keutzer. OCCOM—efficient computation of observability-based code coverage metrics for functional verification. *IEEE Trans. CAD*, 20(8), 2001.

[83] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: Probabilistic soft error reliability on the cheap. In *Proc. ASPLOS*, 2010.

[84] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester. Vicis: A reliable network for unreliable silicon. In *Proc. DAC*, 2009.

[85] J. Flich, T. Skeie, R. Juarez-Ramirez, O. Lysne, P. Lopez, A. Robles, J. Duato, M. Koibuchi, T. Rokicki, and J. Sancho. A survey and evaluation of topology-agnostic deterministic routing algorithms. *IEEE Trans. Parallel and Distributed Systems*, 23(3), 2012.

[86] H. Foster. Trends in functional verification: A 2014 industry study. In *Proc. DAC*, 2015.

[87] N. Foutris, D. Gizopoulos, M. Psarakis, X. Vera, and A. Gonzalez. Accelerating microprocessor silicon validation by exposing ISA diversity. In *Proc. MICRO*, 2011.

[88] M. Fujita, I. Ghosh, and M. Prasad. *Verification Techniques for System-Level Design*. Morgan Kaufmann Publishers Inc., 2008.

[89] K. Ganesan and L. K. John. Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. *IEEE Trans. Computers*, 63(4), 2014.

[90] A. Ghofrani, R. Parikh, S. Shamshiri, A. DeOrio, K. T. Cheng, and V. Bertacco. Comprehensive online defect diagnosis in on-chip networks. In *Proc. VTS*, 2012.

[91] C. Glass and L. Ni. The turn model for adaptive routing. In *Proc. ISCA*, 1992.

[92] P. Gratz, B. Grot, and S. Keckler. Regional congestion awareness for load balance in networks-on-chip. In *Proc. HPCA*, 2008.

[93] C. Grecu, A. Ivanov, R. Saleh, and P. Pande. Testing network-on-chip communication fabrics. *IEEE Trans. CAD*, 26(12), 2007.

[94] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. The stagenet fabric for constructing resilient multicore systems. In *Proc. MICRO*, 2008.

[95] G. Hamerly, E. Perelman, J. Lau, and B. Calder. SimPoint 3.0: Faster and more flexible program analysis. *Journal of Instruction Level Parallel*, 2005.

[96] S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan. TSOtool: A program for verifying memory systems using the memory consistency model. In *Proc. ISCA*, 2004.

[97] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz mesh interconnect for a teraflops processor. *IEEE Micro*, 27(5), 2007.

[98] IBM. *Power ISA Version 2.07B*, 2015.

[99] IBM. *POWER9 Processor Users Manual*, 2018.

[100] Intel. *Intel Core2 Extreme Quad-Core Processor QX6000 Sequence Specification Update*, 2008.

[101] Intel. *Intel Core2 Extreme Processor QX9000 Series Specification Update*, 2011.

[102] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2015.

[103] Intel. *2nd Generation Intel Core Processor Family Desktop Specification Update*, 2016.

[104] Intel. *6th Generation Intel Processor Family Specification Update*, September 2016.

[105] Intel. *Desktop 3rd Generation Intel Core Processor Family Specification Update*, 2016.

[106] Intel. *5th Generation Intel Core Processor Family Specification Update*, 2017.

[107] Intel. *Desktop 4th Generation Intel Core Processor Family Specification Update*, 2017.

[108] Intel. *Intel Atom Processor C2000 Product Family Specification Update*, January 2017.

[109] C. Ioannides and K. I. Eder. Coverage-directed test generation automated by machine learning – a review. 17(1), 2012.

[110] C. Iordanou, V. Soteriou, and K. Aisopos. Hermes: Architecting a top-performing fault-tolerant routing algorithm for networks-on-chips. In *Proc. ICCD*, 2014.

[111] H. Jeon and M. Annavaram. Warped-DMR: Light-weight error detection for GPGPU. In *Proc. MICRO*, 2012.

[112] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, J. Kim, and W. J. Dally. A detailed and flexible cycle-accurate network-on-chip simulator. In *Proc. ISPASS*, 2013.

[113] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proc. ISCA*, 2017.

[114] J. Keane, S. Venkatraman, P. Butzen, and C. H. Kim. An array-based test circuit for fully automated gate dielectric breakdown characterization. In *Proc. CICC*, 2008.

[115] J. Keshava, N. Hakim, and C. Prudvi. Post-silicon validation challenges: How EDA and academia can help. In *Proc. DAC*, 2010.

[116] M. Kinsy, M. H. Cho, K. S. Shim, M. Lis, G. Suh, and S. Devadas. Optimal and heuristic application-aware oblivious routing. In *IEEE Trans. Computers*, 2013.

[117] M. Koibuchi, A. Jouraku, and H. Amano. The impact of path selection algorithm of adaptive routing for implementing deterministic routing. In *Proc. PDPTA*, 2002.

[118] R. Komuravelli, S. V. Adve, and C.-T. Chou. Revisiting the complexity of hardware cache coherence and some implications. *ACM Transactions on Architecture Code Optimization*, 11(4), 2014.

[119] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9), 1979.

[120] D. Lee and V. Bertacco. MTraceCheck: Validating non-deterministic behavior of memory consistency models in post-silicon validation. In *Proc. ISCA*, 2017.

[121] D. Lee, T. Kolan, A. Morgenshtein, V. Sokhin, R. Morad, A. Ziv, and V. Bertacco. Probabilistic bug-masking analysis for post-silicon tests in microprocessor verification. In *Proc. DAC*, 2016.

[122] D. Lee, O. Matthews, and V. Bertacco. Low-overhead microarchitectural patching for multicore memory subsystems. In *Proc. ICCD*, 2018.

[123] D. Lee, R. Parikh, and V. Bertacco. Brisk and limited-impact NoC routing reconfiguration. In *Proc. DATE*, 2014.

[124] D. Lee, R. Parikh, and V. Bertacco. Highly fault-tolerant NoC routing with application-aware congestion management. In *Proc. NOCS*, 2015.

[125] T. Lehtonen, D. Wolpert, P. Liljeberg, J. Plosila, and P. Ampadu. Self-adaptive system for addressing permanent errors in on-chip interconnects. In *IEEE Trans. VLSI Systems*, 2010.

[126] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *Proc. SC*, 2017.

[127] M. Li, Q.-A. Zeng, and W.-B. Jone. DyXY: A proximity congestion-aware deadlock-free dynamic routing method for network on chip. In *Proc. DAC*, 2006.

[128] S. Li, V. Sridharan, S. Gurumurthi, and S. Yalamanchili. Software-based dynamic reliability management for GPU applications. In *Proc. IRPS*, 2016.

[129] 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*, 33(10), 2014.

[130] Q. Liu, C. Jung, D. Lee, and D. Tiwari. Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection. In *Proc. MICRO*, 2016.

[131] D. Lustig, M. Pellauer, and M. Martonosi. PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *Proc. MICRO*, 2014.

[132] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi. ArMOR: Defending against memory consistency model mismatches in heterogeneous architectures. In *Proc. ISCA*, 2015.

[133] D. Lustig, A. Wright, A. Papakonstantinou, and O. Giroux. Automated synthesis of comprehensive memory model litmus test suites. In *Proc. ASPLOS*, 2017.

[134] S. Mador-Haim, R. Alur, and M. M. Martin. Generating litmus tests for contrasting memory consistency models. In *Proc. CAV*, 2010.

[135] B. W. Mammo. *Reining in the Functional Verification of Complex Processor Designs with Automation, Prioritization, and Approximation*. PhD thesis, University of Michigan, 2017.

[136] B. W. Mammo, V. Bertacco, A. DeOrio, and I. Wagner. Post-silicon validation of multiprocessor memory consistency. *IEEE Trans. CAD*, 34(6), 2015.

[137] B. W. Mammo, D. Lee, H. Davis, Y. Hou, and V. Bertacco. AGARSoC: Automated test and coverage-model generation for verification of accelerator-rich SoCs. In *Proc. ASPDAC*, 2017.

[138] E. E. Mandouh and A. G. Wassal. CovGen: A framework for automatic extraction of functional coverage models. In *Proc. ISQED*, 2016.

[139] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi. CCICheck: Using μhb graphs to verify the coherence-consistency interface. In *Proc. MICRO*, 2015.

[140] L. Maranget, S. Sarkar, and P. Sewell. A tutorial introduction to the ARM and POWER relaxed memory models, 2012.

[141] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proc. MICRO*, 2007.

[142] A. Meixner and D. J. Sorin. Detouring: Translating software to circumvent hard faults in simple cores. In *Proc. DSN*, 2008.

[143] A. Meixner and D. J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *IEEE Trans. Dependable and Secure Computing*, 6(1), 2009.

[144] A. Mejia, J. Flich, and J. Duato. On the potentials of segment-based routing for NoCs. In *Proc. ICPP*, 2008.

[145] A. Mejia, J. Flich, J. Duato, S.-A. Reinemo, and T. Skeie. Segment-based routing: An efficient fault-tolerant routing algorithm for meshes and tori. In *Proc. IPDPS*, 2006.

[146] P. Mishra and N. Dutt. Graph-based functional test program generation for pipelined processors. In *Proc. DATE*, 2004.

[147] S. Murali, L. Benini, T. Theocharides, N. Vijaykrishnan, M. J. Irwin, and G. De Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design & Test*, 22(5), 2005.

[148] A. Nahir, M. Dusanapudi, S. Kapoor, K. Reick, W. Roesner, K.-D. Schubert, K. Sharp, and G. Wetli. Post-silicon validation of the IBM POWER8 processor. In *Proc. DAC*, 2014.

[149] P. J. Nair, D.-H. Kim, and M. K. Qureshi. ArchShield: Architectural framework for assisting dram scaling by tolerating high error rates. In *Proc. ISCA*, 2013.

[150] S. Narayanasamy, B. Carneal, and B. Calder. Patching processor design errors. In *Proc. ICCD*, 2006.

[151] A. Nassar, F. Kurdahi, and W. Elsharkasy. NUVA: Architectural support for runtime verification of parametric specifications over multicores. 2015.

[152] R. Nathan and D. J. Sorin. Nostradamus: Low-cost hardware-only error detection for processor cores. In *Proc. DATE*, 2014.

[153] M. Naylor, S. W. Moore, and A. Mujumdar. A consistency checker for memory subsystem traces. In *Proc. FMCAD*, 2016.

[154] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. NDSS*, 2005.

[155] M. Palesi *et al*. Design of bandwidth aware and congestion avoiding efficient routing algorithms for networks-on-chip platforms. In *Proc. NOCS*, 2008.

[156] R. Parikh and V. Bertacco. Formally enhanced runtime verification to ensure noc functional correctness. In *Proc. MICRO*, 2011.

[157] R. Parikh and V. Bertacco. uDIREC: Unified diagnosis and reconfiguration for frugal bypass of NoC faults. In *Proc. MICRO*, 2013.

[158] D. Park, C. Nicopoulos, J. Kim, N. Vijaykrishnan, and C. R. Das. Exploring fault-tolerant network-on-chip architectures. In *Proc. DSN*, 2006.

[159] S. B. Park and S. Mitra. IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors. In *Proc. DAC*, 2008.

[160] A. Pellegrini and V. Bertacco. Cobra: A comprehensive bundle-based reliable architecture. In *Proc. SAMOS*, 2013.

[161] A. Pellegrini, J. L. Greathouse, and V. Bertacco. Viper: Virtual pipelines for enhanced reliability. In *Proc. ISCA*, 2012.

[162] M. Pirretti, G. Link, R. Brooks, N. Vijaykrishnan, M. Kandemir, and M. Irwin. Fault tolerant algorithms for network-on-chip interconnect. In *Proc. ISVLSI*, 2004.

[163] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proc. ISCA*, 2009.

[164] A. Prodromou, A. Panteli, C. Nicopoulos, and Y. Sazeides. NoCAlert: An on-line and real-time fault detection mechanism for network-on-chip architectures. In *Proc. MICRO*, 2012.

[165] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proc. ISCA*, 2002.

[166] V. Puente, J. Gregorio, F. Vallejo, and R. Beivide. Immunet: A cheap and robust fault-tolerant packet routing mechanism. In *Proc. ISCA*, 2004.

[167] E. A. Rambo, O. P. Henschel, and L. C. V. dos Santos. Automatic generation of memory consistency tests for chip multiprocessing. In *Proc. ICECS*, 2011.

[168] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. In *Proc. CGO*, 2005.

[169] P. Ren, Q. Meng, X. Ren, and N. Zheng. Fault-tolerant routing for on-chip network without using virtual channels. In *Proc. DAC*, 2014.

[170] M. Rinard. Analysis of multithreaded programs. In *Static Analysis*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001.

[171] S. Rodrigo, J. Flich, A. Roca, S. Medardoni, D. Bertozzi, J. Camacho, F. Silla, and J. Duato. Addressing manufacturing challenges with cost-efficient fault tolerant routing. In *Proc. NOCS*, 2010.

[172] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras. Non-speculative load-load reordering in TSO. In *Proc. ISCA*, 2017.

[173] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang. Fast and generalized polynomial time memory consistency verification. In *Proc. CAV*, 2006.

[174] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1), 2003.

[175] D. Sabena, M. S. Reorda, L. Sterpone, P. Rech, and L. Carro. On the evaluation of soft-errors detection techniques for GPGPUs. In *Proc. IDT*, 2013.

[176] J. Sancho, A. Robles, and J. Duato. An effective methodology to improve the performance of the Up*/Down* routing algorithm. *IEEE Trans. Parallel and Distributed Systems*, 15(8), 2004.

[177] S. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware. In *Proc. MICRO*, 2006.

[178] M. D. Schroeder, A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. Autonet: A high-speed, self-configuring local area network using point-to-point links. *IEEE Journal of Selected Areas in Communications*, 9(8), 1991.

[179] F. Sem-Jacobsen and O. Lysne. Topology agnostic dynamic quick reconfiguration for large-scale interconnection networks. In *Proc. CCGrid*, 2012.

[180] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7), 2010.

[181] O. Shacham, M. Wachs, A. Solomatnikov, A. Firoozshahian, S. Richardson, and M. Horowitz. Verification of chip multiprocessor memory systems using a relaxed scoreboard. In *Proc. MICRO*, 2008.

[182] A. Shafiee, M. Zolghadr, M. Arjomand, and H. Sarbazi-azad. Application-aware deadlock-free oblivious routing based on extended turn-model. In *Proc. ICCAD*, 2011.

[183] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, Inc., 2005.

[184] J. Smolens, B. Gold, J. Hoe, B. Falsafi, and K. Mai. Detecting emerging wearout faults. In *Proc. SELSE*, 2007.

[185] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011.

[186] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. ISCA*, 2002.

[187] A. Strano, D. Bertozzi, F. Trivino, J. Sanchez, F. Alfaro, and J. Flich. OSR-Lite: Fast and deadlock-free NoC reconfiguration framework. In *Proc. SAMOS*, 2012.

[188] A. Strong, E. Wu, R. Vollertsen, J. Sune, G. Rosa, T. Sullivan, and S. Rauch. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. IEEE Press Series on Microelectronic Systems. Wiley, 2009.

[189] S. M. Tam, H. Muljono, M. Huang, S. Iyer, K. Royneogi, N. Satti, R. Qureshi, W. Chen, T. Wang, H. Hsieh, S. Vora, and E. Wang. SkyLake-SP: A 14nm 28-core Xeon processor. In *Proc. ISSCC*, 2018.

[190] L. Tan, B. Brotherton, and T. Sherwood. Bit-split string-matching engines for intrusion detection and prevention. *ACM Transactions on Architecture Code Optimization*, 3(1), 2006.

[191] O. Temam. A defect-tolerant accelerator for emerging high-performance applications. In *Proc. ISCA*, 2012.

[192] S. Thiruvathodi and D. Yeggina. A random instruction sequence generator for ARM based systems. In *Proc. MTV*, 2014.

[193] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi. TriCheck: Memory model verification at the trisection of software, hardware, and ISA. In *Proc. ASPLOS*, 2017.

[194] S. Tselonis, V. Dimitsas, and D. Gizopoulos. The functional and performance tolerance of GPUs to permanent faults in registers. In *Proc. IOLTS*, 2013.

[195] S. Tselonis and D. Gizopoulos. GUFI: A framework for GPUs reliability assessment. In *Proc. ISPASS*, 2016.

[196] E. Viaud, F. Pêcheux, and A. Greiner. An efficient TLM/T modeling and simulation environment based on conservative parallel discrete event principles. In *Proc. DATE*, 2006.

[197] E. Wachter, A. Erichsen, A. Amory, and F. Moraes. Topology-agnostic fault-tolerant NoC routing method. In *Proc. DATE*, 2013.

[198] I. Wagner and V. Bertacco. Reversi: Post-silicon validation system for modern microprocessors. In *Proc. ICCD*, 2008.

[199] I. Wagner and V. Bertacco. Caspar: Hardware patching for multicore processors. In *Proc. DATE*, 2009.

[200] W. Wang, S. Yang, S. Bhardwaj, S. Vrudhula, F. Liu, and Y. Cao. The impact of NBTI effect on combinational circuit: Modeling, simulation, and analysis. *IEEE Trans. VLSI Systems*, 18(2), 2010.

[201] X. Wang, J. Keane, T. T. H. Kim, P. Jain, Q. Tang, and C. H. Kim. Silicon odometers: Compact in situ aging sensors for robust system design. *IEEE Micro*, 34(6), 2014.

[202] D. Weaver and T. Germond. *The SPARC Architectural Manual (Version 9)*. Prentice-Hall, Inc., 1994.

[203] B. Wile, J. Goss, and W. Roesner. *Comprehensive Functional Verification: the Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., 2005.

[204] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. ISCA*, 1995.

[205] J. Wu. A fault-tolerant and deadlock-free routing protocol in 2D meshes based on odd-even turn model. *IEEE Trans. Computers*, 52(9), 2003.

[206] B. Zandian, W. Dweik, S. H. Kang, T. Punihaole, and M. Annavaram. WearMon: Reliability monitoring using adaptive critical path testing. In *Proc. DSN*, 2010.

[207] J. J. Zhang, T. Gu, K. Basu, and S. Garg. Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator. In *Proc. VTS*, 2018.

[208] Z. Zhang, A. Greiner, and S. Taktak. A reconfigurable routing algorithm for a fault-tolerant 2D-mesh network-on-chip. In *Proc. DAC*, 2008.