# Correct Communication in Multi-core Processors

by

**Andrew Whitehouse DeOrio**

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
2012

To my family

# Acknowledgments

I would like to thank my advisor Professor Valeria Bertacco, who introduced me to research. Her willingness to let me explore new ideas has been a wonderful freedom, and our discussions of engineering trade-offs have improved my work. Moreover, my writing and presentation skills have grown significantly as a result of her mentoring and her careful eye.

I am also grateful for my committee members. Professor Todd Austin has been a consistent source of valuable feedback throughout my studies. I am appreciative of Professor Scott Mahlke's contributions, and our friendly interactions at the north campus gym track. My musical tastes have expanded as a result of my relationship with Professor Stephen Rush. He introduced me to the pyrophone [75], and to the saints of free jazz. Dr. Raj Yavatkar has been a mentor to me in the professional world, as well as the academic world, supporting my work at Intel Corp. and at the University.

Early in my graduate school career, I was fortunate to work with Ilya Wagner – he taught me the ropes of being a graduate student. I am grateful for Joseph Greathouse and Andrea Pellegrini for our illuminating discussions and colorful experiences. I am also appreciative of the people I have collaborated with on many projects: Adam Bauserman, Debapriya Chatterjee, Ilya Wagner, Rawan Abdel-Khalek, Ritesh Parikh, Daya Khudia, Jialin Li, Qingkun Li, Matt Burgess, Jin Hu, Gregory Chen, Kostantinos Aisopos, David Fick, and Professors Li-Shiuan Peh, Dennis Sylvester, and David Blaauw.

Finally, I would like to thank my family: for my mom, for many Thursday night dinners with Dad, and for my siblings Allison and Scott.

# Preface

Computer chips, the most complex artifacts ever made by man, are susceptible to problems with correct functionality due to their intricacy. Incorrect operation of silicon chips has lasting, and sometimes devastating, effects on computer systems and their manufacturers: from incorrect computation results, to security vulnerabilities affecting end users, to financial impact on the vendors. Furthermore, new chips are increasingly fragile, liable to break as the transistors that comprise them become small enough to be measured in atoms.

A typical modern computer usually includes a single chip where many processors are connected by a communication medium. This communication medium, a new feature in modern chips, provides many opportunities for catastrophic errors, as it is a complex, unpredictable, unique component.

The goal of this dissertation is to provide a new solution to ensure the correct operation of the communication medium in multicore processors, from the early stages of design to the end user. It addresses failures in several modes, and operates across the different phases of the verification process, integrating them into a cohesive framework. A key finding of this work is the synergy among verification phases, connected by a novel abstraction technique and multipurpose hardware and software. Simply put, it ensures that the design operates as intended. This approach to the development cycle accelerates, automates and extends the reach of the verification process, providing decreased occurrence of — and increased resilience to — failures. With this solution, the communication system of multi-core chips can operate free from errors.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Microprocessor Errors and Opportunities to Mitigate Them

We rely on unreliable devices. Digital computers are at the heart of the phones we use to communicate with each other; they run banking and stock market systems for business and commerce. Automobiles and airplanes are dependent on computer systems for engine function and navigation, and computers can even be involved in international conflict, as was the case with the STUXNET virus [49]. However, there are many challenges to the correct operation of these devices.

Modern computer chips are vastly complex systems comprised of billions of tiny transistors: they are the most complex artifact yet created by man. The doubling of the number of transistors every 18 months driven by Moore's law has moved processor designs from 2,300 transistors with the Intel 4004 in 1971 to recent Intel microprocessors with 2.6 billion transistors. Large transistor counts give rise to today's chip multi-processors, incorporating many cores on one die. Recent IBM chips, such as the POWER7 [126], contain 8 cores, and the Tilera Tile-Gx series ships with 100 cores. Intel has released two experimental chips: the 40 core Single-chip Cloud Computer (SCC) [97], as well as the 80 core Polaris [144].

As the number of cores in a single chip continues to rise, the complexity of these chips also increases. Cores must communicate in order to coordinate their efforts, and as a result, the communication subsystem becomes increasingly elaborate. This system is large, distributed and complex: thus, a large number of interactions must be verified during the design process. In 2008, a 16 core Sun chip required 100 person-years of verification [142], some Intel estimates are in the thousands of person-years [133], and estimates are rising. This daunting and incomplete verification task presents the risk of errors (bugs) escaping into final silicon. This is a prominent issue in processors, where design bugs are often detected after the release of the product, as can be noted in several processor errata documents [6, 68, 69, 67, 70].

In addition to increasingly risky bugs, shrinking transistors foment shrinking device reliability. Current microprocessors are available with transistor dimensions as small as 22nm, corresponding to less than 100 silicon atoms. With critical dimensions shrinking, only a few misplaced atoms may cause a catastrophic device failure. Thus, the possibility of transistors and wires wearing out in the field may soon become a reality.

Increasing core counts and decreasing transistor dimensions create a situation where the communication subsystem is a critical point of failure. Correct operation of this system is jeopardized by transistors susceptible to faults and latent design bugs.

## 1.1 Impact of Failures

Design failures can have impact ranging from minor delays in a product's time-to-market, to the viability of an entire company. Failures can occur early in the development cycle, late in the cycle, or in the final product. Latent failures in the final chip can also be a security concern, opening new avenues for hackers to exploit a system.

Failures **early in the design process** are limited to problems with the functionality of the design, bugs that prevent it from working as expected. At this stage, the design is typically engineered through simulations: fixing a bug first involves understanding the issue. Then, the error can be fixed by modifying the source code. The impact of a bug caught late in this phase may be limited to a schedule delay of a few weeks or months.

**Later in the design process**, silicon prototypes are manufactured for fast, at-speed testing. Failures caught at this stage require re-tooling the manufacturing process for a modified design, called a re-spin. Re-spins may require several months of delay and are very expensive due to the high re-tooling cost, which can range from approximately \$3 million to \$30 million.

Following prototype testing, a new **chip can be shipped to customers**. At this late stage, failures have wide-spread and critical impact. A recall on a faulty product can take a year, at which point newer, competing products may already be available. The life of a company can be jeopardized by failures in the field. For example, the infamous Pentium FDIV bug was discovered in 1994. This bug caused some floating point division operations to compute the wrong result. Ultimately, the defective processors were recalled at a cost of \$475 million [94]. Some of the recalled processors were turned into commemorative key chains.

Transistor wear-out can also result in recalls: a recent Intel 6-Series chipset (codenamed Cougar Point) experienced a wear-out problem in the field. Prematurely aging transistors in one of the SATA controllers caused increasing errors on the link over time. The only fix was a recall in early 2011, at a cost upwards of $1 billion.

## 1.2 Types of Errors

While all errors prevent the correct operation of the chip, their root causes can be different. This dissertation addresses three major categories of errors (Figure 1.1). At the highest level of abstraction, functional bugs are instances where the design does not conform to its specification. Electrical failures manifest at the circuit level, where gates do not perform as expected. Finally, at the lowest level of abstraction, transistor faults are visible as malfunctioning silicon devices or conductors.

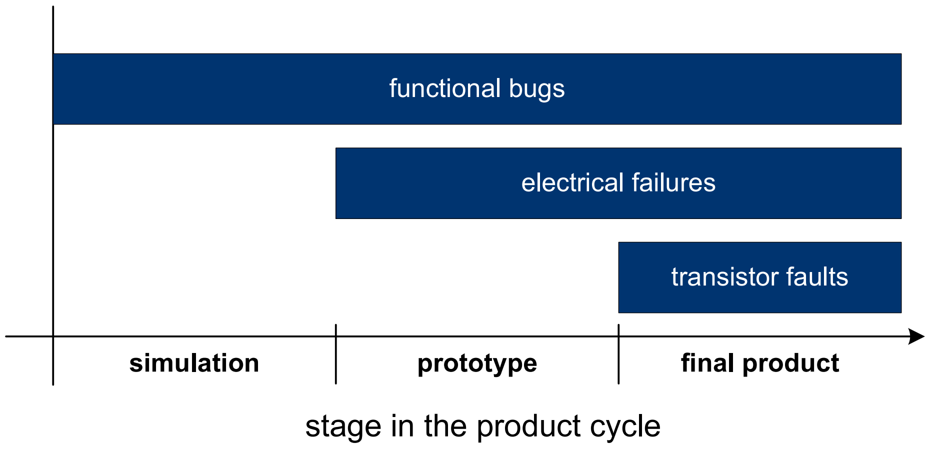


**Figure 1.1 Types of errors addressed by this dissertation** and when they occur. Functional bugs, present from the very first design models, can be addressed in any phase of verification. Electrical failures begin to appear with the first prototypes, and transistor faults due to wear-out occur only in the final product.

**Functional bugs** occur when the functionality of a design does not match its specification, or when the specification is incomplete. These failures are present in a digital design beginning with the first simulations. Functional bugs are deterministic and reproducible in simulation; however, real silicon prototypes may contain latent bugs that do not manifest deterministically due to environmental and electrical variations. An example of a functional bug is the previously mentioned Pentium FDIV bug, where some division operations produced the wrong result.

**Electrical failures** occur in circuits that do not meet voltage, current or timing constraints. This can be due to circuit design problems, as well as manufacturing defects, and occur most often in early silicon prototypes. Electrical errors include timing failures, where a signal does not reach its destination in time, and may also be due to insufficient drive current or voltage distribution. The end result is a circuit that fails to produce the desired output under some conditions. On-chip conditions such as temperature and voltage vary from chip to chip, as well as with the executing program. Thus, electrical failures are not always deterministic. The time-consuming, manual process of debugging electrical failures is a critical barrier in a chip's time-to-market schedule [73].

**Transistor faults** can be caused by a variety of wear-out mechanisms in highly scaled technology nodes. As transistor dimensions approach the atomic scale, oxide breakdown [136] becomes a concern, since the gate oxide tends to become less effective over time. Moreover, negative bias temperature instability (NBTI) [9] is of special concern in PMOS transistors, where increased threshold voltage occurs over time. Additionally, thin wires are susceptible to electromigration [58], because conductor material is gradually worn away during chip operation until an open circuit occurs. Since these mechanisms occur over time, traditional manufacturing tests and burn-in procedures are ineffective in detecting them. An example of a transistor failure is the Intel Cougar Point chipset recall discussed earlier.

This range of failures illustrates the variety of ways in which a digital design may err, despite the best intentions of designers and multi-billion dollar validation efforts. The goal of this work is to provide correct operation by addressing the three sub-problems of functional bugs, electrical failures and transistor faults.

## 1.3 Avoiding Failures and Ensuring Correctness

Ensuring the correctness of a digital design spans the entirety of the design process. Beginning with abstract design models, manufacturers use simulation to conduct pre-silicon verification. Once early prototypes are available, engineers run tests on real silicon during post-silicon validation. Finally, runtime verification employs reliability-enhancing hardware features to check correctness as computation progresses. In the end, the goal of design verification is to ensure correction operation throughout the lifetime of the chip.

**Pre-silicon verification** operates on an abstract design model, and can be based on simulation or formal methods. Formal verification techniques rigorously prove that a design satisfies properties: theorem provers, model checkers and symbolic simulators are examples of formal tools. While these techniques are capable of checking the correctness of a design aspect under all possible execution situations, formal methods can only be deployed for very simple designs. With large designs, the state explosion problem occurs, due to the exponential memory requirements of formal tools. Additionally, formal verification is only as effective as the properties that are checked, a manual process susceptible to human limitations. Writing a complete set of formal properties for a large design can even require more time than the design itself. Today, formal verification is limited to abstraction versions of the design, such as the Mur$\varphi$ [44] tool.

In contrast to formal tools, simulation-based verification is the mainstream approach used in industry to validate large designs and correct any design errors. Here, a model of the design is simulated using hand-written programs or constrained-random inputs. Due to the complexity of the design state space, only a small fraction of the design's functionality can be validated within the available development time window. Bugs at this stage are easier to diagnose than in real silicon, since simulation has the advantage of being fully deterministic and fully observable. Failing testcases can be reliably reproduced to diagnose functional bugs, which in turn manifest consistently. Unfortunately, the slow speed of pre-silicon simulation limits verification to a small set of very short testcases, despite the deployment of large server farms devoted to the task.

**Post-silicon validation** begins when the first silicon prototypes become available. Here, long programs such as operating systems, as well as extensive constrained-random tests, can be executed directly on the prototypes, which are orders of magnitude faster than pre-silicon simulation. Fast execution speed enables high coverage testing at this stage. Tests outcomes are validated with a mix of hardware assertions, comparison of test outputs against a golden model, or with the aid of self-checking mechanisms. A test failure indicates an error, which can be functional, electrical (process, logic or circuit related), or due to a missed manufacturing defect. To diagnose the error, validation engineers must re-run the failed test with the support of a post-silicon validation hardware platform that provides a limited amount of control of test advancement and debugging features, such as on-chip logic analyzers and/or configurable embedded checkers [2, 110]. However, post-silicon failure diagnosis is notoriously difficult, especially when targeting tests that do not fail consistently over multiple runs. The limited observability and controllability characteristics of post-silicon further exacerbate this challenge, making post-silicon diagnosis one of the most difficult tasks of the entire validation effort.

**Runtime verification** solutions have been proposed by the research community to detect and correct failures in the final product. A common trait of these solutions is the use of on-chip checkers to detect functional bugs and transistor faults [147, 148, 14]. If an error is detected, most solutions will provide a correction mechanism, which enables the processor to overcome the errant configuration at a performance cost. Hence, the impact of an error in a hardware module protected by an online verification mechanism is limited to performance degradation, rather than incorrect results or, possibly, a system crash. Thus, a key benefit of these runtime verification solutions is that they enable the verification team to focus its efforts on the design's execution scenarios that arise most frequently.

Furthermore, runtime approaches are the only type of solution able to handle faults caused by transistor wear-out. Since these faults only occur after substantial time and usage, traditional burn-in and manufacturing tests at the factor are unable to mitigate wear-out related errors. Failure mechanisms such as oxide breakdown [136], negative bias temperature instability (NBTI) [9] and electromigration [58] all result in errors occurring later in the lifetime of the chip. Thus, runtime solutions are required to alleviate these errors. In the end, runtime verification enhances the reliability of the final chip, making it resilient to functional bugs, electrical failures, and transistor faults.

## 1.4 Dissertation Overview

Digital designs today face increasing complexity and decreasing reliability, which challenge the ability of the current design process to deliver a correct product that can function for years. This dissertation develops a number of solutions that work together in a novel validation framework. Called BiPeD, this framework addresses the problem of correctness within the communication subsystem of a chip multi-processor. The components of our solution are multi-pronged, leveraging opportunities in pre-silicon, post-silicon, and runtime verification. As a whole, they take on the complexity of modern designs by automating and accelerating the verification process, increasing the understanding of the design, and easing debugging. This enables increased coverage, thereby decreasing bugs in the final product. Furthermore, our approach is a long-term solution aware of the lifetime reliability challenges of fragile chips in the field.

Figure 1.2 shows an overview of the solution proposed in this dissertation, which synergizes the different phases of verification to address a variety of errors. The figure divides correctness problems into three failure modes: functional, electrical and transistor (vertical axis). Electrical failures also include manufacturing defects that have escaped line testing, and transistor faults indicate errors that occur due to wear-out. Our solution leverages three verification stages to provide a comprehensive solution, beginning with pre-silicon verification (horizontal axis). Post-silicon validation allows testing of early chips, and runtime solutions enable resilience to transistor faults. The integral components of our solution are completely covered by the shaded region, while complementary parts are partially covered.

In Chapters 3 – 5, we discuss new solutions that address these errors, and how they fit into the BiPeD framework. We evaluate this framework in more detail in Chapter 6,

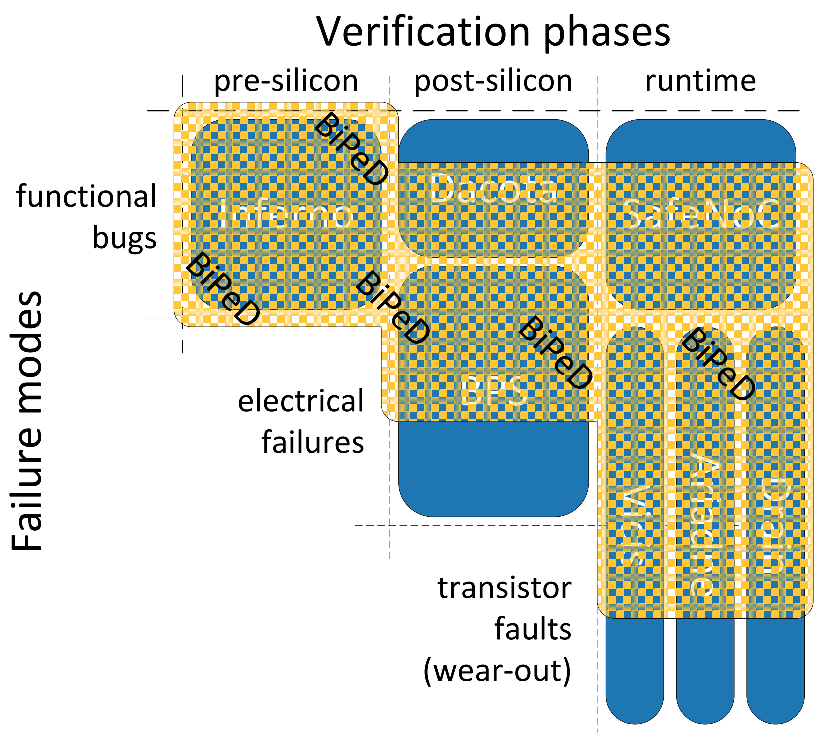


**Figure 1.2 Overview of the solution framework proposed by this dissertation**, which synergizes the different phases of verification within the BiPeD framework to address three modes of failure: functional bugs, electrical failures and transistor faults (vertical axis). To accomplish this, it leverages opportunities in the three phases of verification, pre-silicon, post-silicon and runtime (horizontal axis). Integral component solutions to the BiPeD framework are completely shaded, while complementary solutions are partially shaded.

bringing together the different phases of verification, enabling reuse of verification effort and deploying multipurpose verification hardware.

## 1.5 Dissertation Organization

The remainder of this dissertation is organized by error type. First, Chapter 2 presents a look at the current state of the art in verification. It describes the process by which modern microprocessors are designed and verified using pre-silicon verification, post-silicon validation, and runtime verification. The three modes of failure are addressed in Chapters 3–5, following the flow of Figure 1.2.

Chapter 3 addresses functional bugs, which may be present from the first models of the design and persist through shipment, manifesting in customers' chips. Thus, we take advantage of all three phases of verification to address functional bugs. During pre-silicon verification, we establish an abstraction that defines the correct behavior of the system protocols. This integral part of the BiPeD framework is a software tool for aiding the understanding of the complex protocols that make up the communication subsystem (Section 3.2). The protocols learned during pre-silicon verification are then leveraged by BiPeD during high-speed post-silicon validation, where flexible hardware monitors the on-chip protocols. When an error is detected, Dacota (Section 3.4) is applied to narrow down the source, targeting communication orderings, a common source of errors. Finally, we propose a solution to ensure end-to-end correctness of a network-on-chip communication infrastructure with SafeNoC (Section 3.5).

We address electrical failures in Chapter 4, which may be present beginning with the first silicon prototypes. First, flexible BiPeD hardware detects the occurrence of failures. Then, we invoke BPS, a method to debug and narrow down these difficult errors (Section 4.3) during post-silicon validation. When electrical failures escape post-silicon validation, a runtime solution can address both electrical failures and transistor faults simultaneously.

Transistor faults are discussed in Chapter 5. Runtime verification is the only chance to mitigate wear-out induced transistor faults. Our approach begins by once again leveraging BiPeD's flexible hardware to detect faults. Next, the fault location is diagnosed by Vicis (Section 5.3). Following diagnosis, Ariadne reconfigures around the faults (Section 5.4, and Drain recovers lost data (Section 5.5).

After addressing the three types of errors, we evaluate our BiPeD framework in Chapter 6, which bridges the phases of verification. BiPeD automates understanding the design during pre-silicon verification, and leverages this information to detect errors and debug during post-silicon validation. At runtime, flexible hardware previously used for post-silicon debugging is repurposed as an error detection mechanism. Finally, Chapter 7 briefly summarizes the conclusions of this dissertation.

# Chapter 2
# Verification: State of the Art

The design and manufacture of a microprocessor consists of a series of steps which transform a high-level description of the design into a physical chip. The complexity of microprocessors increases with each generation of chips, bringing new design verification challenges. A typical design verification flow for microprocessors is shown in Figure 2.1, and begins with a specification, a high-level prose description of the desired functionality. First, an architectural model is created from the specification, simulating the functional input/output behavior of the future chip. This model is often written in a high-level programming language, such as C, and is used to refine the details of the specification. It is also used as a reference model for the next steps. As the design cycle continues, a register transfer level (RTL) model is created. This detailed, cycle-accurate model describes the logic and storage that will comprise the chip, and is written in a low-level hardware description language (HDL), such as Verilog or VHDL. Pre-silicon verification is used to locate functional bugs in both the architectural model and the RTL model. This process involves running tests on both models simultaneously and comparing the results. In addition to testing the models through simulation, formal techniques are applied to prove design properties. This is used to prove the absence of certain types of errors. Unfortunately, these techniques do not scale to the size of modern designs.

The next steps transform the HDL model into a silicon prototype. When the first prototypes are available, post-silicon validation begins, where tests are run at full speed on real hardware. At this stage, it is possible to run much longer tests, which are no longer constrained by the slow execution speeds of pre-silicon simulation. Long constrained random tests, operating systems and real application workloads are examples post-silicon tests. Bugs become more difficult to pinpoint during post-silicon validation, since only a few of the design's internal signals are observable. The majority of the design's signals are inaccessible outside the chip. Errors found at this stage include functional failures that escaped pre-silicon verification, and electrical failures that manifest in the chip's circuits. When an error is found during post-silicon validation, it is typically reproduced in the pre-silicon

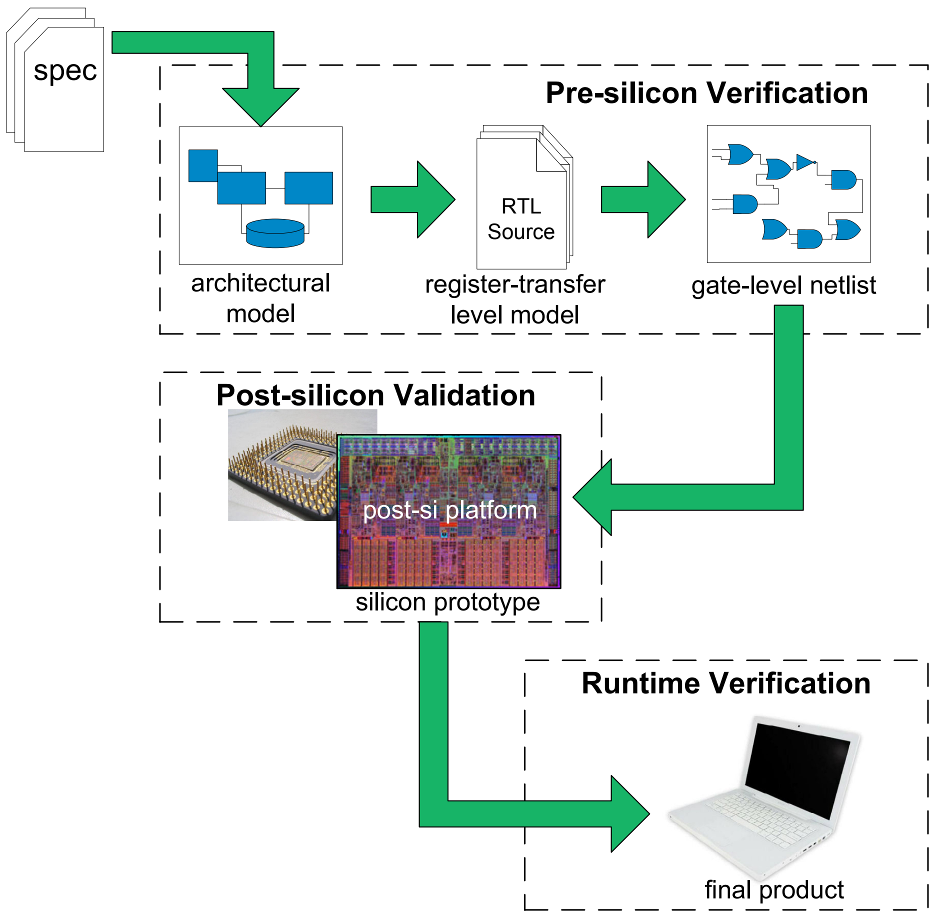


**Figure 2.1 Typical design verification flow for a modern multi-core microprocessor**, which transforms a high level specification to a final product. Three major phases of verification are available to achieve a correct final product: pre-silicon verification is used to verify early models, followed by post-silicon validation, which is performed on silicon prototypes. Finally, runtime verification operates as part of the final product.

model, and the RTL is modified to fix the error. Finally, a new prototype is manufactured, called a re-spin.

Multiple re-spins are often required, as bugs are discovered and fixed during post-silicon validation. When the rate of bug discovery slows and stabilizes, the design is deemed ready for shipment. Because of the vast complexity of a microprocessor design, it is not possible to test every aspect. Thus, designs are shipped with latent bugs. Furthermore, as transistor dimensions shrink, transistors becoming increasingly susceptible to wear-out faults. To mitigate these problems, runtime solutions are implemented as part of the design, enabling it self-check results, and recover from errors. Runtime approaches can be used to address latent function bugs, electrical failures, and transistor faults.

The verification problem can be divided into three major approaches: pre-silicon, post-silicon and runtime. During early design phases, pre-silicon verification is performed on design models, using either formal verification, or simulation-based verification. Next, post-silicon verification works on early silicon prototypes. Runtime solutions ensure correct functionality in the shipped product. Despite these efforts, bugs still slip through the verification process. In addition to the three major approaches to hardware verification, solutions from software verification offer some insights into the verification of hardware. The next sections go into detail on each of these three verification approaches, and discuss the verification target: the communication subsystem.

## 2.1 Pre-Silicon Verification

Pre-silicon verification is used to discover and fix errors in early design models. Since no prototypes are yet available, it is limited to finding functional bugs. There are two major types of pre-silicon verification, formal methods and simulation-based verification.

### 2.1.1 Formal Methods

Formal verification methods exhaustively prove that a design satisfies properties under all conditions. It has the advantage of being able to prove that a portion of the design is bug-free, however, it has significant drawbacks that prevent it from being applied to large designs.

One of the mainstream tools used by the EDA community to analyze complex graphs is model checking, a formal verification technique used to rigorously prove properties of a hardware design. Model checking has the ability to reason about a data set using properties, which are formal descriptions of a particular functionality of a design. Properties are verified by the model checker against an internal mathematical representation of the input design. Model checking uses specialized formalisms and logics to reason about graphs. These include including computation tree logic (CTL) and linear temporal logic (LTL) [32]. Formal verification of the memory subsystem notably includes several notable approaches [3, 57, 44, 114].

The first and foremost problem with formal verification is scaling. Formal verification suffers from state space explosion, since the techniques require exponential memory to implement. Furthermore, formal verification tools require a set of properties to prove. The quality and completeness of the properties is directly related to the quality and completeness of the verification exercise. Moreover, while formal verification of abstract representations has been accomplished with success, its results are inadequate, as the implementations of these abstract models are much more complex and cannot be validated. For these reasons, formal verification is limited in practice to very small design blocks, leaving the large, system-level properties to slow, incomplete pre-silicon simulation.

### 2.1.2 Simulation-based Verification

Logic simulation is a central aspect of the modern integrated circuit development process. It is the primary tool used to validate a wide range of design aspects, foremost among these being the correctness of the system's functionality, both in its behavioral description, as well as in its structural (gate-level) one. Most industry design flows invest the largest portion of their time and resources precisely on this task [47], in an attempt to provide the best possible guarantee that the system satisfies its original functional specification. Large server farms, comprising thousands of machines, execute billions of cycles of simulation for months at a time.

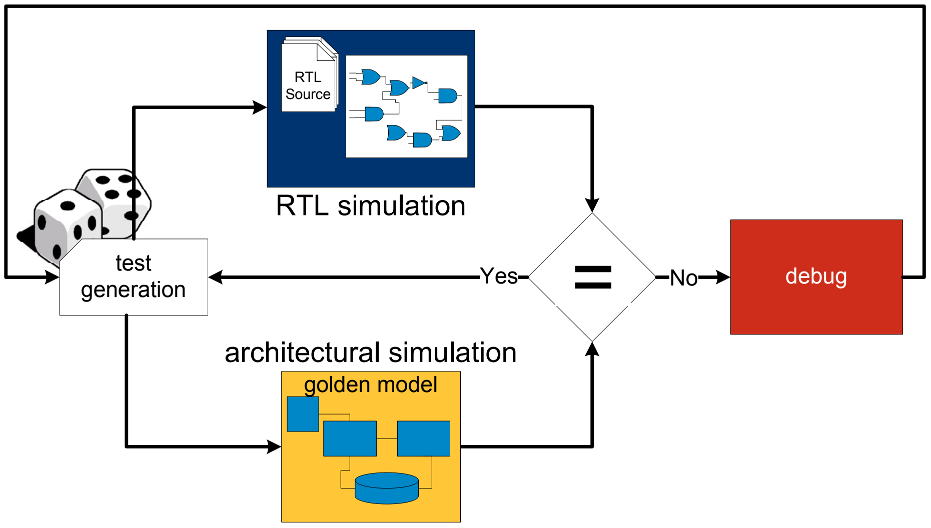


**Figure 2.2 Simulation-based pre-silicon verification for microprocessors** is used heavily in industry. A constrained-random test generator produces valid, random programs, which are provided as input to both an RTL simulator and golden model. If the results match, testing continues, otherwise, the bug is fixed manually.

Simulation-based pre-silicon verification, shown in Figure 2.2, is the workhorse of modern pre-silicon verification of microprocessors. Typically, a constrained-random test generator produces input programs, which are run concurrently on an RTL model simulation, and on the architectural golden model. RTL simulations run much slower than architectural one, sometimes at slowdowns on the order of millions of times. When both simulations are complete, their outputs are compared, ensuring that final memory and architectural state match. In addition to checking results against an architectural model, assertions augment the design under simulation, performing software checks as simulation progresses. When a check fails, the testcase stops, and the location of the failure is identified.

Despite the vast effort of time and resources, functional validation remains an incomplete task, with large portions of the design going unverified. Indeed, while common case scenarios are often checked in this process, buggy and rare corner cases frequently slip through validation, and are consequently latent in the final product, potentially causing malfunctions in the field.

### Logic Simulators

Logic simulation entails evaluating the response of a design over time when subjected to a set of input stimuli, typically selected by the designer to be representative of practical use situations. For most synchronous designs the response is computed once for each cycle of simulated execution. Modern logic simulators read in a design description, then "compile" it to produce machine code emulating the same functionality as the design's primitives, and finally optimize it to minimize the amount of computation required to provide the responses that the user wishes to observe. The input stimuli are commonly provided in the form of a testbench, that is, a program describing implicitly or explicitly the set of input values for each clock cycle of simulation. The testbench may be direct, where input values are selected by a verification engineer, or pseudo-random, that is, inputs are set by a generator abiding pre-set constraints and statistical distributions.

Simulators can be grouped into two families based on their internal architecture: *oblivious* simulators compute all gates (or logic) in the system during every simulation cycle and entail a simpler software design. Oblivious simulators have the advantage of low control overhead, but can spend significant computation time unnecessarily evaluating gates over and over whose output values do not change from cycle to cycle. *Event-driven* simulators limit the amount of computation by selectively simulating in each cycle only those gates who inputs have changed since the previous cycle, and whose output may thus change in response to the switching stimulus. While the sequencing of gate evaluation in oblivious simulation can be statically determined at compile-time, event-driven simulators require a dynamic runtime scheduler, hence entail a more complex software structure. However, this latter approach is vastly more common in commercial tools because the scheduler performance overhead is largely offset by the fact that for many designs only 1 to 10% of the gates switch at each cycle, thus requiring significantly less computation.

Simulation approaches, in contrast to formal methods, are able to tackle a much larger portion of the design. For example, Wood, *et al.* [152] leverages constrained-random simulation to boost the coverage of a design. Here, there is the additional burden of understanding the interactions within a large design occurring throughout a long simulation. A number of previous works have focused on the problem of automatically extracting properties and specifications. Hangal, *et al.* [62] have proposed a tool to extract simple "probable" properties (*e.g.*, one-hot or mutually exclusive signals) through simulation trace analysis, which can then be fed to a formal property checker for verification. In [50], the authors propose a more general approach to automatic property extraction, by evaluating a wide range of possible "time relations" between groups of signals.

## 2.2 Post-Silicon Validation

Following pre-silicon verification, post-silicon validation begins when the first prototype chips are available, and is used primarily to validate large microprocessor designs. It is the first opportunity to test a physical implementation of the design, and thus it is able to uncover problems with the design's circuits (electrical failures). Additionally, functional bugs that evaded pre-silicon verification may be identified at this stage.

Post-silicon validation is conducted on prototype chips connected to specialized validation platforms (Figure 2.3). The platforms are used to run post-silicon tests, a mix of directed and constrained-random workloads. Upon completion of each test, the output of the silicon prototype is typically checked against an architectural simulator. When an error causes a check to fail, the debugging process begins, seeking to determine the root cause of the failure.

In industry practice, the post-silicon validation process begins when the first silicon prototypes become available. These chips are then connected to specialized validation platforms that facilitate running post-silicon tests, a mix of directed and constrained-random workloads. Upon completion of each test, the output of the silicon prototype is checked against an architectural simulator, or in some cases, self-checked [5, 146].

When a check fails, indicating that an error has occurred, the debugging process begins, seeking to determine the root cause of the failure. On-chip instrumentation can be used to observe intermediate signals. Techniques such as scan chains, on-chip logic analyzers [149] and flexible logging infrastructures [2] are configured to trace design signals (only a small number can usually be observed) and periodically transfer data off-chip. Traces are then examined by validation engineers to determine the root cause of the problem. This process is time-consuming and engineering intensive, and is further exacerbated by bugs

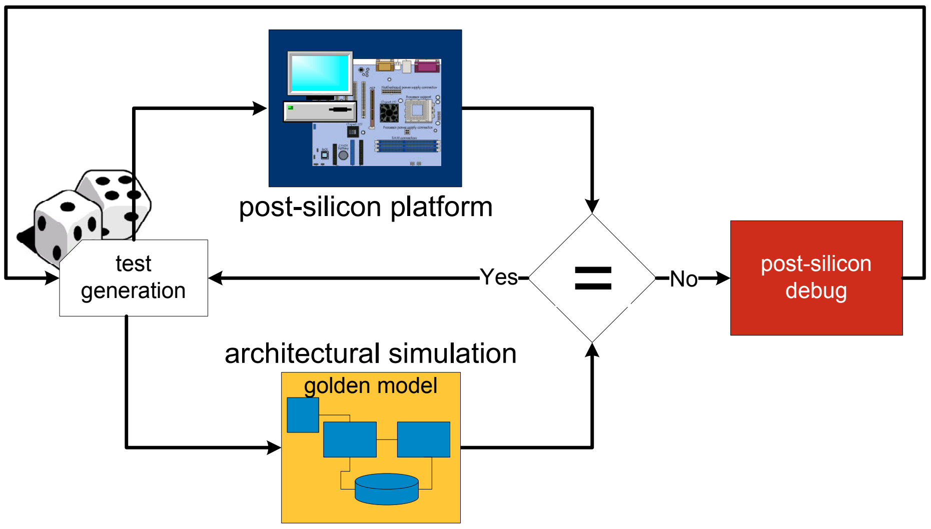
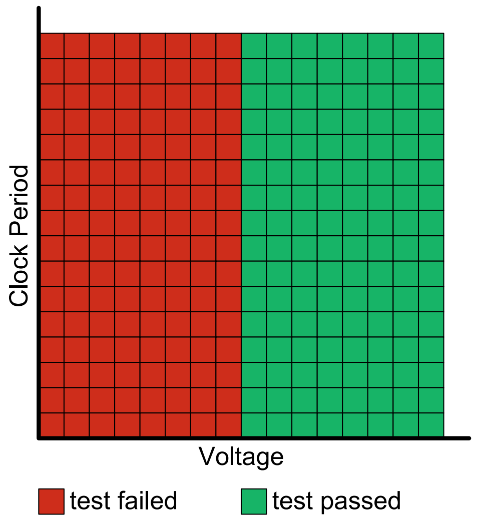


**Figure 2.3 Post-silicon validation flow** typical in industry. Like pre-silicon verification, a constrained-random test generator produces valid, random programs, which are provided as input to both the golden model and the silicon prototype. The prototype is connected to a validation platform that includes a host CPU. If the results match, testing continues, otherwise, the bug is fixed manually.

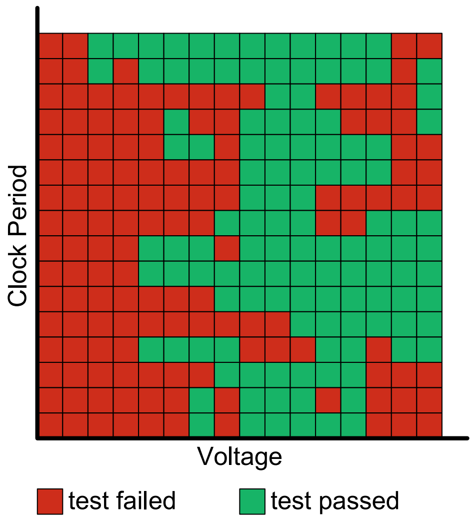
with inconsistent outcomes. Additionally, off-chip data transfers are very slow, which further hinders observability due to limited transfer time. Thus, our goal with this portion of the dissertation to reduce debugging effort, automatically diagnosing the time and location of bugs, while minimizing off-chip transfers.

A powerful industry approach to locating electrical failures is the shmoo plot [13], which shows the relationship between chip failures and changes in environmental conditions. In a typical shmoo plot, two primary parameters are varied and assigned to the X and Y axis of the plot, respectively. For example, the parameters might be voltage and clock period. Then, a single test is run many times, with different parameter values, and the pass/fail status of the test is recorded. With some parameter value pairs, the test passes, with others, it fails. Figure 2.4 shows two theoretical examples of typical shmoo plots, with clock period and voltage as the parameters. Each cell in the plots represents the outcome of one test, with passing results in green, and failing results in red.

The debugging process of non-deterministic failures can be aided by deterministic replay mechanisms [104, 123]. However, these solutions perturb system execution which can prevent the bug from manifesting, and often incurs significant hardware and performance overheads. In addition, in an effort to automate the failure diagnosis process, methods based on formal verification techniques have been proposed [38, 77, 120, 156]. These solutions require deterministic execution and a complete golden (known-correct) model of the design for comparison. However, the limitations of formal methods discussed earlier



**(a)** Clear relationship



**(b)** Relationship difficult to identify

**Figure 2.4 Theoretical shmoo plots** showing test pass/fail status as a function of two parameters, voltage and clock period. These plots are used to debug electrical failures during post-silicon validation. The left plot shows a clear relationship between the error and voltage, while the relationship in the right plot is not as clear. One goal of this dissertation is to help localize those bugs that are most difficult to reproduce.

preclude these techniques from handling industrial size designs. Methods that target small fully deterministic systems [56, 88, 155] are capable of identifying limited types of electrical errors in small circuits. Other solutions targeting different types of errors leverage Bayesian approaches [72] for transient errors and analysis of bug reports for software errors [90]. On the other hand, functional failures have been approached by recording system state using a scan chain, and then comparing passing and failing tests [112]. Early work in troubleshooting circuit boards used signatures to achieve compact observations [54, 124].

Specialized post-silicon debugging approaches often add dedicated hardware units for debugging specific areas, such as the memory subsystem or speed paths [98]. Solutions such as IFRA/BLoG [109, 108] can localize electrical bugs in the processor core, as long as errors are detected on-chip within about 1,000 cycles. When detection takes longer times, modifying the post-silicon test [65] can help, but at the risk of perturbing the system.

An ideal post-silicon validation solution is flexible, and applicable to multiple on-chip subsystems. It minimizes perturbation of the existing logic, as well as minimizing execution overhead. The high speed of post-silicon test execution is an ideal opportunity for increased coverage, and for testing system-wide properties that exceed the limitations of pre-silicon verification.

## 2.3 Runtime Verification

Despite massive industry verification efforts at both the pre-silicon and post-silicon stages, they cannot cover the vast number of operations and interactions in a complex chip. As a result, critical design errors can often escape verification and manifest in the released hardware at runtime. Furthermore, problems due to transistor wear-out don't occur until a chip has been in the field for some time. Thus, runtime verification can be applied to all three major modes of failure: functional bugs, electrical failures, and transistor faults. When runtime verification is applied to transistor faults, it can be referred to as design-for-reliability.

To address the limitations of design-time verification, runtime verification solutions add hardware mechanisms to address on-chip errors. A common trait of these solutions is the use of one or more on-chip checkers which can detect all or some functional errors by monitoring the design's execution. If an error is detected, most solutions will provide a correction mechanism, which enables the design to overcome the buggy configuration at a performance cost. Hence, the impact of an error in an aspect protected by an online verification mechanism is limited to a graceful performance degradation, rather than incorrect results or, possibly, a system crash. Thus, a key benefit of these runtime verification solutions is that they enable the verification team to focus its efforts on the design's execution scenarios that arise most frequently, and relieve the burden of striving to fully verify a design before its release to the market.

Runtime verification has been applied to processor cores, checking pipeline results as computation progresses [148, 99, 145]. In general, these solutions add checker hardware to verify the operation of untrusted components. Solutions to the problem of communication subsystem correctness have also emerged to ensure the correctness of multi-core designs deployed in the field. Meixner, *et al.* [100] approach the problem by adding checkers to verify memory consistency. Another runtime solution, proposed by Chen, *et al.* [29], leverages the constraint graph-based approach developed in [27, 127]. This solution adds dedicated observation hardware at each core, periodically analyzing the collected information with a centralized hardware checker.

For networks-on-chip, runtime solutions have been recently proposed to ensure the correct transfer of data packets through the interconnect. Several works focus on the problem of deadlock, with some proposing deadlock-avoidance solutions by forbidding certain routes [135], and others trying to detect and recover from deadlocks. DISHA, for example, leverages timeouts for detection, and then progressively routes blocked packets through a deadlock-free dedicated link [11]. Other deadlock detection techniques, such as [96] and [92], propose more sophisticated mechanisms based on monitoring the activity at the physical channel level.

On-chip networks have tight area and power budgets, necessitating simple router structures. Architectural approaches to reliable router architectures include triple modular redundancy (TMR) based approaches, such as the BulletProof router [33]. However, in general, *N*-modular redundancy (NMR) approaches are expensive, as they require at least *N* times the silicon area. Another strategy explores the trade-offs of various levels of redundancy [106]. Other work investigates the reliability of single components, for example a reliability-enhanced crossbar [63]. Reconfiguration is approached by [61] for pipelines, by [83] for link failures and by [79] with modular design. Protection against transient errors has been explored in [17, 160, 107].

Additionally, many works propose reliable routing algorithms, able to reroute the network around failed nodes and links. We discuss the state of the art in reliable routing algorithms in Section 5.4.1.

By contrast, checkpointing schemes [132, 115] can be leveraged to provide error recovery for the network, attempting to recover system state when an error is detected. A checkpoint-enabled system logs cache data and architectural state, periodically copying these data to main memory. In the event of an error, state can be recovered from the logs in memory and the system is rolled back to an earlier execution point. These mechanisms work in a proactive manner, always preparing for an error. Consequently, they require significant hardware overhead to accommodate buffers, which can be on the order of 512KB in size [132]. Moreover, they incur performance overheads during normal operation, which can exceed 6% [115], an overhead that is incurred even in the absence of errors.

Other runtime solutions adopt more general end-to-end approaches to handle various errors in the network. A common scheme is the acknowledgment-based end-to-end error detection and recovery technique, in which a data packet is augmented with error detection codes at the source and checked for data corruption at the destination. A successful packet transfer is completed by sending back an acknowledgment messages. Only after the reception of the acknowledgment, a copy of the packet stored at the source is deleted. If the acknowledgment is not received within a certain time interval, the packet can be retransmitted using the source copy. Similar approaches have also been proposed at the switch level, where upstream routers wait for acknowledgements from the downstream routers before deleting a copy of the data from their buffers[103]. Although this is a simple scheme, the additional buffer storage required for its implementation introduces significant area overhead and errors may re-occur upon retransmission. Moreover, the acknowledgment traffic degrades the overall runtime performance of the network, even without errors.

## 2.4 Software Verification

The software verification community, faced with similar challenges, has also developed a number of relevant solutions. Ernst, *et al.* [48] have proposed Daikon, a tools that analyzes software execution traces to suggest a list of possible properties (or annotations) for use with the static checker ESC/Java. Properties can also be generated using static analysis, as in [4, 16]; however, in [16] a program must first be translated into a state machine, a step that may add notable complexity to the process. Ammons, *et al.* perform an analysis to generate a specification of a given application program interface (API) in the form of "scenarios" describing common sequences of instructions [10], while Yang derives constraints on the order of occurrence of instructions [154]. The value of such scenario- or transaction-based simulations and analyses is well recognized. Brahme, *et al.*, for instance, have developed a system to allow verification engineers to write testbenches and analyze results at the transaction-level [24].

## 2.5 Verification Targets

In this dissertation, we address the problem of verifying the communication subsystem. The communication subsystem is responsible for connecting on-chip resources. When the number of cores on chip is small, technologies such as buses and crossbars are sufficient, however, they may soon be precluded by the delay of long wires and the large number of elements that must communicate with each other. Networks on Chip (NoC) help mitigate this problem by decentralizing and distributing communication across the chip using a lightweight networking protocol, resulting in a high throughput, scalable solution. Examples include the 48-core Intel SCC [97], the 64-core Tilera Tile64 [15], and the experimental 80-core Intel Polaris chip [144]. Many components of the solution presented in this dissertation flexible, and apply to different communication infrastructures, in addition to NoCs.

In a typical NoC architecture, each processor core is connected to a router through a dedicated network interface unit. Data messages are divided into packets, which are in turn partitioned into smaller blocks called flits. Packets and flits are transmitted over the interconnect according to a routing protocol. To efficiently handle the communication load among the many cores on chip, these network interconnects are becoming increasingly complex, often implementing a wide range of topologies and providing advanced communication protocols, such as adaptive routing. Moreover, routers are designed to include advanced features such as virtual channels, pipelining, complex allocation schemes, speculation, etc. With such an intricate communication infrastructure, it is a challenge to ensure that the interconnect subsystem will operate correctly under all execution scenarios.

Even worse, electrical failures and transistor device level failures have architecture level ramifications, as a single faulty link or router will cause an entire NoC to fail, halting all traffic. Future processor technology generations will require significant error tolerance to many simultaneous faults.

On-chip networks provide excellent opportunities for building a reliable system, as they provide redundant paths between IPs. As the "glue" that holds a chip together, a NoC should be highly resilient to errors and able to work around functional bugs, faulty routers and links. In contrast, a faulty IP with even little or no protection can be disabled and isolated by the network, thus promoting NoC as a reliable system platform. A disabled IP would not hinder network performance, as the associated router can still be used. An ideal network should be able to diagnose where faults are in its own components and then reconfigure to mitigate those faults, maintaining full connectivity when possible.

# Chapter 3
# Addressing Functional Bugs

Functional bugs, a failure of a design to meet its specification, can manifest in any stage of the design verification cycle. Though these failures are inserted during the pre-silicon design phase, they may not be discovered until post-silicon validation, or even final, shipped silicon. We address functional bugs with our BiPeD framework, which leverages opportunities for supporting solutions during the pre-silicon, post-silicon and runtime verification phases.

In this chapter, we first present an overview functional bugs in Section 3.1. The remainder of the chapter is divided among the opportunities to address functional bugs. During pre-silicon verification, we emphasize an intuitive understanding of the complex protocols underlying the communication subsystem, presenting observed activity in the form of graphical transactions. This is accomplished with Inferno, a software tool that extracts transactions, and is described in Section 3.2. The transactions extracted by Inferno are leveraged by BiPeD to learn the correct behavior of the design. BiPeD then bridges pre-silicon verification to post-silicon validation with on-chip checkers that enforce the previously learned protocols (Section 3.3). Once a post-silicon bug is detected, we narrow down its cause with Dacota, a hardware/software post-silicon approach to checking memory coherence and consistency, a common source of errors in multi-core systems (Section 3.4). To complete our approach in final silicon, we augment the existing CMP interconnect with an end-to-end detection and recovery solution to ensure its functional correctness. This solution is called SafeNoC, and is described in Section 3.5. Finally, in Section 3.6, we summarize our approach to functional bugs.

## 3.1 Functional Bugs

The fast growing complexity of digital hardware designs is exacerbating the challenges of validation and debugging functional bugs. Several factors contribute to the time-consuming nature of these incomplete and ad-hoc tasks, demanding more engineering resources than the design itself. First, pre-silicon debugging is commonly done through a waveform viewer, an arduous task due to the low-level models used to describe a system (*i.e.*, register-transfer level), as well as extremely long simulation traces. Second, the growing adoption of semi-formal verification methodologies to complement simulation-based verification requires engineers to express the correct behavior of a system through a set of properties. However, property description languages are often declarative, hence their use is error prone and adds further challenges for designers, who are commonly trained to use imperative languages. Even worse, these properties are often the result of the personal understanding of a verification engineer who has studied a high-level specification document of the system. A detailed overview of pre-silicon verification techniques is provided in Section 2.1.

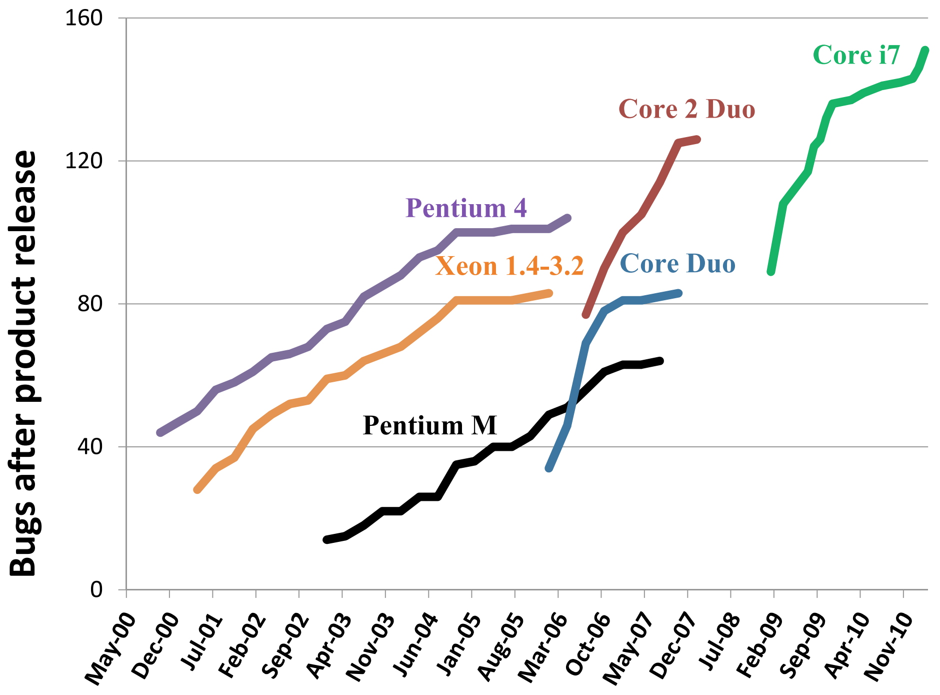


**Figure 3.1 Escaped functional bugs over time** in several Intel microprocessors, based on published processor errata. The chart shows that after product release, bugs accumulate at a high rate for a period of time, and then decrease in their rate of discovery.

Despite massive pre-silicon verification efforts, chips are still released with devastating bugs. Figure 3.1 shows the cumulative discovery of bugs over time in several Intel microprocessors. The chart is derived from published errata documents. Errors in the memory subsystem are becoming increasingly common. Memory coherence and consistency, which provide guarantees as to the order of memory operations, are significant sources of escaped bugs and are likely to become more error-prone as designs move from buses towards complex, non-deterministic interconnects. The system-level properties related to memory operation policies are difficult to verify due to the vast state space they encompass and their decentralized enforcement. As technology moves towards large CMP systems, such as the TILE64 [15] and Polaris [144] microprocessors, the verification problem worsens.

To address these shortcomings, post-silicon validation has emerged as a new complementary approach, promising to bridge the gap between failing pre-silicon verification efforts and the correctness requirements of multi-core systems. Applied to early silicon prototypes, post-silicon validation enables high coverage as a result of fast execution speeds. However, current post-silicon techniques, such as logic analyzers [149], on-chip assertions [143] and scan chains [26] are plagued by limited internal observability. This precludes existing techniques from adequately validating system-wide properties such as memory coherence and consistency, for which it is extremely difficult to detect and diagnose bugs from the system's external interface.

Regardless of massive industry efforts in pre-silicon simulation, formal verification and post-silicon validation, escaped functional bugs that manifest at runtime are a reality. As a result of the large number of interactions and the intricate communication in CMPs, errors in the communication subsystem now account for a significant portion of the reported bugs. For example, in the Core 2 Duo and Core i7, at least 10% and 13% of the design errors reported in the corresponding errata documents are associated with the communication system [68, 69], despite the CMPs having simple interconnects. As CMPs transition towards complex NoC-based interconnects, advanced router architectures, network-level interactions and concurrent communication make the interconnect highly susceptible to design errors.

Functional design errors may affect any part of the interconnect, particularly those involved in complex operations, such as virtual channel and switch allocation in NoC routers, writing and reading from a router's input buffers, as well as the routing protocol itself. Therefore, data packets sent over the interconnect may become corrupted, misrouted, or even deadlocked. Without an appropriate runtime solution to ensure that such escaped design errors do not affect the communication correctness, these issues could lead to critical loss of data and the failure of software applications or of the entire system.

Our goal is to mitigate functional bugs by learning correct design behavior during pre-silicon verification, accelerating checking during post-silicon validation, and end-to-end guarantees at runtime.

## 3.2 Understanding the Pre-Silicon Design with Inferno

The first component of our BiPeD framework is integral to the overall solution. Our goal is to define an abstraction that is intuitive for validation engineers, and useful for automated checking. Inferno [40] is a software tool that fulfills these goals, so-called because it "infers" a design's behavior. It streamlines the engineering effort dedicated to verification by presenting a technology to automatically extract the semantic protocol of any communication interface in a design from simulation data. The protocol is then summarized in a compact, abstract diagram, which retains all the key behavioral aspects observed while removing the low-level timing information. This form, called a *transaction diagram*, represents the semantic behavior of the communication interface. In addition to being useful for pre-silicon debugging, transactions and protocols can be leveraged for post-silicon validation and runtime verification, discussed in Chapter 6.

Our proposed approach greatly streamlines the resources needed for the verification and debugging of communication-centered designs by attacking the problem from several directions: i) it greatly reduces the need to rely on waveform analysis to debug a design, ii) it lowers the barrier to understanding the design's RTL implementation, and iii) it eases the adoption of formal verification methodologies by automatically generating properties related to the protocol's approved behavior.

From a structural standpoint (see Figure 3.2) Inferno takes as input a small configuration file, a simulation trace and the design under verification. The configuration file lists either a design module whose I/O interface is the target of the analysis, or the specific signals to consider. Source code of the design under verification is used only to determine the signals' directions at the interface of choice. The range of design interfaces that Inferno may consider for its analysis is very flexible, ranging from any communication interface of the design (such as a module I/O) to any custom-crafted set of signals within the design.

Inferno generates two types of directed graphs, protocol diagrams and transaction diagrams. A protocol diagram includes a vertex for each unique combination of signal values observed at the target interface. Vertices are connected by an edge if the corresponding signal combination follow each other in the simulation trace. Transaction diagrams attempt to grasp the high-level sematic behavior of the design by partitioning the simulation trace into time intervals, with each interval corresponding to a transaction (any transaction can occur multiple times in a trace). Moreover, we deploy several techniques to recognize similarities among different intervals in an attempt to reduce them to a small set of transactions.

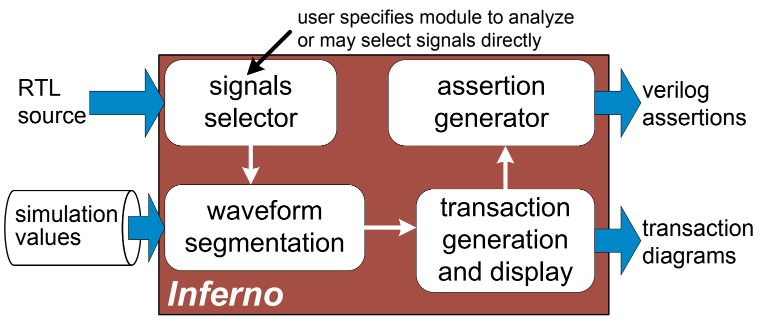


**Figure 3.2 Inferno architecture.** The user selects a design module I/O or a specific list of signals to monitor. Inferno observes the values assigned to these signals over the course of a simulation run. It then analyzes the trace, extracting a list of transactions and presenting them in the form of high-level diagrams. In addition, it generates a set of optimized assertions from the diagrams, which can be used to spot any new transaction.

### 3.2.1 Protocols and Transactions

Inferno begins its analysis by generating a protocol diagram, a time-independent graph that describes all behavior observed at the interface under analysis. Each vertex of the graph represents a unique, observed signal combination, while the edges show transitions between these signal combinations. An edge from vertex *A* to *B* indicates that at some point in the simulation, the interface signals transitioned from the values in *A* to the values in *B*. As a result of time abstraction, all vertices have implicit self-transitions. This algorithm is shown in Figure 3.3.

```
1:ProtocolExtractor (sequence) {
  create_vertex ($s_0$)
2:for i=1 to $t\_max$ {
3:   if !vertex_exists($s_i$) create_vertex ($s_i$)
4:   if ($s_i \neq s_{i-1}$) create_edge ($s_{i-1}$, $s_i$)
5:}
```

**Figure 3.3 Pseudocode for protocol diagram generation algorithm.** The algorithm generates a graph representing all behavior observed at the target interface, abstracting away time.

After generating a protocol diagram, where each new signal combination observed at the interface triggers the generation of a new vertex, tagged by a unique ID, Inferno continues with transaction extraction. The entire simulation can now be encoded as a sequence of these IDs and transactions can be extracted by analyzing this sequence. The transaction extraction algorithm breaks the sequence of IDs down into sub-segments called proto-transactions, which are refined by the algorithm until they reach their final shape as transactions.

Note that transaction diagrams are subgraphs of the protocol diagram by construction,

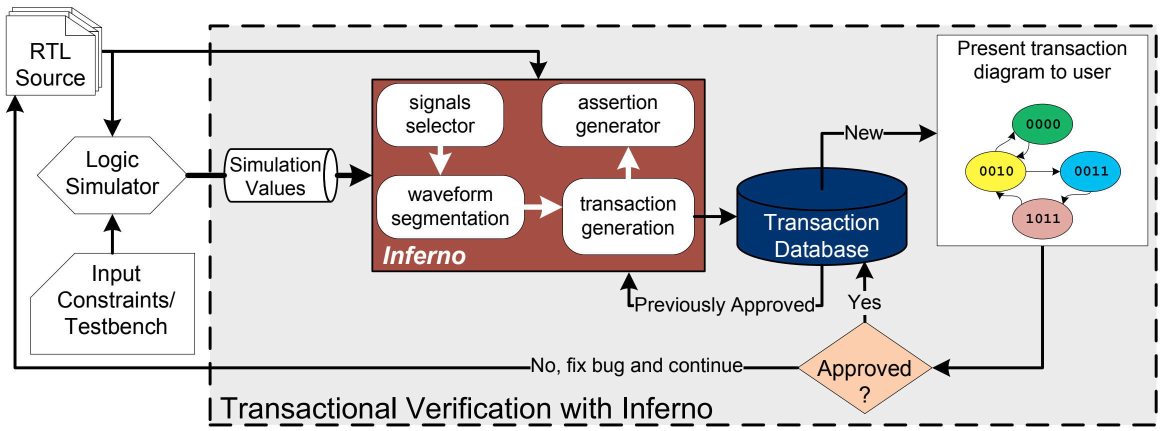


**Figure 3.4  Flowchart of Inferno's transaction extraction algorithm.** The algorithm is comprised of three major stages: boundary identification, repetition folding and boundary refinement.

since they are both extracted by folding the linear sequence of unique IDs. Both diagrams have the same semantic meaning for vertices and edges; however, the transaction extraction algorithm partitions the sequence into segments and thus transactions correspond to subgraphs of the protocol diagram. Moreover, this algorithm implements additional folding and reductions on the sequence compared to the protocol extraction algorithm. It is applied to the complete sequence of IDs and operates in three steps (see Figure 3.4):

1. Boundary identification and labeling
2. Repetition folding
3. Boundary refinement

The extraction algorithm starts with the ID sequence, partitions it into proto-transactions, and then proceeds to refine these until they are completely distilled down to what we call transactions. A proto-transaction is a segment of our initial ID sequence that will be transformed through the algorithm to obtain an instance of a transaction. The following three sections expand on each of these components; line numbers refer to the pseudocode of the algorithm, shown in Figure 3.5.

**Boundary Labeling** (Lines 2–3). The very first goal of the transaction extraction algorithm is to identify the boundaries between proto-transactions. At this stage we only perform an approximate boundary identification, which is further refined in the subsequent phases of the algorithm. Initially, the boundary marker is defined as the first repeated ID tag. This ID is identified by analyzing the trace from the beginning of the simulation and it indicates the completion of a proto-transaction. We use this ID to partition the simulation trace into multiple proto-transactions; each of them will eventually become an occurrence of a transaction. The intuition behind this heuristic approach is that the stable interface value at the end of a reset sequence almost certainly marks a proto-transaction boundary (although not necessarily the only one), and frequently it is also repeated at the completion of each proto-transaction, hence it is observed in simulation as the first repeated label.

```
 1:TransactionExtractor (sequence) {
 2: new_boundary_chars =
 3:       first_repeated_vertex(sequence)
 4: do {
 5:   boundary_chars = new_boundary_chars;
 6:   segments = split(sequence,boundary_chars)
 8:   for each segment {
 9:     sub-segs = substr(segment,length > 1 &&
10                 occurring > 1 consecutively)
12:     for each unique sub-seg {
13:       count_repetitions(sub-seqs, sub-seg)
14:       remove_all_but_first(sub-seg,segment)
15:     }
16:   }
17:   for each pair of segments (i,j) {
18:     if (j is a suffix of i)
19:       new_boundary_chars += last_char(i-j)
20:   }
21: }while(new_boundary_chars!=boundary_chars)
22:}
```

**Figure 3.5 Pseudocode for dynamic transaction extraction algorithm.** The algorithm progresses in three stages: boundary identification and labeling, repetition folding and boundary refinement.

While this is not the only viable technique to identify transaction boundaries, we found experimentally that it works well in many practical cases. We have also considered an alternative approach of setting the boundary to the label with the highest number of occurrences in the trace. Intuitively, this suggests that we should extract the highest number of transaction instances. However, we have not found this second approach to produce good results. We believe the reason may lie in the fact that the former approach creates a better correspondence with design intent. In addition, in our implementation, we provide the option for a user to specify a signal in the design whose rising or falling edge indicates the start of a transaction. Some designs have such a signal readily available, for example the request signal in an arbiter.

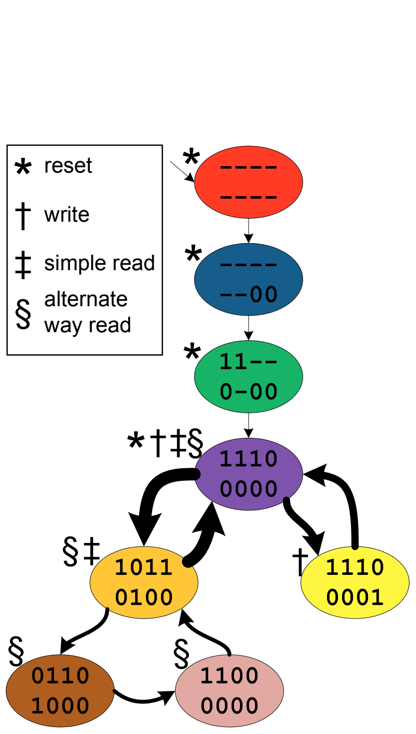
**Repetition Folding** (Lines 8–14). In this phase we identify repetitions within each proto-transaction identified during boundary labeling. A typical scenario is a burst-read transaction, where a read sequence is repeated multiple times within the same segment. Clearly all variants of burst-read operations should be matched as the same type of transaction, regardless of the specific number of reads in each proto-transaction. We achieve this by identifying repetitions within each segment and then matching segments that are identical except for the number of repetitions. At this point we have obtained a baseline set of proto-transactions.

**Boundary Refinement** (Lines 17–19). The transaction extractor algorithm described so far works well in the case where all proto-transactions do actually terminate with the same ID. When this is not the case, it fails to detect some of the boundaries, with the result that a chained sequence of several transactions may be clustered together in one single proto-transaction. The last refinement phase addresses this problem by refining the boundary set. It consists of one final pass through all the proto-transactions identified so far, checking if any proto-transaction *A* is the suffix of another proto-transaction *B*, *i.e.*, if *B* is composed of a preamble followed by *A*. In this case, we can reasonably conclude that the boundary between them constitutes a new transaction boundary. At this point, we would repeat the extraction process using the new boundaries in addition to the original one and regenerate the proto-transactions through the repetition folding phase. The process is repeated until we reach convergence. Now, all proto-transactions have been refined to final transactions. We found in practice that one refinement pass is usually sufficient: all of our testbenches converged with one single refinement step.

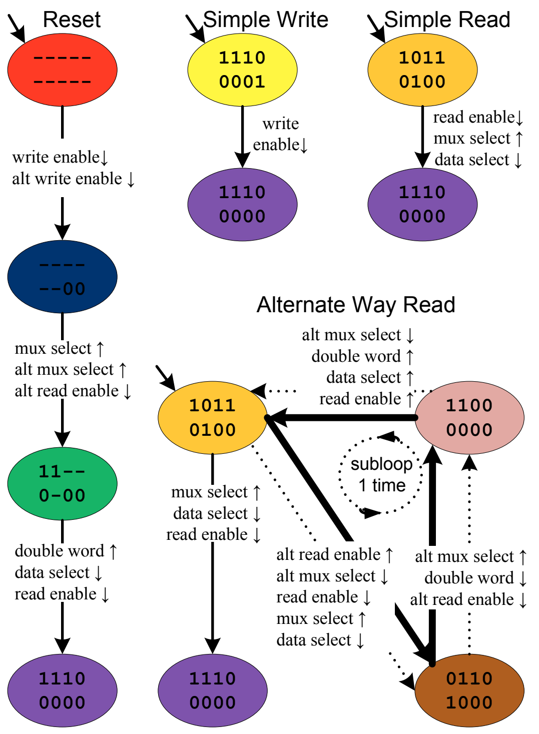
Upon completion of this process, Inferno has generated a set of diagrams, each representing a distinct transaction. For each transaction, we display the corresponding diagram, report the number of times it occurred during the simulation, and the simulation time of its first occurrence. Inferno's inference algorithm does not track the dependency of a transition from past events (that is, past signals combinations that occurred at the interface under study). In that it is similar to a Markov model that only grasps the relation between the present and the immediately previous state. We found experimentally that the simplicity of this algorithm works well in hardware designs. We believe this is due to the fact that designs frequently carry information relating to past-events in storage elements, whose values we can simply monitor with Inferno.

### 3.2.2 Evaluation Case Study: OpenSPARC T1

In this case study, we demonstrate the capabilities of Inferno on an industrial microprocessor design, OpenSPARC T1[140]. Sun Microsystems' OpenSPARC T1 is a complex processor design, which encompasses half a million lines of code (LOC). The system consists of 8 SPARC cores, each capable of executing 4 simultaneous threads. OpenSPARC T1's memory sub-system consists of level one (L1) data and instruction caches at each core connected to a unified level two (L2) cache and four main memory controllers.



**(a)** Protocol diagram



**(b)** Transaction diagrams

**Figure 3.6 Protocol and transaction diagrams from the OpenSPARC T1** data cache interface. Dashes indicate the high-Z state, occurring only in the reset sequence. Statistical line weighting on this graph indicates the most commonly exercised path, a common read, shown by heavy lines. The transaction diagrams represent a simple read, simple write, and alternate way read. Arrows next to the signal names denote rise and falling edges.

We studied the behavior of the control signals of the L1 data cache interface with Inferno. The data source was a full system simulation running the regression suite provided with the OpenSPARC release. We quickly obtained a set of four distinct transactions represented in Figure 3.6b. Upon inspection of OpenSPARC's specification manual, we determined that these precisely represented all the data cache interface's valid modes of operation. Note how each transaction in Figure 3.6b terminates with the same vertex, which is precisely the boundary vertex ID in this analysis.

In Figure 3.6a we plot the corresponding protocol diagram, where we indicate the correspondence between each transaction and its sub-graph in the protocol diagram. In addition, our visualization techniques for protocol diagrams make it immediately apparent that the simple read operation is by far the most common activity. In fact, the two thickest edges in the diagram of Figure 3.6a correspond to the edge starting a simple read transactions and the edge internal to this transaction.

Our case study of OpenSPARC T1 demonstrates that Inferno is capable of tackling large, industrial-scale designs. We could extract the functional activity of the L1 data cache interface with Inferno's high degree of automation. The abstraction made possible by transactions made the interface easy to understand and the concurrent simulation processing allowed us to monitor results without waiting for a lengthy, tedious simulation to complete.

### 3.2.3 Leveraging Transactions in Later Verification Phases

While the abstraction of transactions presents a useful way to understand complex protocols during pre-silicon verification, the information provided by transactions and protocols can be used to detect bugs during post-silicon validation and runtime verification. We discuss these opportunities in the next section.

## 3.3 Briding Pre- to Post-silicon with BiPeD

Pre-silicon verification and post-silicon validation methodologies are very different, traditionally sharing little information or hardware between them. This section applies the pre-silicon verification techniques of Inferno (Section 3.2) to learn the correct behavior of a design, and then uses this information to find bugs during post-silicon validation. During pre-silicon verification, our BiPeD framework learns the correct behavior of a design's communication patterns. In post-silicon, this knowledge is used to detect errors by means of a flexible hardware unit.

Our goal is to learn the correct behavior of a system's protocols during high-observability, low-speed pre-silicon verification, and then detect violations of these protocols during high-speed, low-observability post-silicon validation. This approach targets difficult post-silicon bugs that manifest in the inter-block interactions of a complex chip, automatically determining their time of manifestation and providing a complete and detailed set of intuitive of debugging information related to the system's activity preceding and leading up to the failure.

In the remainder of this section, we discuss the application of Inferno to aid BiPeD in learning correct design behavior, and then enforce correct protocol behavior during fast post-silicon validation.

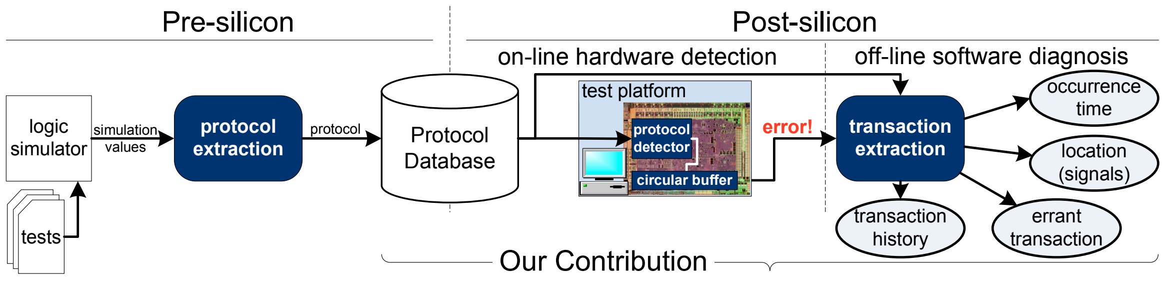


**Figure 3.7 Methodology overview.** During pre-silicon verification, full observability is leveraged to learn the protocols that define interfaces between design blocks. These protocols are then programmed into flexible hardware during post-silicon validation, where the protocol can be checked at high-speed during high-coverage tests. When a bug is detected, a recent history of observed activity is transferred off-chip for analysis by a companion software algorithm. The algorithm extracts intuitive transaction diagrams, representing the activity leading up to the bug and presents to the user a rich set of debug information.

### 3.3.1 Learning Correct Design Behavior During Pre-silicon Verification

During pre-silicon verification, BiPeD learns the semantics of a design's protocols with protocol extraction software. Later, these semantics are checked at-speed by flexible protocol detection hardware during post-silicon validation. When a check fails, the history of the activity observed on the failed interface is transferred off-chip for analysis by an off-line software algorithm. The result is a rich set of debugging information, which includes a trace of intuitive, high-level descriptions of the behavior leading up to the failure. Figure 3.7 shows an overview of this process.

Our approach leverages the abstraction techniques of Inferno (Section 3.2). Protocols describe the operation of a block or communication interface, and are represented by a graph with a vertex for each unique combination of signal values observed on the interface. There is an edge from vertex A to vertex B if there is at least one occurrence of signal combination A immediately followed by B in the simulation trace. Transaction diagrams represent the high-level semantic behavior of the design and are obtained by partitioning the simulation trace into multiple intervals, each corresponding to a transaction. Each distinct transaction typically repeats many times in a trace.

Leveraging the full observability of the design during pre-silicon verification, the protocols for a design's interfaces are generated during pre-silicon verification. First, interfaces are identified by designers: each interface is defined by a set of the design's signals, usually control signals. Passing testcases are then run on the system, generating traces for protocol extraction. The end result is a protocol representing the expected behavior of the interface, saved to a "protocol database" for use during post-silicon validation. This process is re-

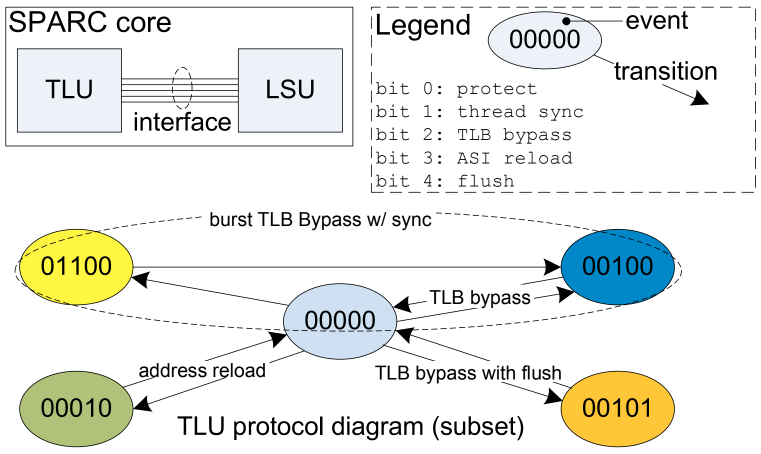


**Figure 3.8 Protocol subset of the OpenSPARC T2 TLU/LSU interface,** which defines its valid behavior. Each vertex represents an event, and each edge a transition. The bits in a vertex indicate signal values. Transactions are a subgraphs of the protocol, and are labeled.

peated for each interface selected, and it is illustrated with an example in Figure 3.8. As an alternative to automatic protocol extraction, protocols may also be specified by hand.

The example interface in Figure 3.8 is a subset of the OpenSPARC trap logic unit (TLU) interface, which regulates the communication between the TLU and Load/Store unit. Each unique set of values observed on the interface's signals constitute an event, that is a vertex; edges connect subsequent events, as shown in the protocol diagram at the bottom of the Figure.

### 3.3.2 Post-silicon Failure Detection

BiPeD leverages the abstraction of protocols for post-silicon bug diagnosis. The fast execution speeds of post-silicon validation enable high coverage, thus the protocols learned during pre-silicon verification can now be stressed with heavy testing. Our in-hardware solution monitors a number of interfaces simultaneously, confirming that the observed events and transitions conform to the protocol. When a mismatch is detected, our solution considers the past history of events and uses it to diagnose the bug, identifying the time of the failure, the errant transaction, the modules and signals involved, etc.

In order to monitor the interfaces of interest and check them against their corresponding communication protocol, the design is augmented with flexible hardware protocol detectors. During post-silicon validation, a number of protocols are programmed into "detector" hardware blocks, one block for each monitored interface. At runtime, the detector hardware units monitor the interfaces' activity to check that it conforms to the known protocol.

Figure 3.9 shows a diagram of a detector block. The detector checks all activity of its corresponding interface, by sampling all its signals at each clock cycle. These signals must be available by means of an on-chip debug infrastructure; frequently MUX selection trees are available that connect to many signals in the design. The interface signals are first routed to a content addressable memory (CAM) that matches their sampled values against known protocol events. If a matching event is found, a priority encoder converts it to an index value. Transitions are checked by consulting the transition CAM with a pair of indexes: the one from the current event and the one from the previous event – which is stored in a local register. The encodings of transitions are known by their indexes in the event CAM when the transition CAM is programmed. If either CAM fails to find a match, an error detection is flagged.

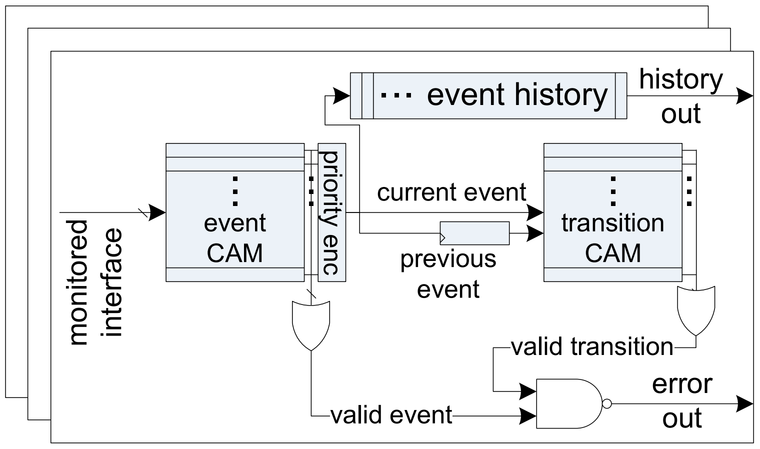


**Figure 3.9 Protocol detector hardware** units validate protocols during post-silicon validation. Each unit leverages two CAMs: one for events and one for transitions, Both pre-programmed with data collected during pre-silicon validation. The circular buffer maintains a history of events, transferred off-chip upon bug detection.

When a mismatch is detected by the hardware, execution is suspended and the event histories stored in the detection hardware's circular buffer are transferred off-chip for software analysis. The offline software analysis considers the event histories as well as the protocols as input. All event histories are examined in parallel, extracting high-level debugging information. Some debugging information is immediately available: the time at which execution was suspended, and the interface(s) that flagged the error. The time provided corresponds to bug manifestation time. In addition, we can deduce the modules and signals involved in the bug by using the protocol database, which contains the names of the signals included in each interface and their connected modules.

The algorithm then processes the contents of the circular buffer for each interface that flagged an error: events are first decoded and then reconstituted into the interface signal values that they represent. The result is a partial trace of activity observed on the flagged interface(s). At this point, BiPeD applies the Inferno transaction extraction algorithm (Section 3.2).

A number of observations make extraction on partial traces possible in our work. First, we noted that interfaces tend to return to a "stand-by" state between transactions. An example of this is the `00000` boundary in the TLU interface of Figure 3.8. We found that even with a partial trace, the stand-by event is typically the first repeated event. Thus, by using this event as stand-by we have been successful in extracting transactions in our experimental evaluation. We also explored other methods of identifying the best boundary events to separate transactions in partial traces. We considered storing the boundary events found in pre-silicon analysis along with their protocols in the protocol database. However, we found that different testcases would occasionally highlight different boundary events, and when using the union of all these events our transactions would be too fragmented. This latter approach may be useful during pre-silicon verification, when the design is changing frequently while the same testcases are re-executed. However, during post-silicon validation, many varied testcases lead to the extreme fragmentation mentioned above.

In developing BiPeD, we tried other methods of identifying the best boundary events to separate transactions. One possibility was for the protocol database to store not only events and transitions, but a set of boundary events as well. Transaction extraction would then use this pre-determined set. The problem that arose with this approach was that different testcases would sometimes yield different boundary events. When the set of all boundary events was used to extract transactions, they tended to be small and fragmented, due to large number of boundary events. While this approach may be useful during pre-silicon verification, when the design is changing, but the same testcases may be reused, it is not effective for post-silicon validation. Post-silicon validation enables the execution of many tests not run during the pre-silicon stage. Thus, we found that dynamically detecting boundary events was more effective that saving and restoring them from the protocol database.

The result of a transaction extraction on a partial trace is a sequence of high-level, intuitive transaction diagrams representing the behavior of the interface preceding the failure. The last transaction in the sequence is the erroneous transaction which caused the protocol detector to suspend execution. BiPeD marks this transaction and indicates the exact event or transition that caused the mismatch. Thus, BiPeD is able to identify the erroneous transaction, event and transition.

In order to identify candidate signals for further debugging, BiPeD uses the errant event or transition. If it was a transition that caused the mismatch, the exact signals within the interface are identified by comparing the pair of events that the transition connects. On the other hand, if an unknown event is flagged, BiPeD compares this event to the other events in the interface (from the protocol database), identifying those events which are most similar to the errant event. Furthermore, these signals are used to identify the hardware block responsible for the error.

#### BiPeD Integration

The BiPeD framework bridges pre-silicon verification with post-silicon validation. By leveraging the high-level, compact, intuitive transactions and protocols of Inferno, BiPeD is able to learn the behavior of a design's interfaces during pre-silicon verification and enforce it during post-silicon validation.

Once a bug has been detected, the exact source of the error must be identified. In the next sections, we present two solutions that complement BiPeD's detection mechanism to narrow down the source of the bug.

## 3.4 Accelerating Post-Silicon Validation with Dacota

Two common sources of errors in the memory subsystem are the coherence and consistency protocols. These protocols are responsible for enforcing the ordering among memory operations. During post-silicon validation, it is possible to validate large, distributed properties. In contrast, pre-silicon simulation does not always scale to large systems, like the memory subsystem. Thus, post-silicon validation is especially well-suited to verifying the ordering of memory operations, a difficult, and sometimes impossible, task for slow pre-silicon techniques.

This next step of our dissertation work introduces a novel solution to identify functional bugs in the memory ordering of CMPs in post-silicon validation. Our solution, called Dacota [43] (**D**ata-coloring for **CO**nsistency **T**esting and **A**nalysis), offers the benefits of high validation coverage and debugging support at a very small performance impact and near-zero area overhead. Enabled only during post-silicon validation, it incorporates a simple in-hardware activity logging mechanism that observes selected system activity during program execution. Periodically, a software-based validation algorithm examines the logs to detect violations in the ordering of memory operations, indicative of an error in memory coherence or consistency.

When Dacota is enabled, an activity logging mechanism located at each level one (L1) cache stores a compact encoding of memory accesses. The caches are temporarily reconfigured to include an *access vector* associated with each line: the access vectors contain a

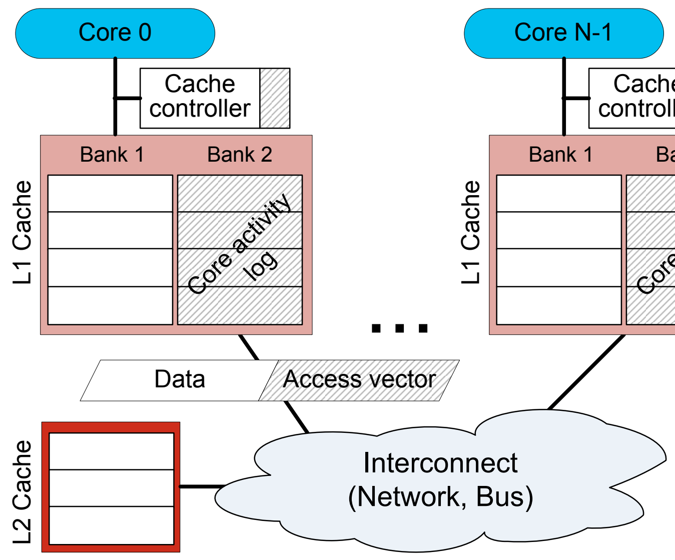


**Figure 3.10 CMP reconfiguration for Dacota validation.** Cache lines are partitioned to include an *access vector* tracking the order of memory accesses. A portion of each cache is reclaimed and used as *activity log* storage for load/store operations. Finally, the cache controllers are augmented to include supporting hardware.

counter "color" value, incremented with each store operation to the line and used to disambiguate accesses to the same cache line during execution. In addition, after each load and store, individual CMP cores log the address and color values of the access in an *activity log*, which is also maintained in the local cache. Thus, the logs record the history of memory accesses in program order for each individual core. When local cache storage is exhausted, activity logs are aggregated and validated by a software-based algorithm, leveraging the existing processor cores for computation. Validation is performed by building a graph from the activity log where vertices represent memory accesses and edges indicate the observed ordering between them. Correct memory ordering is then checked by inspecting the graph for cycles. By leveraging existing cache storage and CPU computation resources, Dacota incurs an extremely small silicon area overhead. Finally, Dacota can be completely disabled upon product shipment, leading to zero performance impact to the end user. Optionally, this approach to post-silicon validation can be re-used as a runtime, enabling it to validate the design in the final product.

### 3.4.1 Dacota Operation

Dacota's architecture is embedded in the CMP design to be validated and requires minimal hardware modifications. A schematic of its components is shown in Figure 3.10. We assume a generic CMP architecture where multiple simple processing elements (cores), each

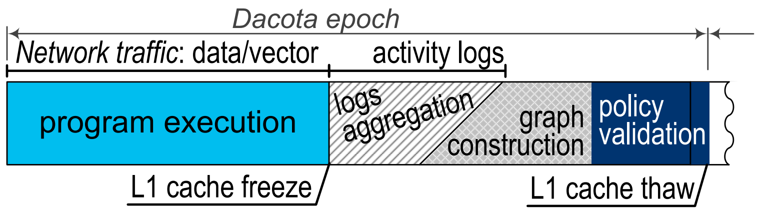


**Figure 3.11 Dacota execution flow.** When Dacota is enabled, normal benchmark execution progresses with data and access vectors transferred together and while activity is logged in the background. When log resources are exhausted, they are aggregated and analyzed by a graph-based algorithm. If an error is found, the logs are presented to the user for diagnosis, otherwise execution resumes.

with private L1 caches, are connected via an on-chip interconnect fabric to a shared L2 cache. When Dacota is enabled, the system is reconfigured so that a portion of the cache resources are reserved for Dacota's *core activity logs* (hashed blocks in the L1 caches of Figure 3.10). The portion of the cache used by Dacota is configurable, a simple implementation allocates half of the cache to core logs, and the other half to normal data storage.

Processor activity is organized into *epochs* (Figure 3.11), where program execution alternates with Dacota's checking phase, which can be divided into log aggregation, graph construction and policy validation. During normal program execution, Dacota monitors activity in the background by updating and transmitting access vectors along with data, and logging snapshots of the vectors. When log resources are exhausted, program execution stops, data in transit is allowed to reach its destination, and all data portions of the caches are frozen. All cores then drain their activity logs into a dedicated region of un-cacheable memory (*log aggregation*). Next, each core in the system builds a *consistency graph* representing the ordering of memory operations. This consistency graph is examined by the *policy validation algorithm* to expose memory ordering errors. If the analysis exposes an error, information from the activity logs can be leveraged to support subsequent diagnosis and debugging. Otherwise, activity logs and access vectors are cleared and the next epoch may begin.

When Dacota is active, each cache line is partitioned to include additional information alongside the data block, in an *access vector* that records the number and the order of store operations issued to the line. Each core modifying the data in the cache line also updates the corresponding access vector. Traveling throughout the system with its data block, the access vector records the order of store operations to cache lines. A snapshot of the access vector is also stored in the local cache's *core activity log* upon each memory operation, along with the address. These logs are later analyzed to validate the ordering of memory operations. When required to support specific consistency models (such as Weak

Consistency), Dacota may also log special memory synchronization instructions.

When logging resources are exhausted, Dacota's analysis algorithm validates the ordering of memory operations from the contents of the activity logs. The algorithm builds a consistency graph from the aggregated core activity logs; vertices represent memory accesses, while directed edges indicate operation sequencing as perceived by different cores. Analysis of the consistency graph determines if a memory ordering error has occurred with respect to the memory consistency model. Errors manifest as a loop in the graph. The analysis engine can also expose coherence violations if they are manifest in the consistency graph or through incompatible access vectors in the activity log. Dacota's analysis algorithm is executed entirely in software on the existing CPU resources of the CMP.

### 3.4.2 Activity Logging Hardware

The activity logging system in Dacota records the order of shared memory accesses observed by different cores. To this end, we maintain both an access vector attached to each cache line, as well as activity logs residing in local caches. The information in the access vectors and logs is updated concurrently with program execution and incurs no performance overhead during this phase. However, when the log storage resources are exhausted, normal execution is suspended and the logs are aggregated and analyzed by the policy validation engine.

#### Access vectors

Dacota logs the order of accesses to cache lines using a scheme based on data coloring. Each cache line is partitioned into two parts: one for program data and the other configured as an *access vector*; these travel together through the interconnect and caches of the CMP. Each core has a dedicated entry in the vector, updated when that core performs a store access to the cache line. An additional entry in the vector is reserved for a counter tracking the total number of stores to the line since the beginning of the epoch.

At the beginning of an epoch, the counter and all entries of the vector are initialized to zero. With each issued store, Dacota automatically increments the counter and copies its value to the vector entry associated with the issuing core. Updates to the counter are accomplished automatically by Dacota hardware and do not require read-modify-write operations to be issued by the CPU. A saturated counter triggers the end of the program execution phase in an epoch, a necessary measure to ensure counter uniqueness. The end result is an access vector with monotonically increasing counter values indicating the chronological order in which cores modified a line.

To manage access vector updates, we make a small addition to the cache controllers (hashed areas in Figure 3.10). This hardware component is responsible for incrementing the counter and updating the vector entries. At the cost of increased hardware complexity, it would be possible to eliminate the counter entry from the access vector and simply retrieve the counter value by extracting the highest value in the access vector entries. This alternative approach incurs higher area cost, but could be interesting for system where the resources for the access vector are extremely limited.

### Core Activity Log

While access vectors record the order of accesses to individual cache lines, *activity logs* record the order of accesses among different cache lines. Stored in a reconfigured portion of the caches, activity logs record a series of access vector snapshots. When a core issues a load or a store, Dacota copies the updated access vector to the activity log, together with the type of access (load/store) and the cache line tag. The activity log is maintained as a queue and entries are allocated in program order, but copied in order of completion, which may differ from program order. By leveraging the order and contents of the activity log, Dacota can later reconstruct the order of memory operations perceived by each core. To reconfigure Dacota's portion of the L1 cache as a queue, we augment it with a simple up-counter that cycles through the allocated ways and sets. The tag array stores a portion of the location's address, while the data block stores spill-over address bits and the instantaneous value of the access vector associated with the line. In addition, since the order of completion of memory operations may be different from program order, we add a small index table for conversion between outstanding memory accesses and entries in the cache. Because we only need to index the outstanding memory accesses, the table can be quite small.

In developing Dacota, we observed that logging each memory access led to prohibitively large storage requirements. Thus, we optimized our design to log a load access only if it triggered a cache miss, either because the data block is not cached, or because the copy in the cache is obsolete due to modification by another core. No loss in coverage is incurred by this optimization, when operating under the simple assumption that (hit) accesses to local caches are serviced correctly.

### 3.4.3 Policy Validation Algorithm

Dacota's policy validation algorithm takes the activity logs as input, builds a directed graph representing the memory operation ordering, and checks the graph for errors. The algorithm is invoked every time log resources are exhausted, and begins with aggregating the access logs. This process overlaps with the graph construction process, which may begin as soon as the first logged data is available, and it is followed by policy validation through graph analysis. To minimize the area overhead of Dacota we implemented the checking algorithm in software running on the CMP's cores.

#### Access Log Aggregation

When a core detects that its log is full or the counter of the accessed line's vector reached the maximum value, it broadcasts a message requesting validation. Upon receipt of the message, all cores are required to stop execution and complete all pending memory operations. Then, the data portions of the caches are frozen, and the activity logs are transferred to un-cacheable memory, where they are accessible by all cores for graph construction and analysis.

#### Graph Construction

After activity logs are relocated to main memory, Dacota proceeds to build a directed graph, representing the ordering of memory operations. Vertices in the graph represent unique memory accesses issued during the epoch; while edges represent the ordering constraints specific to the consistency model adopted in the system. As graph construction progresses, Dacota also conducts a coherence invariant check using the access vectors of the individual lines.

The pseudocode for the graph construction algorithm is given in Figure 3.12.a. The algorithm iterates over each core and log entry, performing a preliminary check to verify that all store operations to an individual cache line are compatible with a unique ordering of events. In other words, the algorithm checks that for a single line all cores agree on the same order for write operations. For this coherence check, we employ a data structure that maps the line's address to a complete list of stores issued to this line. Each time the line's address is encountered in a log, its access vector is compared to the list to see if there are any violations. If the entries of the vector reveal an access that was not previously observed, the ID of the core that issued it is added to the list in the proper location. After this preliminary check, we use the information in the log entry to augment the graph for

```
Graph_Construction()
  Graph G, Coherence_Order_Map M
  Activity_Log L[0..N-1]
  Foreach core c in N
    Foreach entry e in L[c]
       If Exists M[Address(e)]
          Verify_Coherence(M,e)
       Add_Coherence_Order(M,e)
       Add_Vertex(e,G)
       Edges E = Ordering_Edges(L[c],e)
       Add_Edges(E,G)
    End
  End
```

**Figure 3.12 Graph construction algorithm.** The algorithm iterates through all history logs, generating vertices and ordering edges for the graph, and checking the coherence invariant.

the consistency model. At the end, the graph is checked for loops, which are indicators of an error in memory operation ordering. Graph construction rules vary with the consistency policy.

### Graph Analysis

Consistency graphs in Dacota are constructed to reflect the order in which accesses performed by individual cores are perceived in the system. In order to find errors, Dacota searches the graphs for loops, employing a modified version of the Depth First Search (DFS) algorithm [81]. The algorithm retains the complexity of the underlying implementation of DFS [129], that is, to $O(E)$, where $E$ is the number of edges in the graph. However, to ensure maximum performance, we aggressively apply transitive closure during graph construction, thus reducing the number of edges.

Note that we do not use any additional hardware for implementing policy validation, thus dramatically reducing the silicon area overhead requirements of Dacota. This is a crucial feature distinguishing Dacota as a post-silicon solution from runtime approaches that also use graph analysis techniques. Moreover, it allows us to parallelize the analysis for common weaker consistency models where several distinct graphs must be constructed. Even for consistency policies that require the construction of a single graph, such as Sequential Consistency, Dacota exploits the cores of the CMP to parallelize the cycle detecting algorithm by starting from distinct graph vertices. To further boost the performance of Dacota during the policy validation phase, we reconfigure the storage previously occupied by the activity logs (now residing in main memory) to be used as regular cache space. When the check completes, caches are reconfigured once again to make space for the activity logs, the data cache is thawed and the next Dacota epoch begins.

#### Checking Algorithm Requirements

To minimize the area overhead of Dacota, its checking algorithm is designed to run in software, as opposed to dedicated hardware. Therefore, for our approach to be reliable, we require computational correctness from the cores, as well as their ability to access main memory. Main memory accesses use the same subsystem that Dacota is verifying, however, bugs in the memory subsystem are unlikely to cause analysis errors due to the absence of memory race conditions during Dacota analysis. The cores drain the activity logs into disjoint memory regions and then use this information without overwriting it. Moreover, since the logs are aggregated in the uncacheable memory, caches are not involved in the transfer, eliminating the chance that coherence or consistency bugs corrupt the analysis process.

### 3.4.4 Evaluation

We evaluate the efficiency of Dacota in a simulated CMP system, determining its error coverage, performance and area impact. Investigating how different configurations affect Dacota, we explore several activity log lengths and their effect on consistency graph size and policy validation algorithm runtime.

Our simulation framework was based on a CMP system modeled with the Wisconsin Multifacet GEMS memory sub-system simulator [95]. The CMP contained 16 cores, each with a 16 entry load/store buffer, 128KB L1 cache, a single 4MB L2 cache and a 4x4 on-chip mesh interconnect. The MOESI directory protocol was used as the coherence protocol, along with the Total Store Ordering consistency model. For several additional experiments, we evaluated systems with token coherence, as well as crossbar and switch-based interconnects. Dacota was implemented as a simulator plug-in, which included methods for access vector manipulation, core activity log management, the policy validation algorithm was implemented using the Boost Graph Library [129]. The runtime of the algorithms was calculated using the SimpleScalar architectural simulator [25].

The set of workloads used to evaluate Dacota was a combination of real world programs and random stimulus. We used the ten SPLASH2 benchmarks [151], considering sections of 10,000,000 instructions generated by the Virtutech Simics simulator [93]. In addition, to induce more stress on the memory subsystem, we created eight tests of directed random stimulus with varying degrees of data sharing, each containing 1,000,000 memory accesses. Three of these benchmarks used a fairly small address space that could fit into the cores' L1 caches without eviction, while the other five used significantly larger memory ranges and could not be fully contained in the L1 caches. Additionally, we used the GEMS built-in random test generator executing the "barrier" and "locks" patterns.

### Design Error Coverage

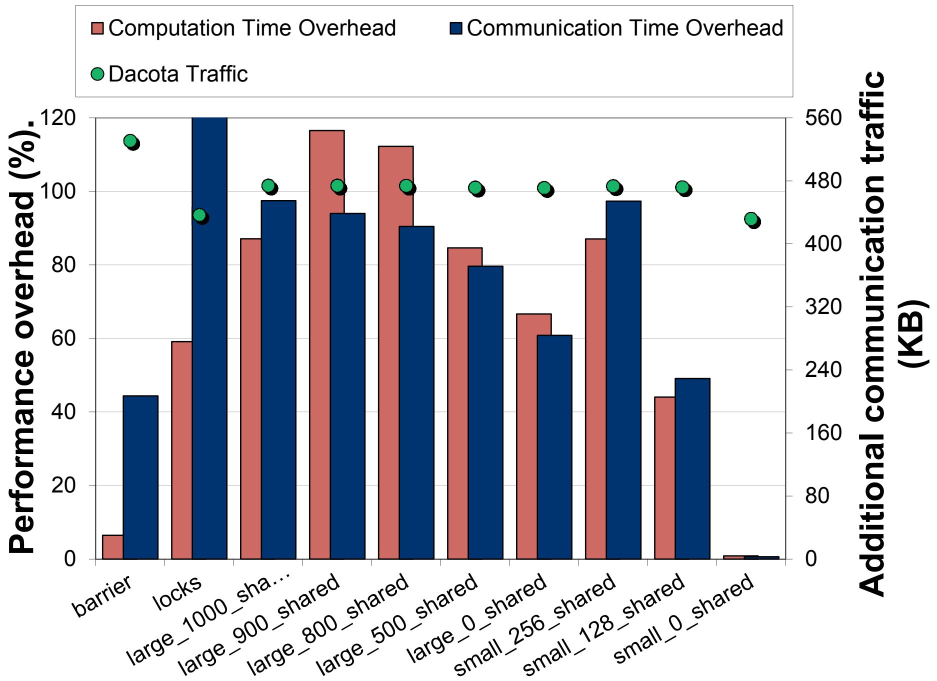
In our first experiment, we introduced eight coherence and consistency related errors into our simulation model, inspired by known issues with industrial CMPs. We then ran our SPLASH2 and random stimulus benchmarks with Dacota enabled, recording the number of cycles required to discover the bug (Table 3.1). We found that the activity logging scheme of Dacota is capable of quickly finding complex coherence and consistency bugs.

| **Bug name** | **Error description** | **Avg. cycles to expose** |
|---|---|---|
| *shared_store* | store to a shared line may not invalidate other caches | 0.252M |
| *invisible_store* | store message may not reach all cores | 1.32M |
| *store_alloc_1* | store allocation in any core may not occur properly | 1.93M |
| *store_alloc_2* | store allocation in a single core may not occur properly | 2.27M |
| *reorder_1* | invalid store reordering (all cores) | 1.38M |
| *reorder_2* | invalid store reordering (by a 1 core) | 2.82M |
| *reorder_3* | invalid store reordering (to a single address) | 2.87M |
| *reorder_4* | invalid store reordering (to a single address by a 1 core) | 5.61M |

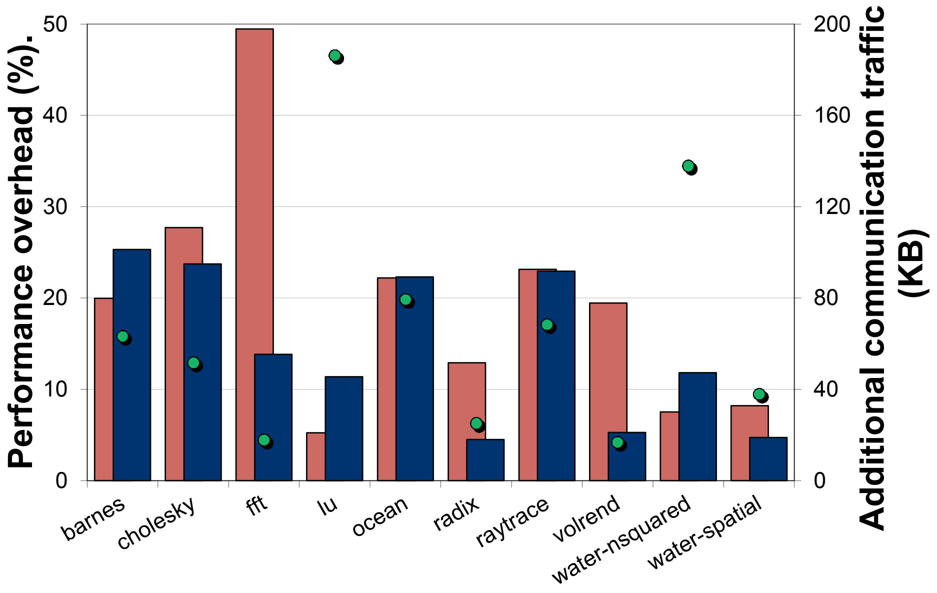
**Table 3.1 Design error coverage by Dacota.**

### Performance with Dacota

We also investigated the computation and communication overhead of Dacota relative to normal program execution time. In this study, we assumed that one half of the L1 cache (64kB) was devoted to data and associated access vectors, and the activity log was limited to 256 entries. As shown in Figure 3.13, the performance overhead was well within the acceptable range for post-silicon solutions: approximately 26% for the SPLASH2 benchmarks (for comparison, overheads of 200-300% are perfectly acceptable in this domain). The overhead for random benchmarks is somewhat higher due to the nature of these tests, which were designed specifically to stress the memory subsystem. Our implementation of Dacota serializes graph construction and log transfers: in practice they could be overlapped,



**(a)** Constrained random stimulus



**(b)** SPLASH2 benchmarks

**Figure 3.13 Dacota performance overhead** and additional traffic for an activity log size of 256 entries. **a.** Overhead for random stimulus. **b.** Overhead for SPLASH2 benchmarks.

leading a smaller aggregate overhead than the one reported in Figure 3.13. Moreover, since Dacota can be disabled upon shipment, these performance overheads are only incurred during in-house analysis of a prototype.

**Area Overhead**

To analyze the area impact of Dacota, we implemented the additional hardware required by our solution in Verilog HDL. The module included a block for updating the counter and access vector, a state machine for activity log management and an index table for conversion between program order and performance order. The module was synthesized with Synopsys Design Compiler targeting a TSMC 90nm library. The area for a control module, one of which is added to each processor core in a CMP system, is 5,216$\mu$m$^2$. For comparison, an OpenSPARC T1 [87] chip occupies approximately 378mm$^2$. Placing a Dacota module on each of the 8 cores in this design results in an overhead of 0.01%. This low overhead is largely due to Dacota's reuse of existing hardware structures, such as cache storage.

We found that Dacota is effective in detecting subtle consistency and coherence bugs, showing its promise as a solution to the problem of validating the order of memory operations in CMP systems. Furthermore, Dacota enables post-silicon debugging support, providing invaluable information to the validation team. Before shipment, the checking functionality of our solution can be disabled, completely eliminating performance degradation to the end user. Alternatively, it can double as a runtime solution.

## 3.5 Ensuring End-to-end Correctness at Runtime with SafeNoC

Despite the extensive efforts of pre-silicon and post-silicon verification, it is a challenge to ensure that the communication subsystem will operate correctly under all execution scenarios and that chip communication integrity is always preserved To address this issue, and complete our approach to functional bugs, we propose SafeNoC [1], a runtime end-to-end detection and recovery mechanism to guarantee the functional correctness of the communication fabric in CMPs. SafeNoC targets the problem of functional design errors that have escaped design-time verification of the CMP interconnect, including functional bugs in the router's architecture and in the routing protocol implementation. It ensures that all packets sent over the interconnect are correctly received at their intended destinations. To this end, we augment a baseline network-on-chip with a small and simple checker network that operates concurrently with the primary interconnect. For every packet sent over the primary network, a look-ahead signature is sent concurrently over the checker network. Each destination node checks each data packet against the pool of look-ahead signatures available at that node. If a mismatch is detected, a recovery process is initiated: all flits in-flight in the NoC are reliably transmitted through the checker network to all destination cores. There they are reassembled into the original packets via a software reconstruction algorithm leveraging the signature information.

Our detection and recovery mechanisms are independent of NoC topology, router architecture and routing protocol. Moreover, SafeNoC can detect and recover from a wide variety of interconnect design errors, by relying on a novel recovery approach, in which erroneous flits and packets are collected from the network and used to reconstruct original data packets. In contrast, traditional end-to-end detection and recovery techniques for NoCs, such as the 'retransmission-based' solution of [103], generally rely on retransmission to recover from an error, which requires large end-to-end buffering storage. When compared to such methods, SafeNoC has a lower area overhead, only 5% of system area and 27% of the NoC silicon footprint vs. 79% for a basic retransmission-based method. Moreover, SafeNoC hardware additions are simple, mostly decoupled from the existing interconnect hardware and can be formally verified to be functionally correct. Finally, our approach has minimal performance impact on the NoC operation, incurring a slight performance slowdown only when an error manifests.

The SafeNoC solution relies on adding a simple and lightweight checker network that works concurrently with the original interconnect. This network is designed to be simple enough so that it can be formally verified and guaranteed to be free of any functional bugs. As a result, it provides a reliable medium through which we implement our detection and recovery processes. In the detection phase, whenever a packet is to be sent over the primary network, a signature of that packet is computed and sent through the checker network. The signature serves as a look-ahead packet and a unique identifier of the corresponding main packet, and it is used as a basis for detecting errors in the main interconnect. When a destination receives a data packet, it recomputes its signature and compares it against previously received look-ahead signatures. If a match is not found within a certain timeout period, an error is flagged and recovery is initiated. During the recovery phase, in-flight flits and packets are recovered from the network, and reliably transmitted through the checker network to all destinations. Any destination that has a mismatched signature, runs a software-based reconstruction algorithm, in which it uses the recovered flits to reconstruct the original data packets, so that they match their corresponding signature. Figure 3.14 shows a baseline CMP interconnect overlayed with our checker network. Both the checker router and the NoC router connect to the network interface, to which we also add two signature calculation units. Some additions to the primary NoC routers are also required for recovering in-flight flits. Once all flits have been recovered, the microprocessor cores run a novel software algorithm to reconstruct the original packets.

SafeNoC is not limited to a particular interconnect topology or router architecture. For

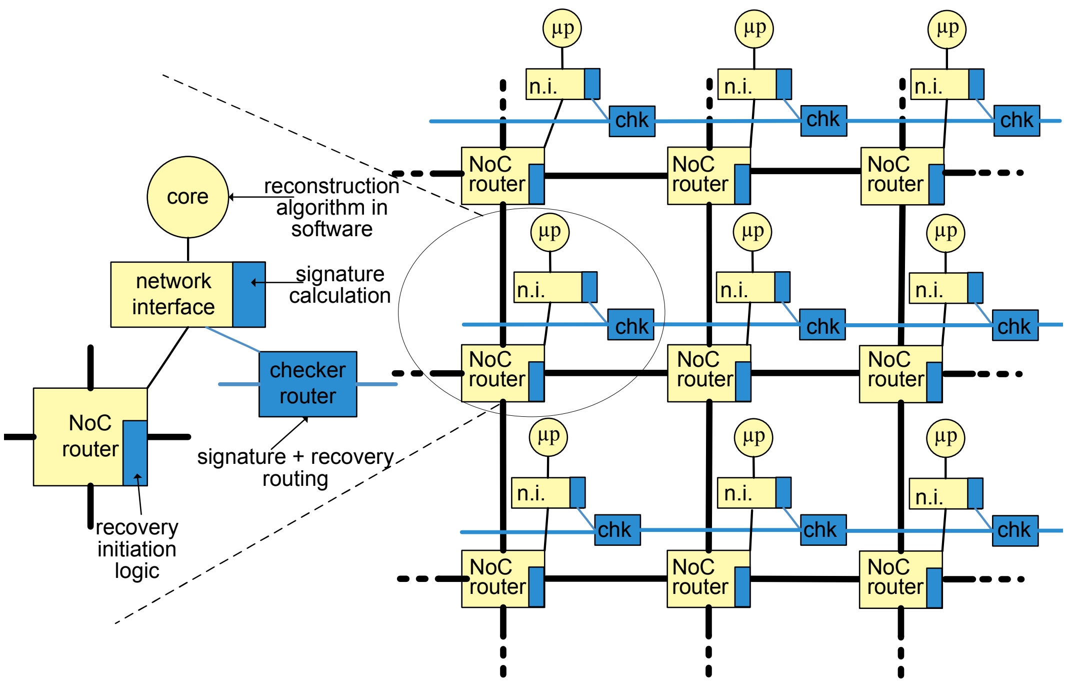

core
reconstruction algorithm in software
network interface
signature calculation
checker router
NoC router
signature + recovery routing
recovery initiation logic
µp
n.i.
chk
NoC router

**Figure 3.14 High-level overview of SafeNoC.** SafeNoC augments the original interconnect with a lightweight checker network. For every data packet sent on the primary network, a look-ahead signature is routed through the checker network. Any mismatch between a received packet's computed signature and its look-ahead signature flags an error and triggers recovery.

our experimental evaluation, we chose a general baseline router architecture that is input-queued and uses virtual-channels and wormhole routing. We also assume the data flit width to be 64-bits and that the primary routers have built-in error-correcting code (ECC) functionalities to protect against bit-level data corruption.

### 3.5.1 Error Detection Hardware

SafeNoC uses a checker network for error detection. The checker network is designed to have three main properties: i) it should be small, so to incur low area overhead. ii) it should have a simple router architecture, topology and routing algorithm, so that its design is simple and can be formally verified. iii) Finally, it should have low latency, so that it can deliver look-ahead signatures before actual data packets arrive through the primary network. We chose a ring topology for the checker network because of its simplicity and small area overhead. In addition, since the checker network transmits look-ahead signatures of fixed size (32-bits in our case, as we discuss in the following section, we can tailor the channel bandwidth accordingly, achieving both efficient bandwidth utilization and area savings. Note that having a checker network that meets the properties mentioned above is not a strict requirement for the correctness of the SafeNoC solution. A checker network that consistently lags behind the primary interconnect would introduce an additional performance penalty but would not prevent detection and recovery. Finally, to optimize the performance of the checker network, we leverage a simple, single-cycle latency, packet-switched router, based on the solution proposed in [78].

Each destination router maintains a timeout counter for every look-ahead packet it receives. The counter is incremented at every cycle until the data packet has been received and its signature is re-computed. If the signature matches any of the look-ahead signatures, then this packet is considered to have arrived correctly. However, if the counter exceeds the timeout and there is no match, an error is flagged. In such situation, either the expected packet may not have been delivered, or the network may have delivered an altered packet. In either case, SafeNoC can detect this mismatch and trigger the recovery process.

### 3.5.2 Recovery Algorithm

When an error is detected, the interconnect enters a recovery phase, consisting of five steps: *network drain*, *packet recovery*, another *network drain*, then *flit recovery* and *packet reconstruction*.

In the first step, a *network drain* phase is initiated, during which the network is forced to drain its in-flight packets for a preset amount of time. As shown in Figure 3.15.a), during this phase new packets are not injected into the network, while in-flight packets continue moving towards their destinations. If those packets are error-free, they will match their signatures and will be ejected from the network. As a result, this draining stage clears the network from in-flight traffic, except for packets and flits that are problematic.

The network then enters *packet recovery*, where we try to recover packets that are deadlocked within the network. We use a token-based protocol, in which a token circulates through the checker network, and primary interconnect routers can operate only when they hold the token. During this phase, primary routers remain active, except for all the virtual channel allocation functionalities, so that they are prevented from processing new data packets. If there is a deadlock in the network, then there is at least one packet in a router that is blocked waiting for allocation. Therefore, when a router receives the token, it checks its input buffers to determine if there is such a packet, in which case it is retrieved and sent over the checker network, as shown in Figure 3.15.b). Since all other router functionalities are still active, the entire packet can be drained and is then transmitted over the checker network to its destination. Once the token has circulated through all primary routers, they resume their full functionality and the entire network enters the second *net-*

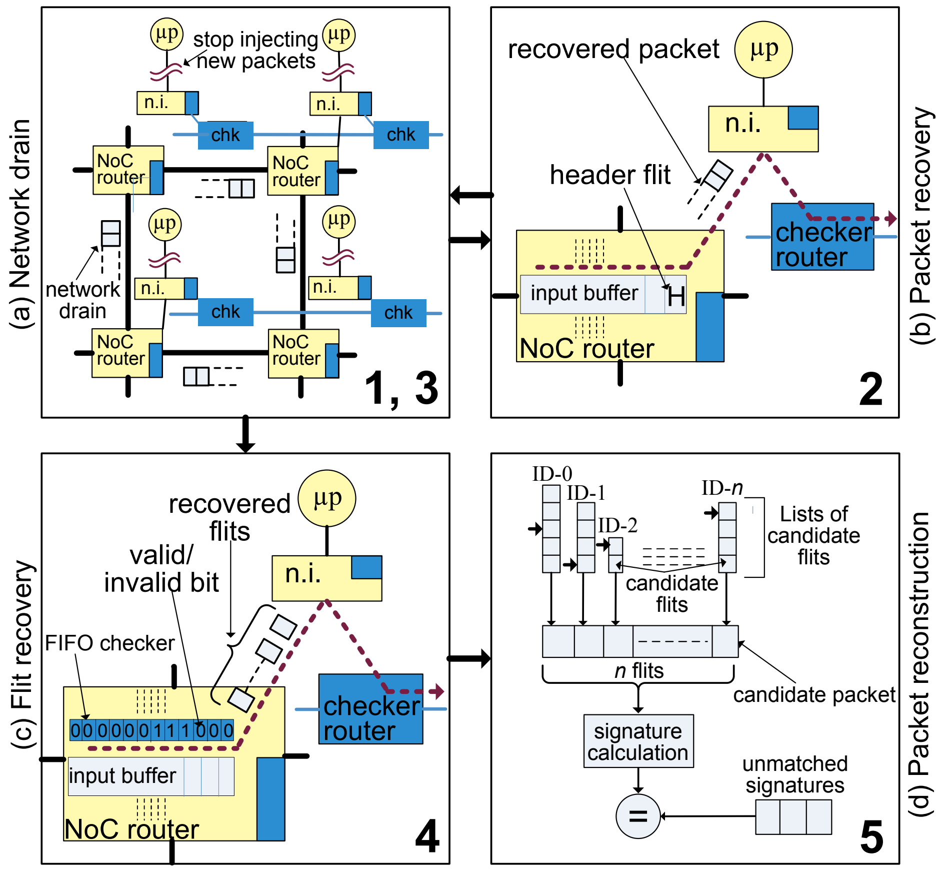


**Figure 3.15 SafeNoC recovery process.** Recovery proceeds in five steps with network draining occuring twice. The last step is executed in software, while the others are implemented in hardware.

*work drain* phase. If the previous phase has recovered packets involved in deadlocks, then removing them breaks the deadlock cycles and the remaining packets would automatically drain from the network during this second network drain. As a result, they would no longer require reconstruction at their respective destinations, which ultimately greatly reduces the effort of packet reconstruction.

In the next step, *flit recovery*, we recover stray flits from the network. A flit is considered stray if it is stuck in a router buffer or if it is delivered to the wrong destination. All stray flits are candidates for the reconstruction process. Figure 3.15.c) illustrates this phase: we added a FIFO checker to every input buffer of each router, in order to identify valid stray flits. The FIFO checker has 1-bit entries and its own read and write pointers that follow those of the input buffer. A write to the input buffer changes a corresponding entry in the FIFO checker to a valid entry and a read invalidates it. Using the same token-based protocol as in the previous phase, a router holding the token examines its FIFO checkers for valid entries. If any exist, the corresponding flits are transmitted over the checker network to all destinations in the network. As for stray flits at the network interface buffers, their presence in the buffers at this point of the recovery process indicates that they have not matched any signatures, thus they are also candidates for reconstruction, and they are also circulated over the checker network.

During the last phase, *packet reconstruction*, the processor cores that have flagged an error run a software algorithm to reconstruct the original packets using the flits collected in the previous steps (Figure 3.15.d). Candidate flits are organized in separate groups, one for each flit ID, and an index is maintained for each group to indicate which flits have already been considered. For each flit ID, we choose a candidate and add it to the set of current candidates. The current candidates are then assembled into a new packet and its signature is computed. If the signature matches any of the remaining look-ahead signatures, then this packet's reconstruction is deemed sucessful, the packet is delivered to the application and all its flits are removed from the candidate groups. If a match cannot be found, the process is repeated, generating new sets of candidates, until all possible combinations have been tried. The algorithm ends when all look-ahead signatures have been matched.

### 3.5.3 Evaluation

To evaluate SafeNoC, we modeled a CMP system in Verilog HDL and with a cycle-accurate C++ simulator. Using the hardware implementation, we formally verified the portion of the system involved in recovery, ensuring that it operates correctly. We also analyzed the area overhead of the SafeNoC solution, synthesizing the Verilog design with a 130nm target library. The impact of recovery on performance was evaluated using the C++ simulator modeling a variety of functional bugs in the baseline system. The model was simulated with two different types of workloads: directed random traffic (uniform, transpose and bit complement), as well as application benchmarks from the PARSEC suite [18].

Both the C++ and Verilog experimental setups model the same baseline system, based on the Booksim [37] simulator. The main network, an 8x8 mesh using XY routing, was augmented with a ring checker network. The main NoC routers are based on the input-queued VC router of [37], with 5 ports , 4 pipeline stages, 2 virtual channels, and 8 flit buffers. Data packets consist of 16 flits of 75 bits each, including ECC and flit IDs. We also integrated SimpleScalar [25] in our architectural simulation to estimate the reconstruction algorithm's execution time.

To analyze SafeNoC's performance impact as well as its ability to detect and recover from various types of design errors, we injected 11 different design bugs into our C++ implementation of SafeNoC, as described in Table 3.2.

In Figure 3.16, we analyze SafeNoC's recovery time by bug. The reconstruction time

| Bug name | Bug description |
|---|---|
| dup_flit | a flit is duplicated within a packet |
| misrte_1flit | a flit is misrouted to a random destination |
| misrte_3flit | 3 flits of a packet are misrouted to a random destination |
| misrte_1pkt | a packet is misrouted to a random destination |
| misrte_2pkt | 2 packets are misrouted to random destinations |
| misrte_flit_pkt | a packet is misrouted, another packet's flits are misrouted |
| dup_pkt | a packet is duplicated |
| dup_misrte_pkt | a packet is duplicated and one copy is misrouted |
| reorder_flit | flits within packet are reordered |
| deadlock | some packets are deadlocked in the network |
| livelock | some packets are in a livelock cycle in the network |

**Table 3.2 Functional bugs injected in SafeNoC.**

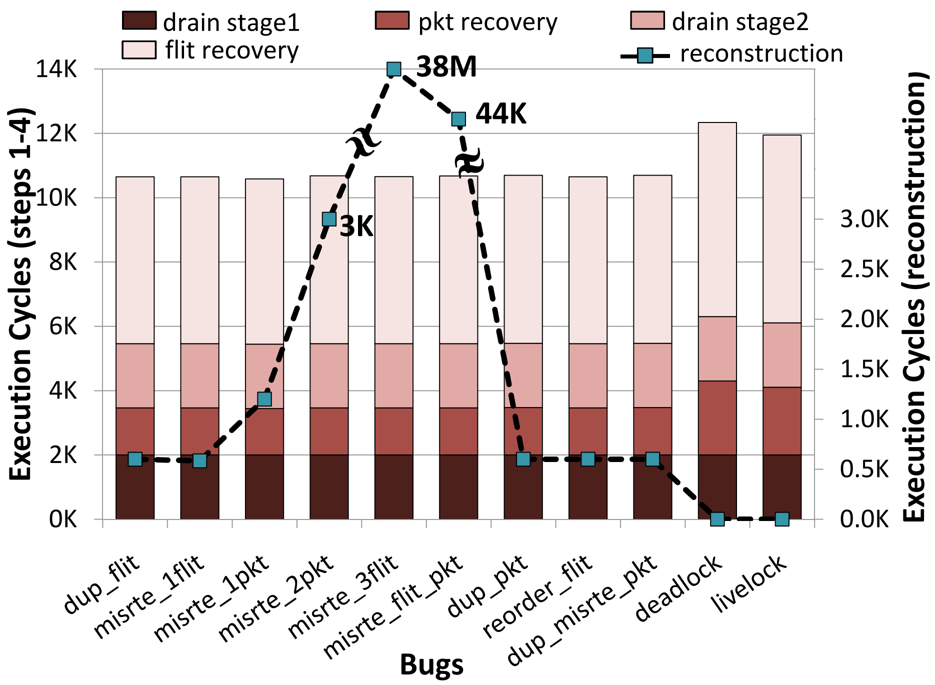


**Figure 3.16 SafeNoC recovery time by bug.** Execution cycles for the first 4 steps of recovery (bars-left axis) and for packet reconstruction (line-right axis).

varies widely, depending on the severity of the bug and the number of flits and packets it affects. For example, bug *misrte_1flit* mixes one packet's flit among the flits of another packet. As a result, the reconstruction algorithm has two candidate flits in the system and it requires only 1,200 cycles to complete. At the opposite end, bug *misrte_2pkt* affects 32 flits in 2 different packets. Therefore, the reconstruction algorithm must consider two candidate flits for each position within the packet, requiring up to 38M execution cycles to complete. On the other hand, packet recovery time is constant for almost all bugs, at 1,473 cycles, required for the token to traverse all routers, with the exception of *deadlock* and *livelock*, where entire packets are retrieved from the primary network and transmitted over the checker network. For these two bugs, packet recovery requires 2,900 cycles and salvages 90 packets from the network on average. Once these packets are recovered, they do not need reconstruction, thus the corresponding reconstruction time is 0. Finally, the flit recovery time depends on the severity of the design error. The more packets are affected by the error, the more stray flits are left in the primary network, and the more must be recovered. Thus, on average, SafeNoC requires between 11K to 38M cycles to recover the system from a bug, assuming uniform clock domains on cores and NoC.

#### Area Overhead

We evaluated the area overhead of SafeNoC, and our results indicated that SafeNoC leads to a 27% silicon area overhead with an 8x8 mesh primary network, which corresponds to a 5% area overhead over a complete CMP with 64 SPARC cores and the same baseline 8x8 interconnect. We compared this area overhead to a mainstream end-to-end, acknowledgement-based error recovery scheme as in [103]. Area overhead in these systems is primarily due to large data buffers needed to store the packets in-transit. We estimated the size of these buffers by monitoring the number of packets at each source waiting for acknowledgment. For a 8x8 primary network with 16 flit data packets, up to 7 data packets can be awaiting acknowledgments at a single source and the retransmission-based system incurs an area overhead of 79% over the baseline network.

### 3.5.4 BiPeD/SafeNoC Integration

Functional bugs can be addressed at runtime with BiPeD when it is coupled with the SafeNoC solution (Section 3.5). SafeNoC provides an end-to-end guarantee for the functional correctness of the communication fabric in a CMP. It augments a baseline network-on-chip with a small and simple checker network that operates concurrently with the primary interconnect. For every packet sent over the primary network, a look-ahead signature is sent concurrently over the checker network. Each destination node checks each data packet against the look-ahead signatures, and a recovery process is initiated if a mismatch is detected. While SafeNoC is effective in reassembling packets packets that were subject to many types of errors, it does not handle the case of silently dropped packets. BiPeD's flexible hardware can be used to detect and avoid the occurrence of dropped packets, enabling an even more robust runtime approach.

In order to detect and avoid dropped packets, the protocol detector is programmed with a runtime assertion ensuring that packets leaving FIFO storage are stored into the next FIFO. As long as this holds, packets will not be lost. The signals needed to observe this property are the FIFO control signals, the output port control, and the FIFO full signal from the neighboring routers. The key idea is to ensure that a flit leaving the current router's FIFO will be sent through an output port to a neighboring router with available FIFO storage.

## 3.6 Summary

This chapter has presented a framework for synergizing the phases of verification to address functional bugs. We explored a number of solutions to address functional bugs that fit within this framework. Functional bugs, which may be present beginning with early models, can be addressed during pre-silicon, post-silicon, or runtime verification.

For **pre-silicon** verification, we proposed Inferno, a software tool that operates on a logic simulation trace and automatically extracts transactions, that is, high level descriptions of a design's behavior. Transactions are presented to the user through simple and intuitive diagrams for which we have developed a number of specialized visualization enhancements. Complex, repetitive design simulations are distilled to a compact set of transactions which describe the semantic behavior of the system in a compact, high level format. This enables a closed-loop pre-silicon verification flow, as well as providing a set of protocols to later verification stages.

BiPeD **bridges pre-silicon verification to post-silicon validation**, leveraging the protocols extracted by Inferno to learn the correct behavior of a designs communication patterns during pre-silicon verification. During post-silicon validation, this knowledge is used to detect errors by means of a flexible hardware unit. When an error is detected, bug reproduction is not necessary: a diagnosis software algorithm analyzes information stored in the hardware unit to provide a wide range of debugging information.

Next, to narrow down **post-silicon** functional bugs, we presented Dacota, which addresses a common problem in multi-core systems: validation of memory ordering. When enabled by the verification team, Dacota stores sequence information about issued memory operations, periodically aggregating this information to perform a software-based policy validation. The validation algorithm is implemented purely in software to minimize the area impact of our solution and executes on existing processor resources. Leveraging approximately 6 orders of magnitude performance advantage over pre-silicon simulation, Dacota's post-silicon approach is able to offer significantly higher coverage compared to pre-silicon approaches. We found that Dacota is effective in detecting subtle consistency and coherence bugs, showing its promise as a solution to the problem of validating the order of memory operations in CMP systems. Furthermore, Dacota enables post-silicon debugging support, providing invaluable information to the validation team. Before shipment, the checking functionality of our solution can be disabled, completely eliminating performance degradation to the end user.

Finally, **runtime** verification of functional bugs was addressed with SafeNoC, an end-to-end error detection and recovery technique to guarantee the functional correctness of CMP interconnects. SafeNoC integrates with flexible BiPeD hardware. It augments the interconnect with a lightweight and simple checker network and it detects functional errors by comparing the signature of every received data packet with its look-ahead signature that was delivered through the checker network. In case of mismatches, we use a novel recovery approach during which blocked packets and stray flits are collected from the primary network and are distributed over the checker network to all processor cores, where our reconstruction algorithm reassembles them. SafeNoC can detect and recover from a broad range of functional design errors, while incurring a low performance impact, requiring between 11K and 39M execution cycles to recover from an error.

Taken together, this set of solutions enables a robust resistance to functional bugs in the communication subsystem of a multi-core design.

# Chapter 4
# Addressing Electrical Failures

While functional bugs are present in all verification phases, electrical failures begin to appear during post-silicon debugging, with the first silicon prototypes. Since circuit behavior cannot be accurately predicted with pre-silicon tools, electrical failures must be addressed during post-silicon validation and runtime verification. Ideally, electrical failures are caught during post-silicon validation, before a product ships.

In this chapter, we first provide an overview of electrical failures, their causes, and opportunities to mitigate them, particularly during post-silicon validation (Section 4.1). We leverage the BiPeD framework again, this time for the detection of post-silicon electrical failures. Discussed in Section 4.2, BiPeD can detect failures at full speed on post-silicon prototypes. Following failure detection, we outline a complementary solution for narrowing down the location of these difficult failures during post-silicon validation (Section 4.3). Section 4.4 summarizes our approach to electrical failures.

## 4.1 Electrical Failures

Diagnosing and debugging electrical failures in large, complex modern digital designs is a difficult task. Unlike functional bugs, where a circuit fails to operate properly under any environmental condition, electrical failures result in circuit failures only under certain conditions. For example, operating conditions such as voltage, temperature and frequency may change the behavior of the failure, or simply preclude its manifestation [73, 74].

Electrical failures are often intermittent in nature, and thus are difficult to localize during post-silicon validation. For this reason, implicating an escaped error as having an electrical root cause is problematic. Thus, electrical failures remain largely an internal problem for chip manufacturers. However, they represent a critical challenge during post-silicon validation due to their intermittent nature [112]. Electrical failures contribute to the increasingly large role of post-silicon validation, where failures are located on silicon

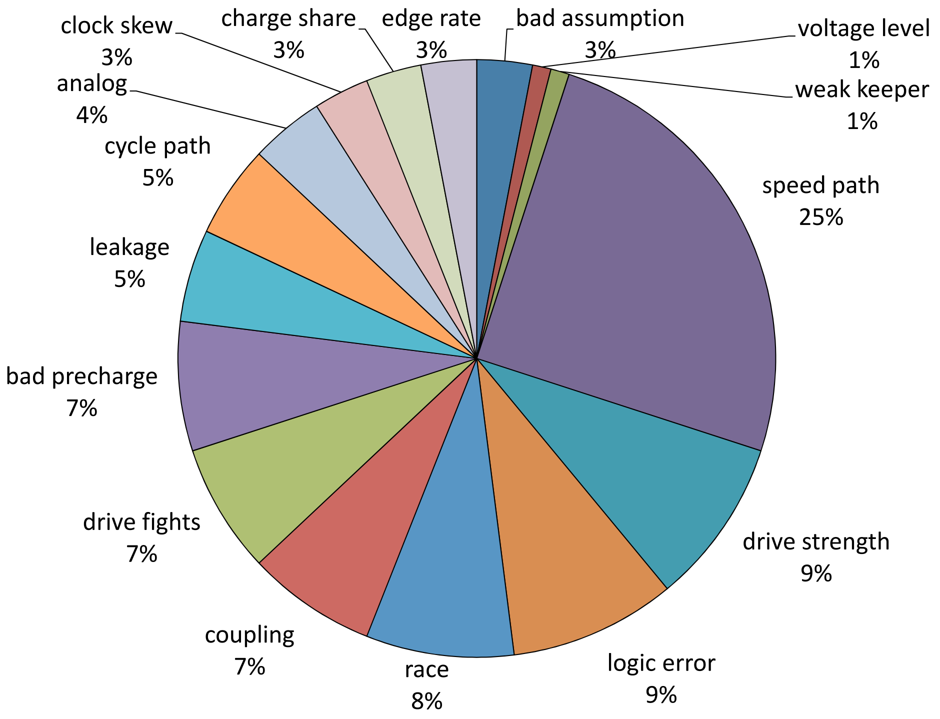


**Figure 4.1 Electrical failure root causes** for a typical industrial microprocessor during post-silicon validation, as reported by Josephson [73]. While these failures are the result of a variety of root causes, they share the difficulty of locating the root cause. This is often due to the intermittent nature of post-silicon electrical failures.

prototypes. First silicon is rarely released as a final product, indicating the large number of difficult failures discovered during post-silicon validation [73].

The root cause of an electrical failure may be due do a number of reasons. Figure 4.1 shows the distribution of root causes of electrical failures during post-silicon validation of an industrial microprocessor [73]. The distribution of failure mechanisms is fairly even, with a large contribution by speed paths. Failures on speed paths, those circuits in the chip that limit its speed, comprise 25% of the root causes. While failures may have different root causes, the end goal of post-silicon validation is to identify the exact signal or signals that are at fault.

Shmoo plots are a common way to begin debugging electrical failures, as discussed in Section 2.2. A shmoo plot attempts to find a relationship between the failure and chip operating conditions, for example, voltage and clock period. The plots show the pass/fail status of a test under different conditions. While these plots help identify possible relationships between operating conditions and error occurrence, much manual effort is still required to debug the root cause of the failure.

Real silicon lacks observability, controllability and deterministic repeatability. As a result, some tests may produce the same outcome over multiple executions, due to the interaction of asynchronous clock domains and varying environmental and electrical conditions. *Bugs that manifest inconsistently over repeated executions of a same test are particularly difficult to diagnose.* Furthermore, the number of observable signals in post-silicon is extremely limited, and transferring observed signal values off-chip is time-consuming. This work addresses precisely this post-silicon validation platform and focuses on the localization of these difficult, inconsistent bugs to ease their debugging.

During post-silicon validation, tests are executed directly on silicon prototypes. A test failure can be due to complex functional errors that escaped pre-silicon verification, electrical failures at the circuit level, and even manufacturing faults that escaped testing. The failed test must be re-run by validation engineers on a post-silicon validation hardware platform with minimal debug support. Post-silicon failure diagnosis is notoriously difficult, especially when tests do not fail consistently over multiple runs. The limited observability and controllability characteristics of this environment further exacerbate this challenge, making post-silicon diagnosis one of the most challenging tasks of the entire validation effort.

Electrical failures that slip through post-silicon validation must be covered at runtime. At runtime, many electrical failures can be observed in the same manner as transistor faults: a decaying circuit eventually fails to meet timing and causes incorrect values to be latched into internal state. This leads to opportunities for multi-use solutions that cover both electrical failures and transistor faults, discussed in Chapter 5.

## 4.2 Detecting Failures with BiPeD

The BiPeD framework is applied to electrical failures by leveraging its protocol detectors. This is similar BiPeD's application to functional bugs, which was discussed in Section 3.3. The goal of BiPeD is to learn the correct behavior of a system's protocols during high-observability, low-speed pre-silicon verification, and then detect violations of these protocols during high-speed, low-observability post-silicon validation. It targets difficult post-silicon bugs that manifest in the inter-block interactions of a complex chip, automatically determining their time of manifestation and providing a sequence of activity leading up to the failure. During post-silicon validation, these protocols are loaded into a small and flexible hardware unit, which monitors the interface(s) at runtime on the silicon prototype. When an error manifests, the hardware detector provides the recent history of the protocol-level activity observed on the interface that flagged the bug to a companion software algorithm. This, in turn, organizes the data as a series of intuitive transactions representing the interface's activity leading up to the failure.

BiPeD eases the debugging process by locating the bug manifestation time and location, tolerating noisy, non-deterministic post-silicon environments without requiring failure reproduction. It provides debugging information that includes the events and transactions in a history of recent activity. Additionally, BiPeD incurs zero performance overhead during regular post-silicon validation, and requiring off-chip data transfer only at the occurrence of a bug.

In the context of electrical failures, BiPeD hardware is programmed with the protocols learned during pre-silicon verification. Next, high-speed, large-volume post-silicon testing begins, with the BiPeD checkers monitoring critical communication interfaces. When a failure is detected, the resulting transactions are used as a starting point for more detailed debugging. In the next section, we present a complementary solution that narrows down an electrical failure to the exact root signal(s) and cycle(s).

## 4.3 Diagnosing Failures with BPS

When a post-silicon failure is detected by BiPeD's flexible protocol detectors (Section 4.2), the failure must then by narrowed down to the root cause signal(s). In this section, we outline an automated debugging approach called BPS [42], "Bug Positioning System." BPS leverages a statistical approach to address the most challenging post-silicon bugs, those that do not manifest consistently over multiple runs of a same test, by localizing them in space (design region) and time (of bug manifestation). BPS leverages existing on-chip trace buffers or a lightweight custom hardware component to record a compact encoding of observed signal activity over multiple runs of the same test. Some test runs may fail, while others may pass, leading to different activity observations. In addition, observations may be affected by variations introduced by the operating environment – both system-level activity and environmental effects. Finally, a post-analysis software algorithm leverages a statistical approach to discern the time and location of the bug manifestation.

Overall, BPS eases debugging in post-silicon validation by localizing inconsistent bugs in time and space, often to the exact problem signal, thus reducing the engineering effort to root-cause and debug the most difficult failures. It targets a wide range of failures, from functional, to electrical, to manufacturing defects that escaped testing. Additionally, BPS tolerates non-repeatable executions of the same test, a characteristic of the post-silicon environment, and thus not part of any mature pre-silicon methodology. It does not require any a-priori knowledge of the design or failures. Finally, BPS provides a scalable solution with

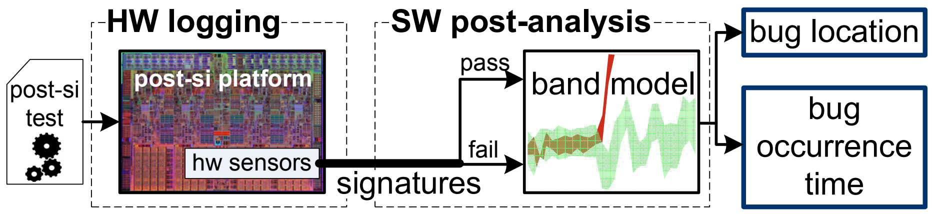


**Figure 4.2 BPS operation.** BPS operates in two phases: first, hardware sensors collect compact encodings of signal activity on the post-silicon platform for a number of executions of the same test: some may pass, while others fail. These observations are then analyzed by post-analysis software, which locates functional, electrical and manufacturing failures in time and space.

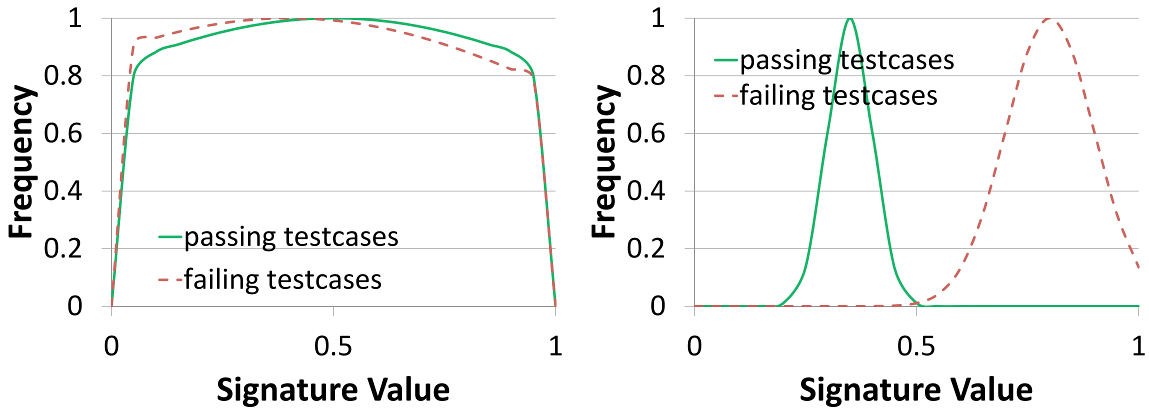
minimal engineering effort, able to handle the complexity of full chip integration typical of post-silicon validation, while minimizing off-chip data transfer through the use of compact encodings of signal activity.

In addition to electrical failures, BPS can diagnose the time and location of functional and manufacturing bugs during post-silicon validation; in particular, those bugs that manifest through inconsistent test outcomes. In these situations, the same post-silicon test may pass for some of its executions and fail other times, due to asynchronous events or electrical and environmental variations on-chip.

To locate these difficult bugs, BPS leverages a two-part approach: logging compact observations of signal activity with an on-chip hardware component, followed by an off-chip software post-analysis. The compact size of observations produced by the hardware are essential for minimizing expensive off-chip data transfers. These signal observations are gathered and reduced to a compact encoding for a number of executions of the same test, some passing and some failing, but usually all slightly different. Finally, the collected data is analyzed by the BPS post-analysis software, leveraging a statistical approach that is insensitive to the natural variations over several executions, but it is capable of detecting the more dramatic differences in signal activity typically caused by bugs. The result is the localization of the bug through the reporting of an approximate clock cycle and the set of signals most closely related to the error.

### 4.3.1 BPS Hardware

The hardware component of BPS logs *signatures*, compact encodings of observed activity on a set of target signals, which are later used by BPS' post-analysis software to locate failures. Signals available for observation are selected at design time, and the most effective choices are typically control signals. Signatures are recorded at regular intervals, called *windows*, and stored in an on-chip buffer. Windows can range in length from hundreds to millions of cycles, and are later used to determine the occurrence time of a bug. Logged data is periodically transferred off-chip for analysis by the BPS software. Simple signatures can often be collected using existing debug infrastructures, such as on-chip logic analyzers [149], flexible event counters [2, 12, 117] or performance counters.



**(a)** Poor statistical separation **(b)** Good statistical separation

**Figure 4.3 Comparison of typical distributions with different signatures.** A signature that exhibits a wide, evenly distributed output (a) does not allow BPS to differentiate correct behavior from incorrect. In contrast, signatures that exhibit good separation (b) are effective.

An ideal signature is compact for dense storage and fast transfer, and represents a high-level view of the observed activity. Furthermore, the signature must exhibit a statistical separation between passing and failing cases, as shown in Figure 4.3. In order to differentiate erroneous behavior from correct behavior, BPS characterizes activity using distributions of signatures. Throughout the development process of BPS, we considered a variety of signatures, including various codes and counting schemes. We found that many traditional codes, such as cyclic, hamming distance and multiple input shift registers (MISR), exhibited a wide range of output (Figure 4.3a) and are very susceptible to noise: small variations among executions led to severe variations in the signature value. Thus, it is difficult to distinguish erroneous from correct behavior with these signatures. This led us to counting schemes, where the amplitude of changes in signal activity leads to approximately proportional changes in signature values. The result is a discernible difference in the distribution of signatures for passing vs. failing testcases (Figure 4.3b) and less vulnerability to noise.

Signatures based on counting schemes include toggle count, time at one and time at zero. We chose a variation of time at one for BPS: the probability of a signal being at one during a time interval (window), $P(time@1)$. This signature is compact, simple and encodes notions of switching activity, as well as timing. By contrast, toggle count expresses the logical activity of the signal, but it does not provide any temporal information. Figure 4.4 shows an on-chip hardware sensor implementation for measuring $P(time@1)$. Signals from the design are connected to counters via muxes, allowing the selection of a subset of the signals to be monitored. Note that we can calculate this signature by simply counting the number of cycles when a signal is at 1 and normalizing to the window length. Furthermore, we noted experimentally that approximately 9 bits of precision are sufficient for accurately locating bugs, offering precision similar to a window size of 512 cycles. Thus, the resulting probability can be truncated and stored with fewer bits. The final result is copied to a memory buffer at the end of each window.

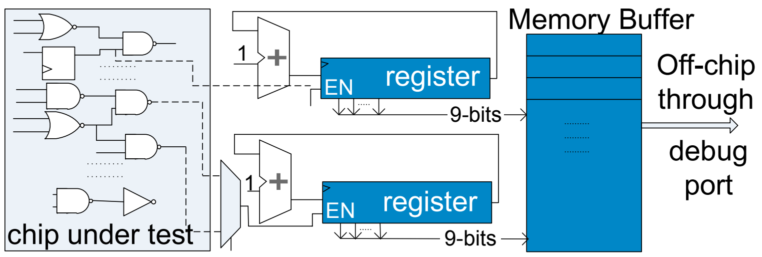


**Figure 4.4 BPS Hardware** collects signatures for a subset of the design's signals. An observed signal's time@1 is tracked at each cycle for the duration of a window; the sum is then truncated to limit its size and saved to a buffer.

Note that it is not necessary to collect signatures for every signal in the design. BPS leverages signals high in the module hierarchy, those most likely to be available for observation in a post-silicon validation platform. To further reduce the amount of data that must be transferred off-chip, BPS uses two signal selection optimizations: first, it excludes data signals, often identified as busses 64-bits wide or more for a 64-bit processor design. Depending on hardware resources, signatures can be collected all at once or in groups. If post-silicon debugging hardware resources are scarce, then multiple executions of the test can be leveraged to complete the signature collection, even if those executions are not identical, since BPS' post-analysis software is tolerant to variation.

## 4.3.2 BPS Post-Analysis Software

After on-line signature collection is completed, off-line software analysis identifies a set of signals indicating where the bug occurred and at what time. BPS uses the signatures from passing runs of the test to build a model of expected behavior, and then determines when failing executions diverge from the model, revealing a bug.

BPS' software begins by partitioning a test's signatures into two groups: those where the test passed, and those where the test failed, as illustrated in Figure 4.5 (top and bottom portions). The signatures in each group are organized by window and signal: for each window/signal combination, BPS considers multiple signature values, the result of multiple

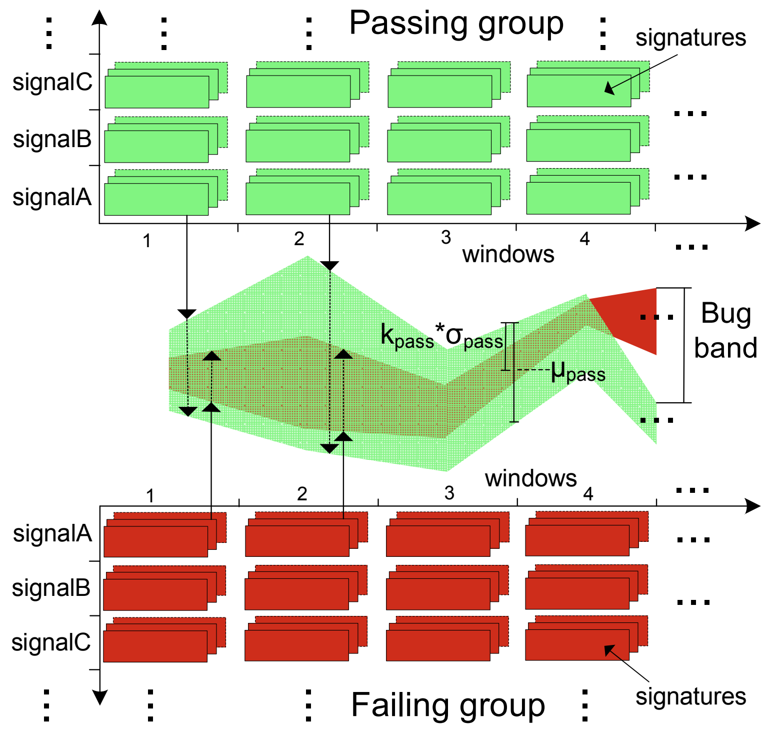


**Figure 4.5 BPS post-analysis algorithm.** Using the passing group of signatures from a test, BPS builds a model of the expected behavior for each signal, shown by the green (light gray) band. The red (dark gray) band shows the behavior of the failing test runs, constructed from the failing group.

executions of the test. Next, passing signatures are used to build a model of acceptable system behavior for each observed signal: the algorithm goes through all the signatures related to one signal, building the model one window at a time.

The middle part of Figure 4.5 illustrates the model built for signalA as a green (light gray) band. Representing the expected behavior as a distribution of values enables BPS to tolerate variations in signature values since, as we discussed above, post-silicon validation is characterized by non-identical executions due to naturally occurring variations among distinct executions. Figure 4.6 illustrates how distributions are used to build a model of observed behavior. The passing band for one signal is generated by computing the mean ($\mu_{pass}$) of the signature values for each window, surrounded by $k_{pass}$ standard deviations ($\sigma_{pass}$), where $k_{pass}$ is a parameter. Thus the band representing the passing signatures is bounded by $\mu_{pass} \pm k_{pass} * \sigma_{pass}$. In order to represent over 95% of uniformly distributed data points, we used $k_{pass} = 2$ for our experiments.

The BPS software now adds the failing group to the model, once again considering each signal in turn and building the model window-by-window. The failing group is plotted in Figure 4.5 as a red (dark gray) band. Similar to the passing group, the failing group is modeled as the mean surrounded by $k_{fail}$ standard deviations ($\mu_{fail} \pm k_{fail} * \sigma_{fail}$). When the failing band falls inside the passing band, we deem the corresponding signal's behavior to be within an acceptable range, indicating that a test failure has not yet occurred or, possibly it is masked by noise. When it diverges from the passing band, we identify this as buggy behavior.

Using this band model, BPS determines when failing signatures diverge from passing signatures: we call the divergence amount a *bug band*. Starting at the beginning of a test execution, the algorithm considers each window in turn, calculating the bug band one signal at a time. The bug band is zero if the failing band falls within the passing band, otherwise it is the difference between the two top (or bottom) edges. As an example, Figure 4.6 shows the model obtained and the bug band calculation for a signal in the memory stage of a 5-stage pipelined processor.

The set of bug bands (one for each signal) is ranked and compared against a threshold that varies with the design (see Section 4.3.4). If no bug band exceeds the threshold, BPS moves on to the next window. When one or more bug bands exceed the threshold, BPS notes the time (represented by the window) and the signals involved, reporting them as the bug time and location.

As an additional filtering step, a set of common mode rejection signals is leveraged by BPS to mitigate the noise present in large designs. To generate this filter, BPS is run with two passing groups of a same testcase, rather than a passing and a failing group. The signals identified in this process are removed from BPS' candidate bug signals list; this helps to minimize the number of false positives. We found that in a complex design (OpenSPARC

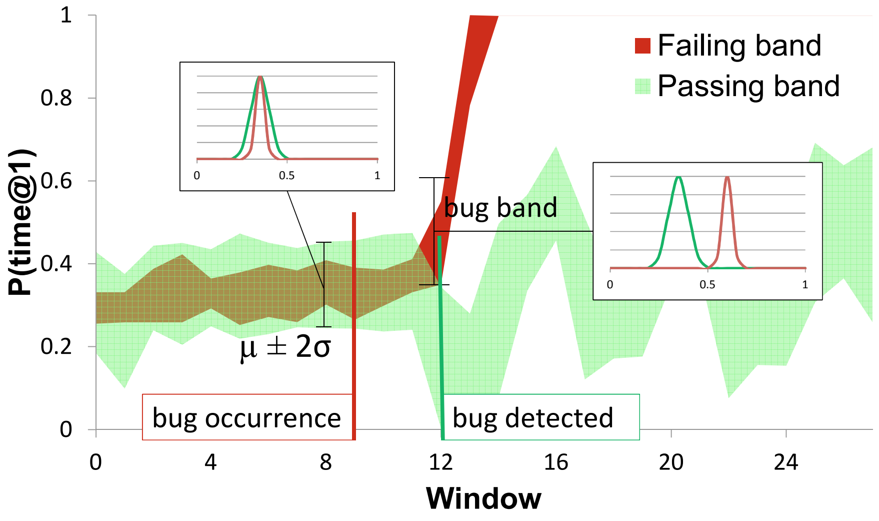


**Figure 4.6 BPS band model.** Band model for a memory control signal from a 5-stage pipelined processor. Each slice of time in the model represents two distributions (passing and failing). The bug is detected when the failing band diverges from the passing one, representing diverging signature distributions. The quantitative amount is measured by the bug band.

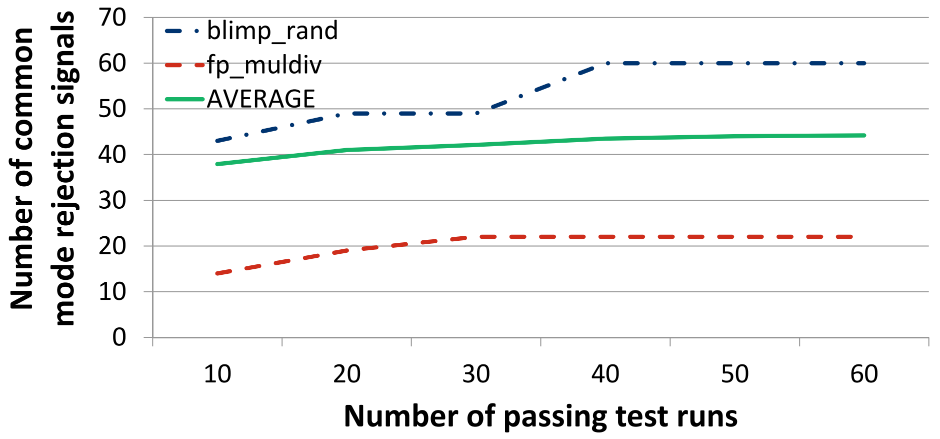


**Figure 4.7 Number of common mode rejection signals** in the OpenSPARC T2 design. As the number of passing tests increases, the average number of signals stabilizes to 44, only 1% of the signals monitored by BPS.

T2), as the number of runs used for identifying common mode rejection signals increased, the resulting list stabilized. Figure 4.7 shows this asymptotic trend for the testcases exhibiting the largest and smallest common mode rejection signal lists, as well as the average. The size of the list is typically small, only 44 signals (some of which are buses), comprising 1% of the design's monitored signals.

## 4.3.3 Tuning Parameters

A number of parameters affect the quality of the results produced by BPS: the bug band threshold, the window length, the number of test executions in both the passing and failing groups and a set of common mode rejection signals.

The **bug band threshold** is used to determine which signals BPS detects, and also causes BPS to stop looking for bugs. Changing this value changes BPS' sensitivity to bugs and noise. In some cases, the design perturbation caused by a bug can be amplified by neighboring logic over time: a higher bug band threshold can cause BPS to detect these neighboring signals after searching longer (more windows) for errors. The result is often a reduction in the number of signals detected, since few signals have a bug band that exceeds the threshold. However, this can also lead to signals that are less relevant to the error, as well as longer detection times. On the other hand, a bug band threshold that is too small can result in prematurely flagging irrelevant signals, halting the search for the bug. In our experiments, we found that a single threshold value could be used for each design. Thus, in practice, the proper bug band threshold is determined when running the first tests, and then reused for the rest.

The **window length** is the time interval (in cycles) of signature calculation, and affects the precision of BPS' timing. Increasing the window length increases the number of cycles that must be inspected after BPS reports the bug detection window. However, large window lengths have the advantage of allowing longer periods of execution between signature logging and thus decrease the volume of data that must be transferred off chip. Thus, the choice of window size is a trade-off between off-chip data transfer times and the precision of bug localization timing.

The **population size of passing and failing groups** primarily affects false negative and false positive rates. When the population of failing runs is small, variations in the failing group have greater impact on the mean. Thus, bugs are triggered more easily, resulting in increased false positives. Conversely, when the number of passing testcases is small, variations impact the mean of the passing population, this time increasing the false negative rate.

### 4.3.4 Experimental Evaluation

In order to evaluate the effectiveness of BPS, we employed it to find bugs on two microprocessor designs with a variety of failures, including electrical, manufacturing and functional bugs. Each processor ran a set of 10 distinct application workloads. The designs are a 5-stage pipelined processor implementing a subset of the Alpha ISA, comprising 4,901 lines of code and 4,494 signals (bits). After excluding data signals, BPS was left with 525 signals for analysis. Our larger industrial design, the OpenSPARC T2 [140] system, has 1,289,156 lines of code and 10,323,008 signal bits. We simulated the system in its single core version (cmp1), which consisted of a SPARC core, cache, memory and crossbar. BPS monitored the control signals at the top level of the design for a total of 41,743 signal bits, representative of the signals that would likely be available during post-silicon debugging of such a large design. Both designs were instrumented to record signatures during logic simulation; execution variations were introduced with variable and random communication latencies. BPS requires only these compact signatures and pass/fail status of the test to operate.

Table 4.1 shows the bugs introduced in 10 different variants of the design, with one bug in each. The failures included functional bugs (design errors), electrical failures and manufacturing errors. Functional bugs were modeled by modifying the design logic, and electrical failures were simulated by temporary single bit-flips persisting for a number of cycles. Manufacturing errors were modeled as single bit stuck-at faults lasting for the duration of the test. Each design variant executed several tests a number of times, with a checker to determine if the final program output was correct. The workloads used as test inputs for the two processor designs included assembly language tests, as well tests from a constrained-random generator. There were 10 tests for each design (Table 4.2), ranging in size from about 20K cycles to 11M cycles. Each test was run 10 times for each bug, using 10 random seeds with varying impact on memory latency. Additionally, each test was run 10 times (with new random seeds) without activating the bug to generate the passing group.

### Bug Localization

Table 4.3 shows the quality of BPS bug detection for the 5-stage pipeline and OpenSPARC T2 designs: eventually, BPS was able to detect the occurrence of every bug. Often, the exact root signal was detected, a few exceptions include 5-stage's `EX SA` and `cache SA`, as well as OpenSPARC's `BR fxn` and `MMU fxn`, where the root bug signal was deep in the design and not monitored by BPS (indicated by light shading). In these situations, BPS was still able to identify signals close to the bug location. In a few cases with the OpenSPARC design, BPS did not find an injected bug, a false negative. Finally, we ob-

| **5-stage pipeline bugs** | **description** |
|---|---|
| ID fxn | functional bug in decode |
| EX fxn | functional bug in execution unit |
| fwd fxn | functional bug in fwding logic |
| EX SA | stuck-at in execution |
| cache SA | stuck-at in cache to proc ctrl |
| ID SA | stuck-at in decode |
| MEM SA | stuck-at in memory |
| WB elect | electrical error in writeback |
| ID elect | electrical error in decode |
| EX elect | electrical error in execute |
| **OpenSPARC T2 bugs** | **description** |
| PCX gnt SA | stuck-at in PCX grant |
| XBar elect | electrical error in crossbar |
| BR fxn | functional bug in branch logic |
| MMU fxn | functional bug in mem ctrl |
| PCX atm SA | stuck-at in PCX atomic grant |
| PCX fxn | functional bug in PCX |
| XBar combo | combined electrical errors in Xbar/PCX |
| MCU combo | combined electrical errors in mem/PCX |
| MMU combo | combined functional bugs in MMU/PCX |
| EXU elect | electrical error in execution unit |

**Table 4.1 Designs and modeled failures.** The bugs introduced in each design variant were functional, electrical and manufacturing (stuck-at) failures.

| **5-stage pipeline tests** | **description** | **length (cycles)** |
|---|---|---:|
| bubblesort | bubble sort | 569,237 |
| combRec | recursive combinations | 4,609,206 |
| fib | Fibonacci numbers | 442,233 |
| hanoi | tower of Hanoi puzzle | 11,593,567 |
| insert | insertion sort | 229,429 |
| knapsack | knapsack problem | 1,597,497 |
| matmult | matrix multiplication | 1,723,423 |
| merge | merge sort | 548,172 |
| quick | quick sort | 277,503 |
| saxpy | scalar alpha x plus y | 60,024 |
| **OpenSPARC T2 tests** | **description** | **length (cycles)** |
| blimp_rand | hypervisor test | 251,480 |
| fp_addsub | floating point add/subt | 913,093 |
| fp_muldiv | floating point mult/div | 238,343 |
| isa2_basic | constrained-random | 452,009 |
| isa3_asr_pr | constrained-random | 1,178,151 |
| isa3_window | constrained-random | 1,282,348 |
| mpgen_smc | constrained-random | 135,251 |
| ldst_sync | thread sync. instrs. | 64,570 |
| n2_lsu_asi | load/store unit test | 62,523 |
| tlu_rand | trap logic unit test | 591,434 |

**Table 4.2 Workloads for both experimental designs.**

served false positives in two testcases, instances where the system detected a bug before it was injected: both were floating point testcases (`fp_addsub` and `fp_muldiv`). Upon further investigation, we found the cause to be three signals that exhibited noisy behavior, but were not included in the common mode rejection filter. When these three signals were added to the filter, the false positives were correctly avoided, highlighting the effectiveness of rejecting noisy signals.

Some bugs were easier to detect than others, for example BPS was able to detect the exact bug root signal in 8 out of 10 testcases with the `PCX atm SA` bug, while a seemingly similar bug, the `PCX gnt SA`, did not manifest in 9 out of 10 cases. `PCX atm SA` had wider effects on the system, and thus manifested more frequently and was easier to detect. By contrast, the `PCX gnt` signal was not often used and thus the related bug did not manifest as frequently.

The number of signals and the time between bug occurrence and bug detection are also a consideration in post-silicon validation: it is easier to debug a small number of signals that are close to the bug's manifestation. Figure 4.8 shows the number of signals identified by BPS for the bugs in each design. Each bar of the figure represents one bug, averaged

| **5-stage** | ID fxn | EX fxn | fwd fxn | EX SA | cache SA | ID SA | MEM SA | WB elect | ID elect | EX elect |
|---|---|---|---|---|---|---|---|---|---|---|
| bubblesort | ✓+ | ✓+ | ✓ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |
| combRec | n.b. | ✓ | ✓ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |
| fib | ✓+ | n.b. | ✓ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |
| hanoi | n.b. | n.b. | ✓ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | n.b. |
| insert | ✓+ | ✓+ | ✓ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |
| knapsack | ✓ | ✓+ | ✓ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |
| matmult | ✓ | ✓+ | ✓+ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |
| merge | ✓+ | ✓ | ✓ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |
| quick | ✓+ | ✓+ | ✓ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |
| saxpy | ✓ | ✓+ | ✓+ | ✓ | ✓ | ✓+ | n.b. | ✓+ | ✓+ | ✓+ |

| **OpenSPARC** | PCX gnt SA | XBar elect | BR fxn | MMU fxn | PCX atm SA | PCX fxn | XBar combo | MCU combo | MMU combo | EXU elect |
|---|---|---|---|---|---|---|---|---|---|---|
| blimp_rand | ✓+ | ✓+ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | f.n. | ✓+ | f.n. |
| fp_addsub | n.b. | f.p. | ✓ | ✓ | ✓ | ✓+ | f.p. | n.b. | ✓+ | f.p. |
| fp_muldiv | n.b. | f.p. | ✓ | ✓ | ✓ | ✓+ | f.p. | f.p. | ✓+ | f.p. |
| isa2_basic | n.b. | f.n. | ✓ | n.b. | ✓+ | ✓+ | ✓+ | ✓+ | n.b. | f.n. |
| isa3_asr_pr | n.b. | ✓ | ✓ | f.n. | ✓+ | ✓ | ✓+ | ✓+ | ✓ | ✓ |
| isa3_window | n.b. | ✓ | ✓ | n.b. | ✓+ | ✓ | f.n. | f.n. | n.b. | ✓ |
| ldst_sync | n.b. | ✓+ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ | n.b. |
| mpgen_smc | n.b. | ✓+ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |
| n2_lsu_asi | n.b. | f.n. | ✓ | f.n. | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ | n.b. |
| tlu_rand | n.b. | ✓+ | ✓ | ✓ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ | ✓+ |

**Table 4.3 BPS signal localization.** Checkmarks (✓) indicate that BPS identified the bug; the exact root signal was located in cases marked with ✓+. Each design includes two bugs involving signals not monitored by BPS (light shading). In these cases, BPS could identify the bug, but not the root signal. "n.b." indicates that no bug manifested for every run of the test; false negatives and false positives are marked with "f.p." and "f.n.".

over all tests used in BPS, using a window length of 512 cycles. We found that the number of signals is highly dependent on the bug, with BPS detecting a single signal for some bugs, such as 5-stage's `MEM SA` and OpenSPARC's `MCU combo`. Other bugs were more challenging, for example, with the 5-stage pipeline's bug `WB elect`, BPS detected 158 signals on average: this was due to very wide-spread effects of this single bug throughout the design. We also noted that this catastrophic bug was caught by BPS very quickly, less than 750 cycles after the bug's manifestation. While BPS monitored 80x more signals in the OpenSPARC experiments, the number of detected signals increased by only 2x, on

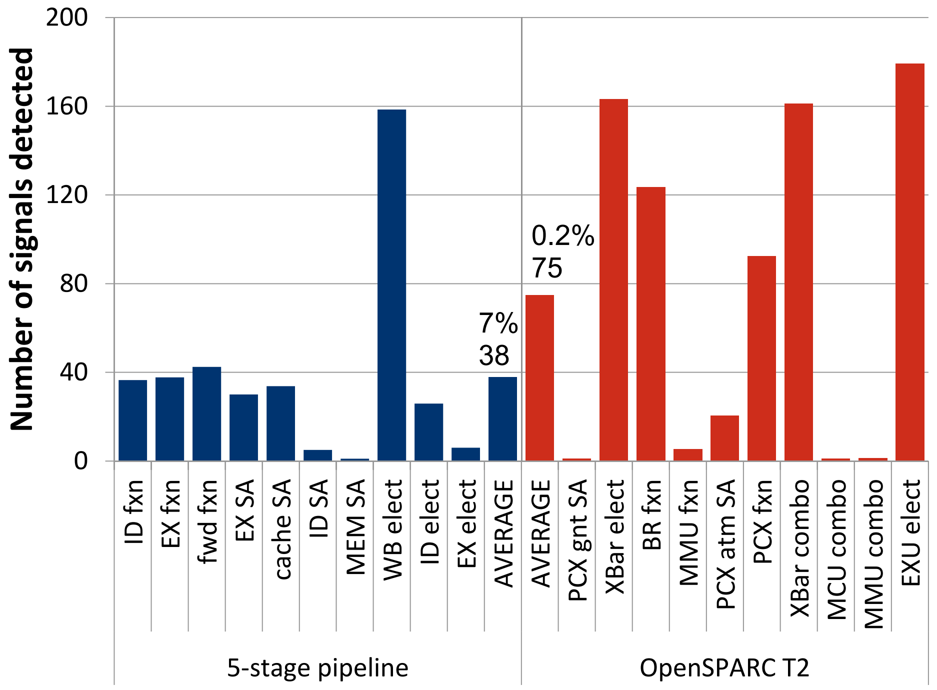


**Figure 4.8 BPS spatial localization,** number of signals identified as closely related to the bug when using a 512 cycle window.

average. This demonstrates BPS' ability to narrow a large number of candidate signals (nearly 42,000) down to a smaller pool amenable to debugging.

The time to detect each bug is reported in Figure 4.9, expressed as the number of cycles between bug injection and detection. Each bar of the figure represents one bug, averaged over all tests, using a window length of 512 cycles. The error bars indicate the error window in the BPS reporting, which corresponds to the window length. The average detection time was worse for the 5-stage pipeline; mostly due to three bugs: the `EX SA` and `cache SA` stuck-at bugs were both inserted into data busses, and thus could not be directly observed by BPS. The effects of the bug required many cycles before observable control signals diverged. In the case of the ID functional bug, the effects of the bug were masked for many cycles in the fib testcase, thus, the bug went undetected until later in the program's execution. In the OpenSPARC design, we noted that most bugs were detected within about 750 cycles, on average. Two bugs were an exception to this rule, both involving the MMU, where bugs involving signals deep in the design remained latent for a time before being detected.

Overall, BPS was successful in narrowing down a very large search space (number of signals * test length) to a small number of signals and cycles. Our experiments show that it was able to correctly reject over 99.999% of the candidate $\langle location, time \rangle$ pairs. By contrast, IFRA [108] achieves 99.8% by this metric.

## Bug Detection Quality

The quality of detection results is evaluated by both the number of signals detected and the ability to detect the source signal of the bug. The accuracy of the time at which the bug is detected is also a consideration, as are the rates of false negatives and false positives. Additionally, the number of signals must be manageable by validation engineers. A variety of factors affects the quality of BPS results, including the bug band threshold, number of executions (of the same test) and window length.

## Bug Band Threshold

Figure 4.10 shows the effect of threshold on the number of false negatives and false positives, reporting the sum total over all bugs and testcases for each threshold. We note that for both designs, the number of false positives starts high and decreases as the threshold increases, the result of a tighter filter for discerning a bug occurrence over system's noise. However, when the bug band threshold is high, the subtler effects of bugs are overlooked by BPS, resulting in more bugs being missed. In contrast, the number of false negatives increases as the threshold increases. Thus, there is a trade-off between false positive and false negative rates, indicated by the minimum of their sum, occurring at 0.3 for the 5-stage

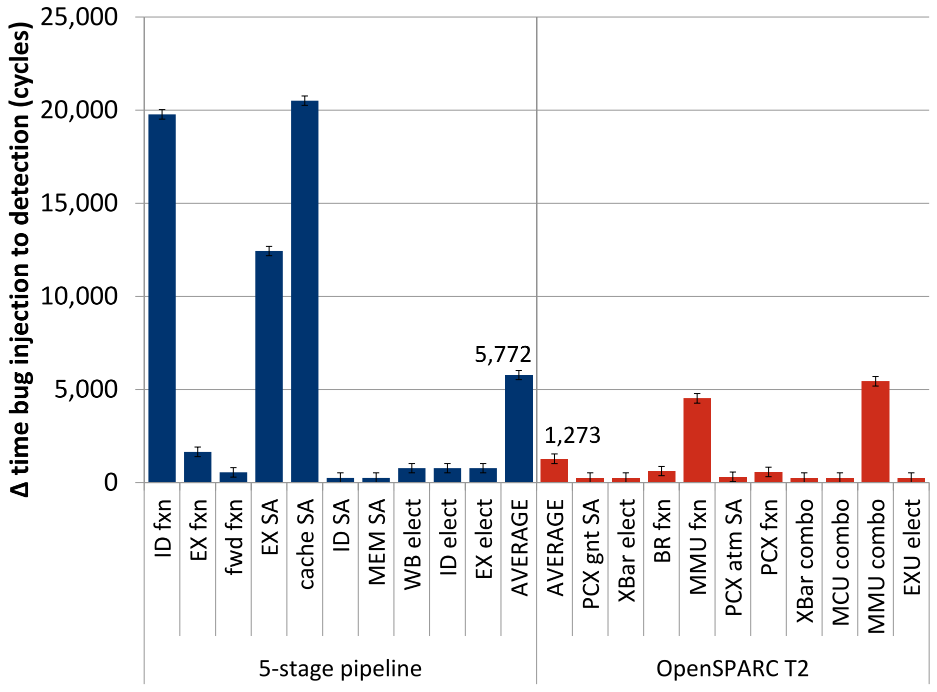


**Figure 4.9 BPS temporal localization,** in cycles between bug injection and detection. Error bars indicate the localization range.

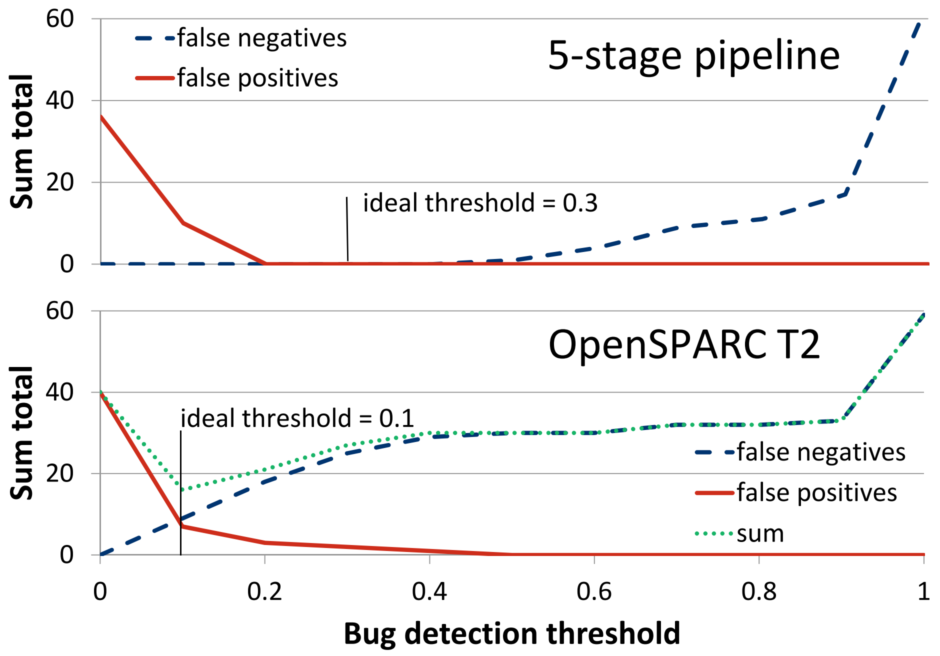


**Figure 4.10 Bug band threshold and quality of results.** As the threshold increases, false negatives increase, while false positives decrease. Thus, we select a threshold by minimizing the sum.

pipeline and 0.1 for OpenSPARC T2. We also noted that the number of signals detected decreased as the threshold increased: with high thresholds, BPS detects neighboring signals after searching longer (more windows) for errors.

### Population Size

The ratio of the number of passing vs. failing runs determines the weight of signatures that may vary from the mean. Figure 4.11 plots the number of false positives and false negatives as the balance of passing vs. failing testcases changes. At the center of the X-axis, the balance is equal, with 10 passing and 10 failing tests. In the case of the 5-stage pipeline, we note that false negatives are not present, regardless of the number of runs, indicating that the wide-spread effects of a bug in a small design will always be caught eventually. In both designs, the number of false positives decreases with more failing testcases. When the number of failing runs is small, only a few data points that differ from the mean can exceed the bug band threshold, thus false positives are more prevalent. Conversely, we see an increase in false negatives as the balance tips towards more failing. Note that for our other experiments, we used an even balance of 10 passing and 10 failing runs.

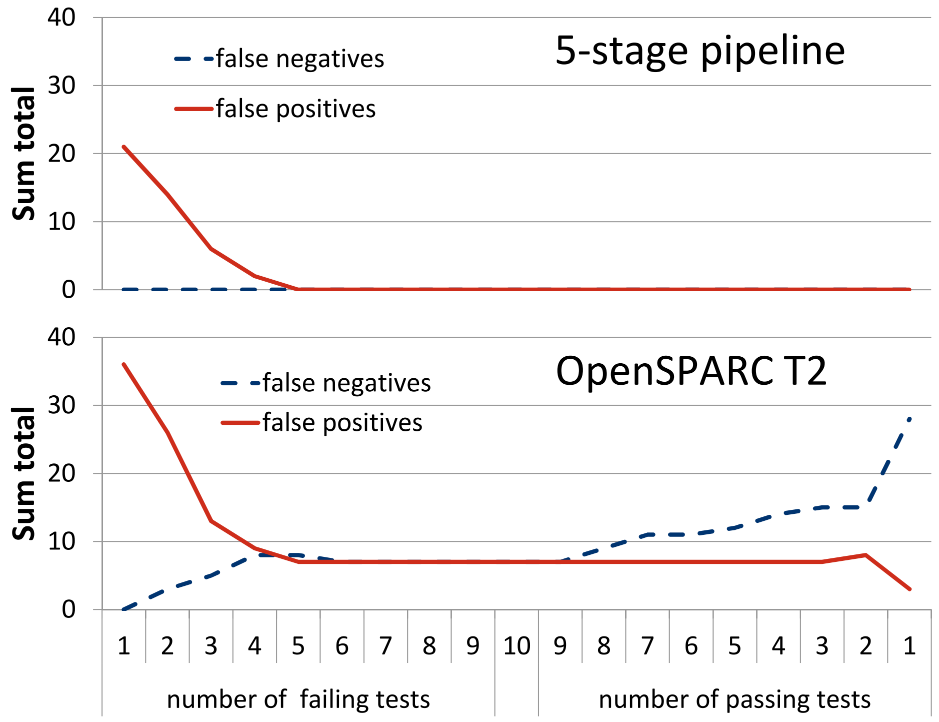


**Figure 4.11 Population size and quality of results.** The plot shows false negatives and false positives as the population of passing and failing runs changes:. 10 passing runs on the left and 10 failing runs on the right.

### Window Length

Next, we examine the interaction of window length with quality of results. BPS measures signatures over a time interval and the interval length affects the time at which bugs are detected. We found that the window length had a significant effect on the number of signals detected and the detection time. Figure 4.12 plots the number of signals detected, as well as the time between bug injection and detection as window length increases. First, we observed that shorter windows yield more accurate time localizations: for long windows, the lag is mostly due to the length of the window itself. Note that failing runs may execute long past the bug detection, since the cycle is identified during post-analysis. Additionally, the number of signals detected increases as the window length increases. This is due to the increased time for the effects of the bug to spread to many signals in the system. Thus, a smaller window length yields more accurate results. We note however, that it would be possible to run BPS multiple times with an iterative approach, strategically decreasing the window size in order to narrow down a bug's location.

**Performance and Area Overhead**

BPS' software post-analysis was run on Xeon Core i7 2.27GHz servers, and the time for running each analysis was approximately 412s for OpenSPARC T2 and 5s for the 5-stage design. This time varies with the number of signals under observation, as BPS must consider more data. CPU time is also linearly dependent on the bug detection window, since BPS must sift through more windows searching for bugs located deep in a test execution.

The hardware logging of BPS' simple signatures can be obtained using either standard flexible debugging infrastructures, or with custom hardware. In the case of pre-existing debugging hardware, signatures can be gathered with no area overhead.

With its unique ability to leverage data from non-repeatable test executions, BPS enables a trade-off between area overhead and the time required to gather signature data. With a small area budget, the signatures for a set of signals can be gathered a few signals at a time. Leveraging fast post-silicon execution, a test is run multiple times, recording signatures from a different subset of signals with each run. Variation among different runs averages out in BPS' statistical approach, and thus does not impact the diagnosis quality.

We evaluated the area overhead of a BPS hardware sensor implementation in Verilog HDL, synthesized with a 65nm TSMC target library. A signature recording unit capable of 100 signals over 100 windows, recording signatures for the 41,743 signal bits in the OpenSPARC design would require 418 test executions, a reasonable demand at fast post-silicon execution speeds. With 9-bit precision for signature storage, full precision for a window length of 512 cycles, the resulting memory buffer comprises 1.33 mm$^2$. The hardware to

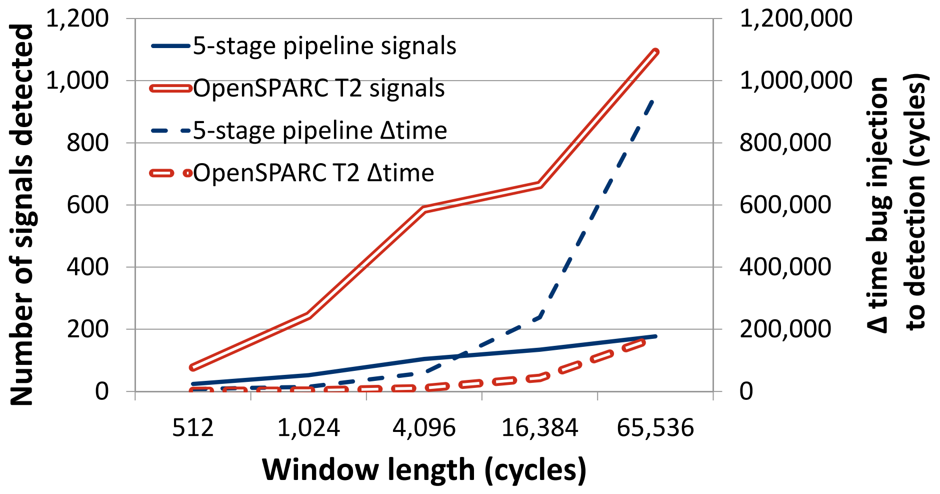


**Figure 4.12 Impact of window length on quality of results.** The plot reports the number of signals detected, and the time between bug manifestation and bug detection as window length increases. Shorter window lengths yield faster, more precise time localization.

generate these signatures occupies 23,240 $\mu$m$^2$, resulting in a total area of 1.35 mm$^2$. Compared to the OpenSPARC T2 system (342 mm$^2$[45]), the area overhead of BPS is 0.396%, less than half the overhead of IFRA [108]. When comparing storage, the dominant factor in both BPS' and IFRA's overhead, BPS requires 11KB with this configuration, compared to 60KB for IFRA.

### Limitations

While BPS is effective in localizing a wide variety of functional, electrical and manufacturing failures, it has a few limitations.

The signals available to BPS for observation play a role in its ability to accurately localize bugs. The scope of signals available for observation during post-silicon validation varies with the quality of its debug infrastructure. When the signals involved in a bug are monitored by BPS, it is highly effective in identifying failures down to the exact source signal, illustrated qualitatively in Section 4.3.4. However, when the source signal is deep in the design and not monitored, the accuracy of BPS is reduced. This results in an increased number of signals detected, as well as increased detection time. Thus, BPS is able to identify bugs that originate either within or outside of its observable signals, but it can only identify the exact signal when this signal is monitored.

The interaction of the test with other code running on the post-silicon validation platform also alters the quality BPS' results. When similar code executes, the signal activity recorded by BPS' signatures is an effective input for the post-analysis algorithm. However, when the code path significantly deviates, such as with the introduction of an operating system, differing code paths among test executions can be mistaken for erroneous behavior, resulting in an increased false positive rate. Currently, BPS works best with bare-metal tests that are free from operating system interaction.

The relationship between window size and the duration of a bug also affects BPS. A bug's duration comprises the perturbation in the source signal and the after-effects that may spread to nearby connected logic. When the bug duration is small relative to the window size, its effect on the signature recorded for a window is proportionally small ($bugband < 2\sigma$), sometimes resulting in false negatives, depending on the bug band threshold. The effect of short bug durations can be counteracted by a smaller threshold, as well as by smaller window sizes. We most often observed this phenomena when identifying bug root signals. As window sizes increased, the number of cases where BPS detected the exact root signal decreased, despite being able to detect other signals related to the bug. Upon further investigation, we found that in many cases, the duration of the perturbation of the bug's root signal was small compared to the window size, while the secondary effects of the bug remained observable in the design's behavior.

## 4.4 Summary

We have presented an effective approach to mitigating electrical failures during post-silicon validation. Easing and accelerating post-silicon bug diagnosis reduces the possibility of escaped bugs, and contributes to our larger goal of ensuring correct operation.

First, our BiPeD framework is applied to detect post-silicon failures. When an error is detected, the location of the bug is pinpointed using BPS. BPS is a solution for locating the most challenging electrical failures during post-silicon validation. In addition to electrical failures, we found that it was also effective in locating functional bugs and manufacturing faults with inconsistent program outcomes. BPS has two components: hardware structures that log a compact encoding of observed signal activity and companion post-analysis software. BPS can localize bugs in time and space while tolerating non-repeatable executions of the same test. It provides a fast solution, reducing off-chip data transfers with compact signatures and scales to industrial-size designs. BPS is effective in locating bugs under many different workloads, often to the exact signal.

In the next chapter, we address electrical failures at runtime in conjunction with related wear-out induced transistor faults.

# Chapter 5
# Addressing Transistor Faults

As critical transistor dimensions continue to scale further into the nanometer regime, chips become increasingly susceptible to wear-out induced errors. In order to ensure correct operation throughout the lifetime of a chip, we address transistor faults due to wear-out at runtime. Faults must first be detected, identifying behavior that differs from correct operation. Next, the error is diagnosed, narrowing down the exact hardware block or subsystem responsible for the fault. Once the hardware block is identified, reconfiguration can take place, rearranging connections to work around the error. Finally, a recovery mechanism aids in recouping any data lost due to the error. This process is shown in Figure 5.1.

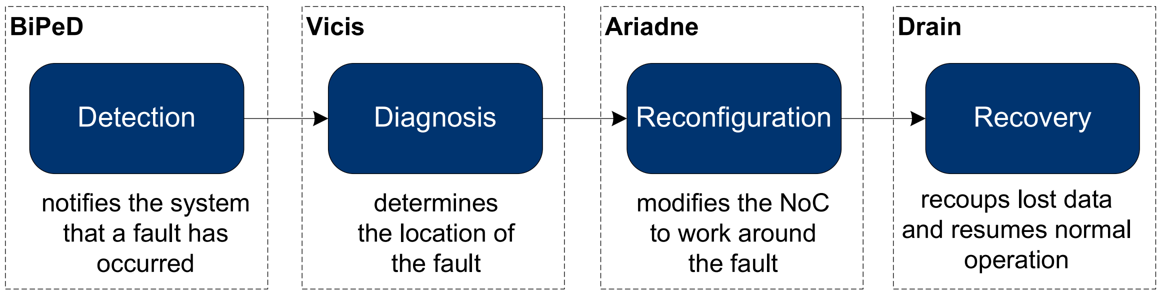

BiPeD
Detection
notifies the system that a fault has occurred
Vicis
Diagnosis
determines the location of the fault
Ariadne
Reconfiguration
modifies the NoC to work around the fault
Drain
Recovery
recoups lost data and resumes normal operation

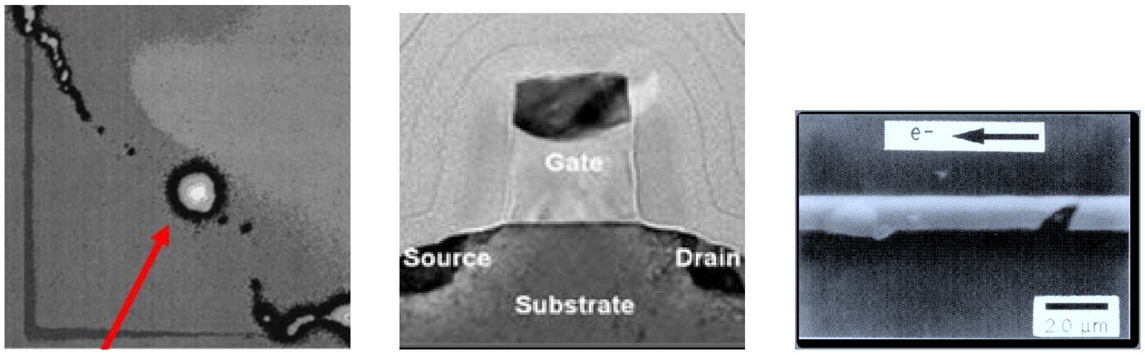
**Figure 5.1 Transistor faults are addressed in four steps:** detection, diagnosis, reconfiguration and recovery (bubbles in the chart). The portions of this dissertation that cover the four steps are shown by the dashed boxes.

In Section 5.1, we discuss the origins of transistor faults, and how they affect the on-chip communication mechanism. We also present a detailed fault model. Next, we again leverage BiPeD's flexible protocol detectors to detect faults (Section 5.2). Following fault detection, we apply an NoC router architecture capable of diagnosing faults called Vicis (Section 5.3). After the fault location has been diagnosed, the Vicis architecture reconfigures around small errors. When faults cannot be contained within the router, our Ariadne routing algorithm is invoked to reconfigure the interconnect around broken routers and links (Section 5.4). Finally, any data isolated due to a network reconfiguration is recovered by Drain (Section 5.5).

## 5.1 Transistor Faults

With continued aggressive transistor scaling, these tiny parts become increasingly susceptible to silicon wear-out. Faults that are the result of worn-out transistors, which can cause both electrical and transistor problems, must be addressed at runtime. Both pre-silicon and post-silicon verification will miss these faults, since they occur over time. First, the communication architecture — generally a network-on-chip — must be capable of diagnosing and reconfiguring broken components. Next, as faults accumulate, a reliable routing algorithm should address broken communication links and disconnected network nodes. Finally, following a reconfiguration by the reliable routing algorithm, data must be recovered from the broken components so that computation can resume.

Transistor faults can be caused by a variety of wear-out mechanisms in highly scaled technology nodes. As transistor dimensions approach the atomic scale, oxide breakdown [136] (Figure 5.2a) becomes a concern, since the gate oxide tends to become less effective over time. Moreover, negative bias temperature instability (NBTI) [9] is of special concern in PMOS devices, where increased threshold voltage is observed over time (Figure 5.2b). Additionally, thin wires are susceptible to electromigration [58], because conductor material is gradually worn away during chip operation until an open circuit occurs (Figure 5.2c). Since these mechanisms occur over time, traditional burn-in procedures and manufacturing tests are ineffective in detecting them.



**(a)** oxide breakdown **(b)** NBTI **(c)** electromigration

**Figure 5.2 Wear-out mechanisms,** which can eventually cause permanent faults at runtime. Oxide breakdown affects thin gates, which degrade over time. Negative bias temperature instability (NBTI) causes changes in the threshold voltage of PMOS devices. Finally, electromigration results in eroding wires, leading to conductor fault.

The rate of transistor wear-out changes as a chip ages. Figure 5.3 shows a hypothetical "bathtub curve", with the expected lifetime of a chip as transistors wear out. The normal lifetime, in between infant mortality and end-of-life wear-out, is decreasing. Our goal in this chapter is to increase the useful life of the chip by trading off performance for

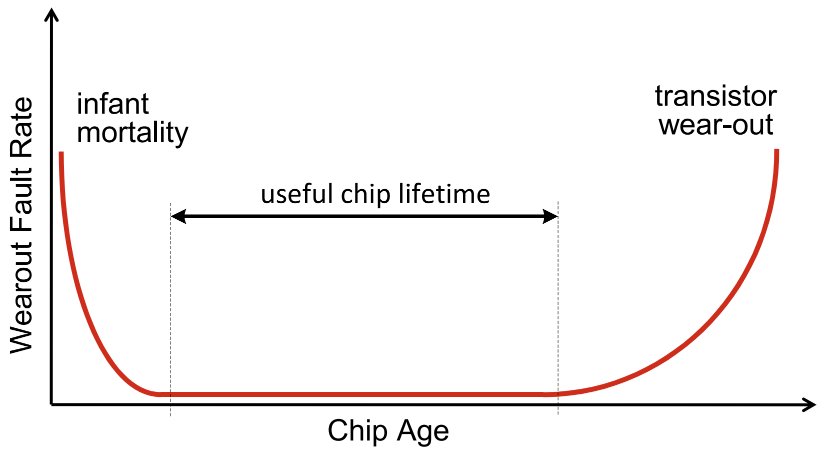


**Figure 5.3 Hypothetical "bathtub curve"** showing the shortened expected lifetime of a chip as transistors wear out [150]. Early in the chip's lifetime, many faults occur as a result of weak transistors. Many of these are eliminated with manufacturing test burn-in. At the end of the chip's life, many fail due to transistor wear-out. Our goal is to enable correct operation in the end-of-life regime, extending mean time to failure.

correctness, enabling graceful performance degradation as transistors wear out.

Our goal is to address wear-out in the transistors of the on-chip communication medium. Continuously shrinking transistor dimensions enable ever-increasing density on modern microchips: each new technology node facilitates additional cores in chip multiprocessors. For example, the Intel SCC [97] contains 48 cores, the Tilera Tile64 has 64 cores [15] and the experimental Intel Polaris chip incorporates 80 cores [144]. However, bus communication and crossbar interconnects have not scaled efficiently: high core counts necessitate efficient, scalable interconnects capable of providing communication among the processor cores. Networks-on-chip alleviate this problem with fast, scalable communication provided by small, distributed, packet-switched routers [36].

Network-on-chip routers communicate via a common interconnect, connecting processor cores, memory controllers, *etc*. At each node (usually a core or memory), a network interface controller (NIC) connects the core to the local router, and converts messages from the core into data packets of varying size for the network. These packets are further divided into flits, the smallest unit of data traveling in the network, which dictates the width of a link connecting two routers. Routers then direct traffic within the network, moving flits from source to destination according to the information encoded in each packet, usually located in the header (first flit) of the packet. In particular, in wormhole routing [105], a single packet's flits may be spread across multiple routers as they traverse the network, until all the constituent flits are collected at the destination. Compared to bus-based systems, network-on-chip designs have the advantage of allowing many messages in flight simultaneously, thus providing efficient communication among many nodes.

While NoCs provide a scalable, distributed communication solution, they are also a single point of failure in a chip multi-processor. Unlike the cores in a CMP, which are uniform, distributed, and therefore inherently redundant, there is only one communication medium in the chip, constituting a weakness in the presence of faults. Unreliable silicon substrates, brought on by aggressively scaled transistors, threaten the reliability of on-chip communication infrastructures, where a single transistor fault in the NoC could cause the entire chip to fail [102]. The possibility of frequent faults in the field is soon expected to become a reality [22, 134], leading to system failure [55, 23] or even causing security flaws [111].

### 5.1.1 Fault Model

With the reality of decreasing transistor reliability and increasing faults, our goal is to mitigate permanent faults, those that affect the hardware for the remaining life of the chip. Thus, our fault model uses stuck-at faults at the hardware level to portray these permanent faults. Our goal is ensure that all permanent faults in router datapath, and control logic are handled. Furthermore, any hardware additions may be susceptible to the same faults that they aim to address: *faults may occur in the reliability hardware itself.*

Our fault model was designed to reflect the accumulation of permanent transistor faults that occur during the lifetime of a chip. We generated a fault model for our architectural simulator by evaluating the impact of faults at the gate-level and projecting it to the architectural level as link failures. To this end, we injected stuck-at faults at the gate outputs of our reliability-enhanced hardware model in randomly selected locations. Faults were injected in both baseline router components, as well as the additional reliability components. The BIST was an exception, since it can be power-gated during normal operation, and thus is much less susceptible to permanent faults. The hardware model was synthesized, placed and routed prior to fault injection. The random selection of faulty gates was weighted by gate area. This is consistent with the breakdown patterns observed experimentally by Keane, *et al.* in [76]. While this model works well for gate-level analysis, it must be abstracted for high-level architectural evaluations.

Our architectural simulator must be informed of the location of faulty links. Thus, we must map gate-level faults to link-level faults. To this end, we leveraged the fault impact analysis in our gate-level model to form a probabilistic link fault model. From 100,000 distinct RTL simulation results, we built a model mapping gate-level errors to link-level errors. Figure 5.4 shows the distribution of faults, mapping gate-level router faults to link

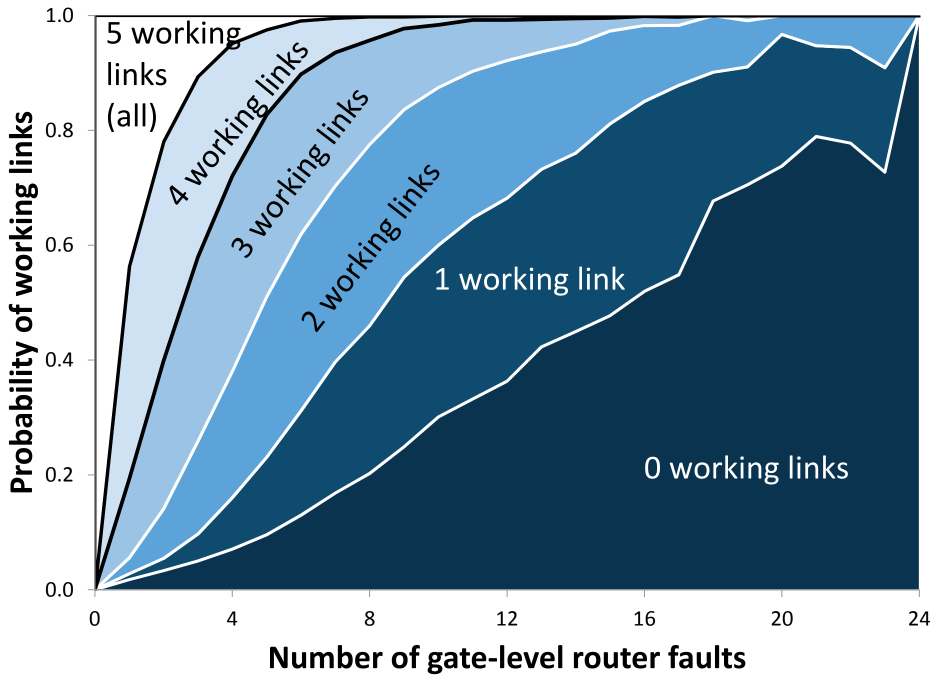


**Figure 5.4 Fault model.** The graph shows the distribution of faults among five router links as a function of gate-level faults. We leveraged 100,000 low-level HDL simulations to form a statistical fault model used by our high-level architectural simulator. The figure shows the mapping of gate-level router faults in the low-level simulations to link failures used in high-level simulation.

failures on our 5-port router. Link failures could range from 0, indicating no faults, to 5, indicating that all links were faulty. For example, with 8 faults in a router, all links will be broken with probability 0.2; 1 link will be functional with probability 0.25, *etc*. This model was used in the architectural simulations to enable fast simulation with industrial benchmarks and longer traces.

In Table 5.1, we also provide detailed data mapping gate-level faults to link faults. Gate faults from 0 to 24 are shown in the leftmost column, while the following columns report the probability of having a given number of working links. At 25 faults and beyond, all links had always failed in our HDL simulations.

Table 5.1 can be used to inject faults in architectural simulations for any size topology comprised of 5-port routers. Links in the architectural simulation are disabled based on the information from the fault model. To use the fault model, first consider a network with $R$ routers and $F_{gate,network}$ gate-level fault injections for the entire network. The faults are then distributed among the routers with uniform random probability, since it is reasonable to assume that each node requires the same silicon area. Next, the gate-level faults at each router ($F_{gate,router}$), which may vary with different routers, are mapped to link faults using the table. The row to use in the table is determined by $F_{gate,router}$, and the number of work-

| gate faults | 0 working links | 1 working link | 2 working links | 3 working links | 4 working links | 5 working links (all) |
|---|---|---|---|---|---|---|
| 0 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 |
| 1 | 0.018 | 0.010 | 0.028 | 0.137 | 0.370 | 0.438 |
| 2 | 0.033 | 0.021 | 0.086 | 0.258 | 0.381 | 0.220 |
| 3 | 0.050 | 0.047 | 0.161 | 0.321 | 0.314 | 0.106 |
| 4 | 0.071 | 0.090 | 0.220 | 0.340 | 0.231 | 0.048 |
| 5 | 0.096 | 0.135 | 0.279 | 0.317 | 0.148 | 0.025 |
| 6 | 0.130 | 0.182 | 0.308 | 0.279 | 0.093 | 0.009 |
| 7 | 0.168 | 0.228 | 0.307 | 0.232 | 0.060 | 0.005 |
| 8 | 0.203 | 0.257 | 0.316 | 0.181 | 0.041 | 0.002 |
| 9 | 0.249 | 0.295 | 0.292 | 0.141 | 0.020 | 0.002 |
| 10 | 0.301 | 0.299 | 0.275 | 0.109 | 0.015 | 0.001 |
| 11 | 0.332 | 0.315 | 0.256 | 0.089 | 0.007 | 0.000 |
| 12 | 0.363 | 0.319 | 0.240 | 0.071 | 0.008 | 0.000 |
| 13 | 0.423 | 0.309 | 0.205 | 0.057 | 0.005 | 0.001 |
| 14 | 0.450 | 0.311 | 0.190 | 0.045 | 0.005 | 0.000 |
| 15 | 0.477 | 0.334 | 0.162 | 0.023 | 0.004 | 0.000 |
| 16 | 0.520 | 0.331 | 0.132 | 0.016 | 0.001 | 0.000 |
| 17 | 0.549 | 0.330 | 0.104 | 0.015 | 0.002 | 0.000 |
| 18 | 0.677 | 0.224 | 0.099 | 0.000 | 0.000 | 0.000 |
| 19 | 0.705 | 0.205 | 0.080 | 0.009 | 0.000 | 0.000 |
| 20 | 0.738 | 0.230 | 0.033 | 0.000 | 0.000 | 0.000 |
| 21 | 0.789 | 0.158 | 0.053 | 0.000 | 0.000 | 0.000 |
| 22 | 0.778 | 0.167 | 0.056 | 0.000 | 0.000 | 0.000 |
| 23 | 0.727 | 0.182 | 0.091 | 0.000 | 0.000 | 0.000 |
| 24 | 1.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |

**Table 5.1** **Fault model** mapping gate-level faults to link faults for a 5-port router. Each cell reports the probability of N gate-level faults resulting in M link faults

ing links is determined by the weighted probabilities in the row. If the number of faults exceeds 24, then all links are considered broken. Once the number of working links is established, random selection determines the direction.

## 5.2 Fault Detection — BiPeD

In the previous section, we explored the origins of transistor faults, and our method for modeling them. Now, we turn to the first step in mitigating them, detecting the occurrence of a fault. Again, we leverage BiPeD's hardware protocol detectors, which have the advantage of flexibility. After post-silicon validation is complete, its programmable detectors can be repurposed for runtime fault detection.

BiPeD's flexible protocol detectors are connected critical router control signals, monitoring them for any errant control flow. When a fault is detected, it triggers the diagnosis mechanism to determine its source, discussed in the next section. Figure 5.5 shows the integration of BiPeD's hardware detectors with Vicis' diagnosis and reconfiguration architecture. The inputs to the detector are connected to a subset of the router's internal control signals. The control signals follow a state machine that govern the flow of data, and this state machine is tracked as a protocol by the detector. When an error causes control flow to violate the router's protocol, BiPeD's detector raises the `error out` signal. This signal is connected to Vicis' BIST controller, and initiates the diagnosis and reconfiguration process. Following reconfiguration, Drain can be used to recover any lost data.

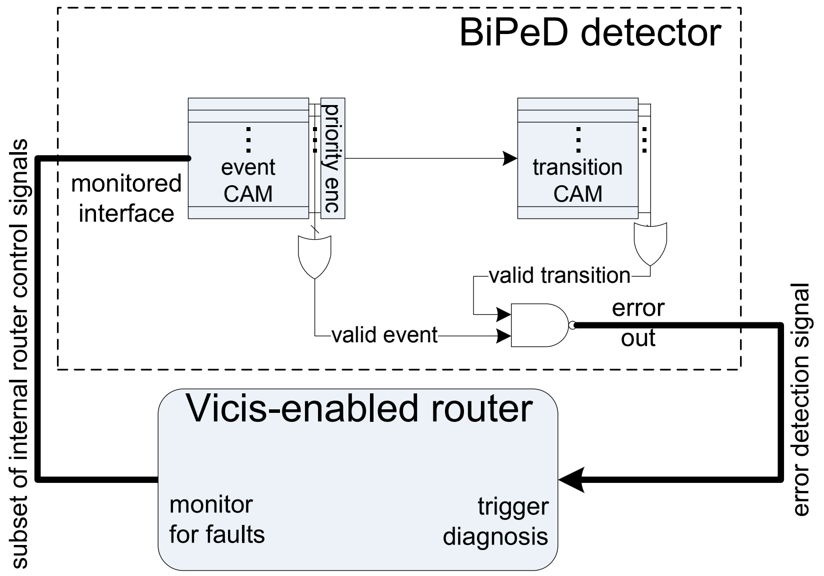


**Figure 5.5 BiPeD error detection connected to the Vicis architecture.** BiPeD's flexible, programmable protocol detectors can be used to perform error detection at runtime. The detectors are connected to internal router control signals, and flag an error when the router's control state machine deviates from correct operation. When an error is detected, Vicis' BIST controller is notified, beginning the diagnosis and reconfiguration process.

## 5.3 Fault Diagnosis — Vicis

After a fault has been detected, diagnosis proceeds with an architecture that can identify faulty hardware components. Vicis [41, 52] is a reliable architecture for networks-on-chip with mesh and torus topologies. It leverages a reconfigurable router architecture and takes advantage of the redundancy inherent in on-chip networks through reconfiguration of individual routers.

As a distributed in-hardware solution, Vicis has the advantage of being able to tolerate many faults, including faults in the reliability components. For systems built on unreliable silicon substrates, Vicis enables graceful performance degradation when transistors inevitably fail.

### 5.3.1 Architectural Features

Vicis uses an architectural capable of diagnosis and reconfiguration to maintain correct execution in a network-on-chip. When a fault is detected, the system goes offline for a Vicis-directed diagnosis and reconfiguration. It first attempts to contain permanent faults within the router, leveraging the inherent structural redundancy in the architecture to work around errors. If the fault cannot be contained within the router, Vicis can notify a companion reliable routing algorithm to network around failed nodes and links.

Reconfiguration at the router level is used to contain faults within the router, so that they are not perceivable at the network level as failed links or routers. Figure 5.6 presents a high-level schematic of a baseline router (in white) with Vicis enhancements (shaded). The baseline router includes input ports and FIFO (first in first out) buffers, decoders, a crossbar, a routing table and output ports. Vicis augments this design with a crossbar-bypass bus to protect against crossbar failures, and with error correcting codes (ECC) to protect datapath elements. Additionally, Vicis can reconfigure the FIFO buffers, the largest router components, to be resilient to a few internal faults. Our port-swapping solution allows Vicis to minimize link faults by reorganizing input ports. Finally, Vicis includes built-in self test (BIST) units to diagnose faulty router components and orchestrate reconfiguration. A complete reconfiguration process requires approximately 152,000 cycles, corresponding to only a few hundred microseconds on a 1 GHz chip. This latency does not appreciably impact application runtime for rarely occurring permanent faults (less than once a day).

#### Crossbar Bypass Bus

In the baseline router, a faulty crossbar would render the entire router inoperable. To address this issue, Vicis adds a crossbar bypass bus, as shown in Figure 5.6, an alternative path for data that may have to traverse a faulty crossbar path. The crossbar controller is configured to direct traffic to either the crossbar or the bypass bus on a packet basis. If multiple flits simultaneously require the bypass bus, one flit is chosen to proceed first, while the others must wait to use it in subsequent clock cycles. In this manner, the crossbar bypass bus may overcome any number of faults in the crossbar. This spare path provided by the bypass bus allows Vicis to maintain correct operation, even when multiple faults appear in

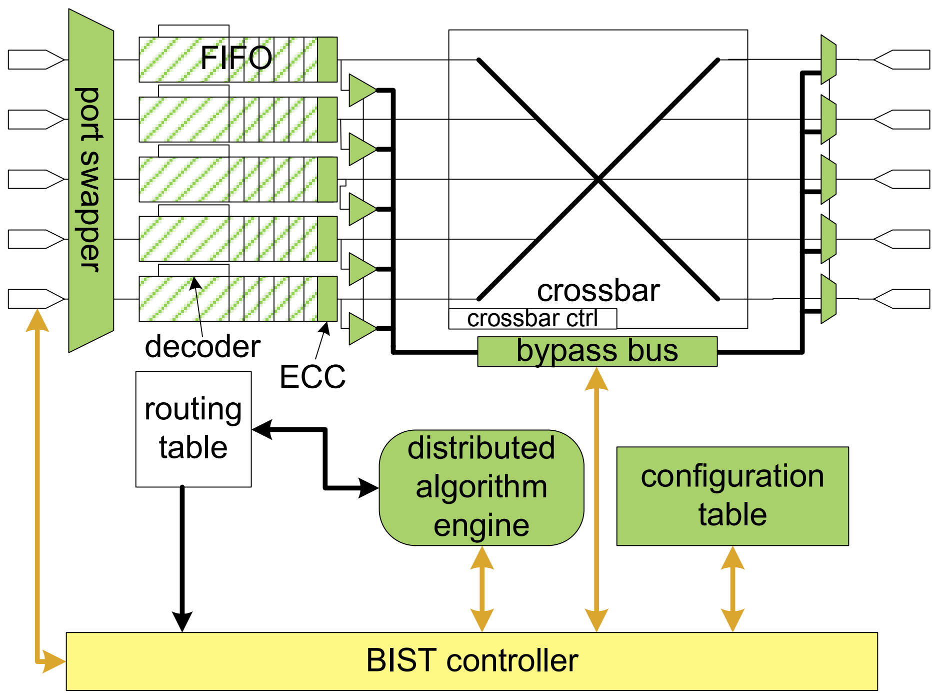


**Figure 5.6 Vicis router architecture.** A Vicis-enhanced router includes ECC units, a crossbar bypass bus, a port-swapper, BIST units for diagnosis, a distributed algorithm engine (green/gray) and flexible FIFOs (hashed), in addition to baseline components.

the crossbar. However, in the case of a single fault, the ECC unit is sufficient to overcome the fault.

### Error Correcting Codes (ECC)

Faults along the datapath can cause data corruption and packet mis-routing. Protecting the datapath with error correcting codes (ECC) enables each component to tolerate a small number of faults while maintaining correct functionality. Previous works have explored the trade-off between energy and reliability by using fine grained error correcting codes [71]. While these studies found that end-to-end ECC was more power efficient than flit-level ECC, they require that packets reach their intended destinations. Errors in header flits that could cause packet mis-routing can only be overcome by flit-level ECC: consequently, Vicis uses flit-level ECC, with an encoder and decoder at the exit of each FIFO. The code adds an additional 6 bits to each 32-bit flit in order to enable 1-bit error correction. Any single fault that manifests along an ECC-guarded datapath section can be corrected when the flit goes through an ECC unit, located in each router at the output of the FIFO buffers. In order to take full advantage of ECC, the BIST unit tracks the location of every datapath fault and, if at all possible, it reconfigures the router to ensure that every distinct path be-

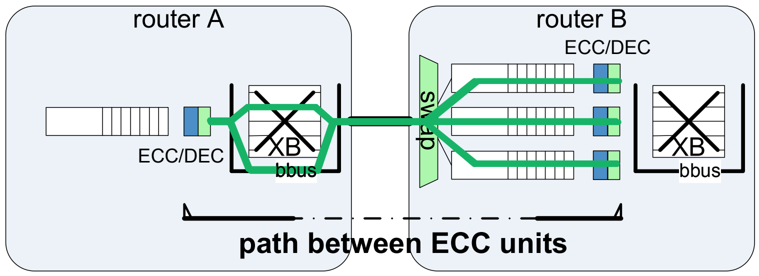


**Figure 5.7 Faults mitigated by ECC.** Datapath faults can be corrected by ECC as long as no more than one fault is encountered between two flit-level ECC units. The bypass bus and port swapper provide alternate paths between routers to reduce the number of faults that the ECC units observe. The example in the picture shows six available paths: through crossbar or bypass bus in router A, and through one of three possible FIFOs in router B (the port swapper selects which buffer to use).

tween two ECC units contains at most one fault. If this cannot be accomplished, the router is deemed faulty.

Six paths are possible between two ECC units, depending on the selection of the bypass bus or crossbar (two options) and the configuration of the port swapper (up to three options). Figure 5.7 illustrates these paths. The port swapper provides three options for the network adapter connection and two options for the other links, but it does not provide all possible swap possibilities. When traversing the network, a flit initially reaches the head of a FIFO in its starting router, goes through an ECC unit for encoding, travels through the crossbar or bypass bus, the link to its next router, the input port swapper and finally reaches its next FIFO. At each unit along the path, faults are diagnosed and cataloged by two BIST units. If two faults accumulate in a same path, the bypass bus and port swapper provide alternative setups to either avoid one of the faults or move one of the them to a different datapath.

For example, consider three faults: one in the crossbar, another in a link, and a third in the default FIFO for the flit in flight. Since the ECC implementation in Vicis can only correct one of these faults, the crossbar bypass bus and input port swapper must mitigate the remaining two. The bypass bus will be used to avoid the crossbar fault, potentially resulting in a loss of performance. The input port swapper will be used to swap in a fault-free input port to the datapath, moving the single-fault input port to another physical link that does not have any other faults. Thus, full functionality is maintained, even with three faults manifesting in the same datapath.

## Flexible FIFOs

Analysis of a baseline router design informed the selection of our reliable architectural features. Assuming a distribution of faults proportional to the router component area, the largest components — those with the most transistors — are the most susceptible to faults. Thus, we strove to provide additional protection to large components. As shown in Table 5.2, the FIFOs comprise the vast majority of the router, 94% of the baseline router area with 32-flit FIFOs. By comparison, 8-flit FIFOs comprise 80% of the router's area. These results were obtained with a 5-port single-cycle router with a routing table sized for a 3x3 network, synthesized with a 45nm target library. Without any reliability feature, a single fault could cause the entire unit to fail. We thus set out to protect this essential component with a flexible design that can overcome many faults.

| **router component** | **area (percent)** | | |
|---|---|---|---|
| | **FIFO size** | | |
| | **8-flit** | **16-flit** | **32-flit** |
| crossbar | 10.5% | 6.0% | 3.0% |
| decoder | 3.0% | 1.5% | 1.0% |
| FIFO buffers | 80.0% | 89.0% | 94.0% |
| output logic | 3.5% | 2.0% | 1.0% |
| routing table | 3.0% | 1.5% | 1.0% |
| total baseline router | 100% 14,495$\mu m^2$ | 100% 26,173$\mu m^2$ | 100% 49,676$\mu m^2$ |

**Table 5.2 Area of the baseline router** by component, for different FIFO buffer sizes. The FIFOs comprise 80-94% of the baseline router area, and thus are especially susceptible to faults.

FIFOs are comprised of a set of identical registers, and are generally implemented with pointers to determine which register is the head and which is tail. When an item is added to the FIFO, the head pointer is incremented; when an item is removed, the tail pointer is decremented. Thus, the registers are accessed in order, with the first item in being the first item out. The use of identical registers provides an opportunity for a flexible design, with the goal of allowing healthy registers to continue working while skipping faulty ones. Figure 5.8 shows an example of flexible FIFO operation. The "X" in the figure indicates a FIFO register that experienced a fault. In a baseline FIFO, the head and tail pointers will at some point try to make use of this position, causing the entire FIFO to fail. With this flexible FIFO design, the faulty register can be skipped using pointer redirection, and enabling the FIFO to continue operation with one less register.

To reconfigure the access to registers in a FIFO, the head and tail pointers are indexed through a redirection table mapping sequential FIFO positions to reconfigurable FIFO positions. This allows some positions to be skipped, as illustrated in Figure 5.9. Effectively,

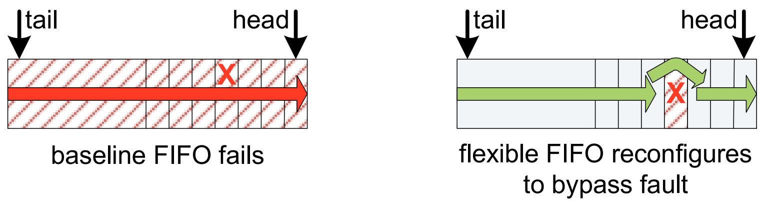


**Figure 5.8 Flexible FIFO design.** A flexible FIFO enables Vicis routers to continue operating correctly when using a partially faulty FIFO. While a normal FIFO can fail with a single error, a flexible FIFO reconfigures around the faulty entry.

this is similar to incrementing (or decrementing) the head (or tail) counter multiple times before accessing the next functional register, thereby skipping over failed registers. The redirection table is indexed by the head and tail pointers, and provides the index to a functional FIFO entry as output. Additionally, the last entry in the table controls the pointer reset signal, thus allowing the system to adapt to the use of smaller FIFOs as the number of faults increases. With this flexible design, a fault in the FIFO causes only a single register position to fail, maintaining the router's functionality as long as at least one functional register remains.

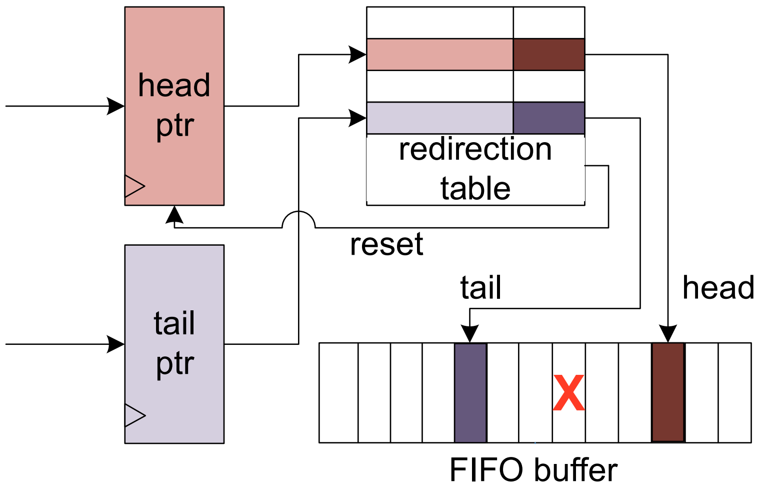


**Figure 5.9 Flexible FIFO buffer logic.** The FIFO registers are indexed by the head and tail pointers (counters) through a redirection table, allowing faulty positions to be skipped, and adjusting for fewer FIFO registers.

## Hard Fault Diagnosis

In order to reconfigure the system, each router must know which of its components contain faults. Furthermore, the use of ECC requires that Vicis knows precisely how many faults are in each part of the datapath. We note that both permanent faults, as well as electrical faults can be diagnosed by Vicis, providing that diagnosis can occur on a faster clock compared to normal operation. Control logic is tested with pattern-based testing and datapath faults are counted using datapath testing, as discussed below.

The reconfiguration process begins when one router broadcasts an error status bit through the network, although not necessarily its location, via an extra wire in each link (we assume that a fault detection solution is in place). The initialized BIST unit then performs a distributed synchronization algorithm with other routers' BIST units, ensuring that each BIST in the network runs all remaining routines in lock-step. After synchronization, each component of each router is diagnosed for faults. The diagnosis step does not rely on information from previous diagnostic phases, or from the detection mechanism, thus all permanent faults are diagnosed (or re-diagnosed), regardless of whether they are responsible for triggering this reconfiguration event, or not. Once all components and routers have been tested, faulty components are disabled and normal operation resumes. Since the BIST units are operational only during reconfiguration, they are power-gated off during normal operation for wear-out protection.

Each functional unit is surrounded by wrapper logic, allowing the BIST to assume control during fault diagnosis. The wrapper simply consists of multiplexers for each input, allowing the unit to switch between normal unit inputs and testing inputs from the BIST. A schematic of this structure is shown in Figure 5.10. Since faults may also manifest in the wrappers themselves, Vicis leverages *interlocked testing* to simultaneously test both the hardware unit, and the wrapper itself. That is, rather than testing the output directly from the module, the BIST unit tests the output after the wrapper mux (as it is indicated by the test flow arrows in Figure 5.10). This allows the BIST to test both the hardware unit as well as the wrapper logic simultaneously.

The information from a complete BIST run is stored in a configuration table, which

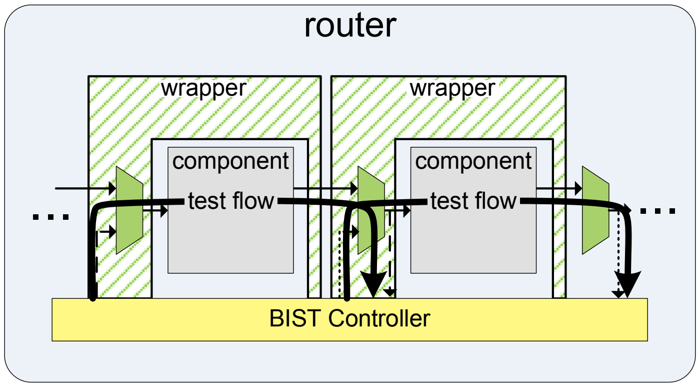


**Figure 5.10 Interlocking router unit wrappers.** Wrapper muxes allow the BIST controller to access each unit. Testing paths are interlocked through two muxes to enable simultaneous testing of the wrapper and the unit under test.

contains two bits for each datapath component: crossbar, inter-router link, input port swapper, and FIFO. Each of these units are represented by two bits to indicate fault free, one fault, and two or more faults. Additionally, a redirection table stores flexible FIFO mapping information (described in Section 5.3.1), which is written directly by the BIST. Fault information is later used by the swapping algorithm. The status of non-datapath (*i.e.* control) units is encoded with one bit indicating functional or faulty. This is determined by a signature match or mismatch. In both cases, Vicis is concerned with the fault status of the component, rather than the exact fault location within the component.

**Datapath Testing**

The datapath test determines the number and location of errors in a router's datapath and in the routing table. Units in the datapath need an exact count of faults for each unit so that the maximum number of errors is not exceeded on any path between two ECC units. The test sends patterns consisting of all 1's or all 0's, looking for bit-flip faults. A custom-designed bit-flip count unit determines if the datapath has zero, one, or more bit flips, obviating the need for multiplexers to inspect each bit individually. Each of the 5 FIFO units in a router reuse the same test, limiting BIST unit overhead. Datapath testing requires about 1,000 total cycles.

**Pattern-Based Testing**

Pattern-based tests are used to test the router's control logic. Vicis uses a linear feedback shift register (LFSR) to generate a number of unique patterns, and a multiple input signature register (MISR) to generate a signature. Each unit type tested with pattern-based testing receives the same sequence of patterns from the LFSR, but each has its own distinct signature. Identical units, such as the decoders, have the same signature. A signature mismatch will flag the corresponding unit as broken. Implementation of the pattern-based test is lightweight due to the simplicity of the LFSR and MISR structures. Pattern-based testing requires approximately 150,000 cycles, and runs 25,000 patterns: this is the dominating factor in overall BIST diagnostic runtime.

**Input Port Swapping**

During the initial evaluation of Vicis, we noticed that often a few faults would disable multiple network links or disconnect important processor nodes. To prevent this, we developed *input port swapping* to consolidate several faults into a single link failure, and to provide additional priority for maintaining connected processors. In order to safely route through the network, the routing algorithm (described in Section 5.4) requires functional bidirectional links. Each link is comprised of two input ports and two output ports, all four of which must be fully functional for the link to be operational. If one of these ports fails, port swapping may be used to maximize functional bidirectional links.

Each input port is comprised of a FIFO buffer and a decode unit, identical for each direction of traffic (see Figure 5.6). Vicis' input port swapper is used to modify which physical links are connected to each input port. For instance, Figure 5.11 illustrates an example where a fault on the South port and a second fault in the adapter FIFO are consolidated, allowing the adapter link to use the former South FIFO. While it would be possible to include an additional output port swapper at the output ports, their small area and consequent low probability of faults did not warrant the area overhead of an additional swapper. On the other hand, the input ports constitute the majority of the total router area, as discussed in Section 5.3.1 and Table 5.2, and therefore are most susceptible to faults. Thus, adding input port swappers provides Vicis with the ability to consolidate the impact of several faults into one, or a few, links.

An example of input port swapper operation is shown in Figure 5.12. The left side of the figure illustrates five routers in a star configuration: the router in the center has a failed input port and the one on the right has a failed output port. Since these two failed ports are on different links, both links would be considered failed and unusable. However, we note that one of the failed ports is an input port, so the connected physical channel can be changed using input port swapping. The port swapping algorithm reconfigures the system to connect the two failed ports, as shown on the right side of Figure 5.12. Thus, in

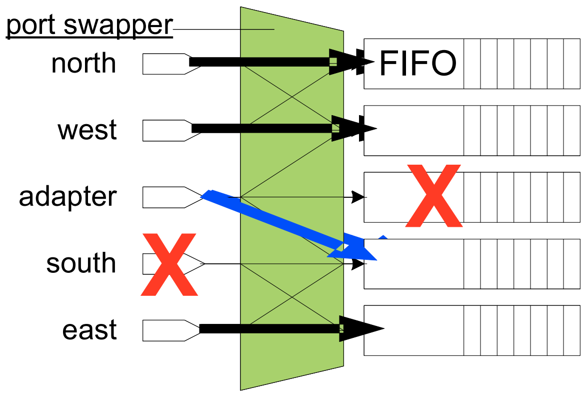


**Figure 5.11 Port swapping unit.** The port swapper allows the FIFOs to be connected to different physical links. The local adapter is equipped with more options to maximize the number of cores connected to the faults network.

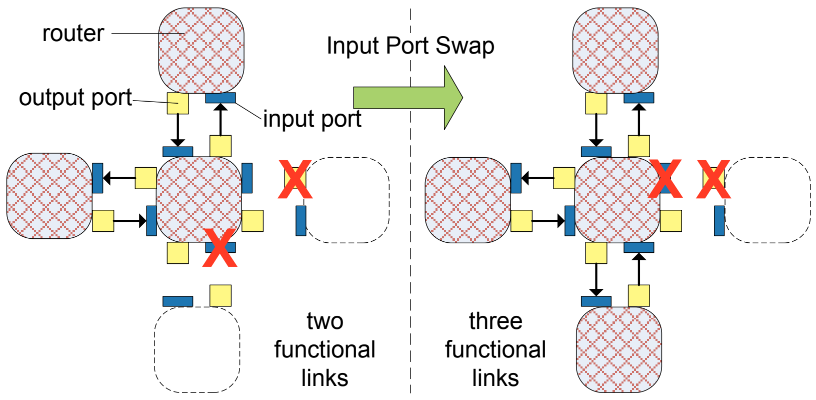


**Figure 5.12 Port swapping example.** Routers connected to the network are hashed in the Figure. On the left, an input port failure on the center router and an output port failure affect two different links. By swapping the failed input port to the link connected to the failed output port, Vicis increases the number of functional links from two to three.

this example Vicis takes advantage of the inherent redundancy of the router to increase the number of functional links for the center router from two to three.

In our implementation of the input port swapper, the link to the local network adapter can be connected to three different input ports, while the other links are able to connect to only two possible input ports. The port swapping algorithm is implemented as a greedy algorithm, taking into account the fault status of connected input ports and FIFOs. Additionally, it considers the number of bit faults along the datapath, so as to avoid connecting paths whose bit-errors exceed that which can be corrected by ECC (see Section 5.3.1). Additionally, it prioritizes the local network adapter link, making sure that it is always connected, if at all possible.

Pseudocode for the port-swapping algorithm is shown in Figure 5.13. The algorithm first eliminates connections that contain control faults and more than one datapath fault. It then connects input ports that have only one viable option. Finally, it selects the highest priority input port among those connected to the FIFO.

After the connection is selected, the port swapping algorithm writes the new port configuration to the router configuration table, a set of registers that keeps track of the current link status. This configuration serves to inform a reliable rerouting algorithm as to which links are functional, enabling it to carry the network reconfiguration forward.

### 5.3.2 Evaluation

We evaluated Vicis using two models: a slower, but more accurate gate-level hardware description model and a faster architectural model. Simulations were conducted by injecting

```
1.  eliminate connections with control faults
2.  eliminate connections with >1 datapath fault
3.  foreach FIFO:
4.    selected direction = NULL
5.    foreach direction connected to this FIFO:
6.      if direction can only connect to this FIFO:
7.        selected direction = this direction
8.        exit loop
9.  if selected direction == NULL
10.   selected direction = top priority
```

**Figure 5.13 Port-swapping algorithm pseudocode.** The port swapping algorithm first disables connections with control faults and too many datapath faults. It then enables connections for which there is only one option. Finally, the algorithm selects the top priority connection among those available.

faults within the system on two different sets of workloads: uniform random traffic and the PARSEC benchmarks [18].

The hardware model was implemented in Verilog HDL, and includes both a 3x3 torus topology and a 3x3 mesh. In both cases, Vicis' reliability enhancements were added to the baseline router. The baseline design is a single cycle, five-port router with one link to a local network adapter, and four links to its neighboring routers. Each router's input is connected to a 32-flit FIFO, which passes through 32-bit data flits. The router was synthesized, and automatically placed and routed to obtain our final simulated netlist. This highly-accurate model was used for simulations, as well to inform the fault model to be used for the architectural simulations.

The architectural model was a custom, cycle-accurate simulator written in C++ with a configuration similar to that of the hardware model. The fast architectural model made it possible to evaluate larger topologies, including an 8x8 mesh and an 8x8 torus. This model also enabled higher testing scalability, making it possible to run longer random tests, and enabling the system to handle the PARSEC benchmarks. The FIFO buffers in the architectural model accommodated 16 flits. Additionally, a statistical model to generate faulty network topologies (described in the next section) was generated, informed by the simulation results of the hardware models.

Test packets were generated at each network-adapter by a random traffic generator which injected traffic to and from all network locations with uniform random probability. Packet length varied from 1 to 10 flits with a uniform random distribution. Additionally, to evaluate correctness, packets injected at each network adapter were checked for arrival at the correct destination with the correct data.

**Fault Tolerance**

We first compared the fault tolerance of a network comprised of Vicis routers to a comparable network implementing triple modular redundancy (TMR). TMR provides probabilistic reliability: since the voter takes the most common signal of the three replicated units, it is possible for only two faults to cause the system to fail. In the worst case, a single fault could cause system failure if it occurred in a clock tree or another non-replicable cell. Unlike BulletProof [33] and other prior work that relies on maintaining total functionality, Vicis is able to tolerate many simultaneous faults, including ones that render entire routers useless, since it is able to route around them. Thus, Vicis can maintain near 100% reliability even for a high number of faults, trading off performance for correctness. A key difference between TMR and Vicis is performance. TMR maintains constant, 100% performance until any component loses one of its redundant versions, after which the entire system fails. On the other hand, Vicis enables gracefully degrading performance as faults accumulate.

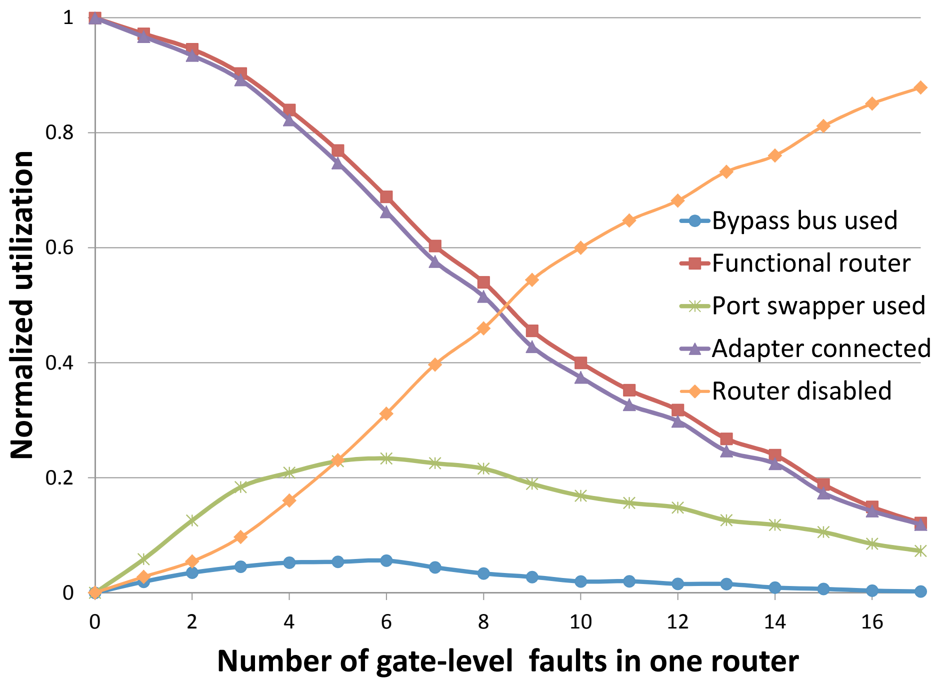


**Figure 5.14 Utilization of reliability features with increasing router faults.** The plot reports the probability of a functional (as well as of a disconnected) router over an increasing number of faults and whether the local network adapter is still functional. Additionally, we indicate which reliability features were used to enable a router to remain functional.

In our next study on the gate-level model, we tested again a gate-level 3x3 torus network, considering eleven different situations with varying simultaneous faults: 1, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100. The case of 100 faults corresponds to approximately one fault for every 2,000 gates. For each number of simultaneous faults, we considered 1,200 different random faulty topologies. Then, we analyzed each router in the topology, considered how many fault it sustained, whether it was still functional and had a functional local network adapter, and what reliability features it was utilizing. The results are shown in Figure 5.14. We note from Figure 5.14 that the input port swapper is very successful at keeping cores connected to the network. As reported in the figure, only a very small fraction of the functioning routers do not have a functional local adapter, as indicated by the closeness of the two curves. The swapper had a high utilization, being used nearly 24% of the time for routers with seven faults.

When considering the utilization of the bypass bus, it was much less often invoked. At seven faults, the crossbar bypass bus was used less than 6% of the time. Two reasons contributed to this: first, the crossbar is relatively small – less than five percent of the total area of the router and thus suffered a smaller incidence of faults. Secondly, the crossbar is protected by both the input port swapper and the ECC mechanism.

### Performance in the Presence of Faults

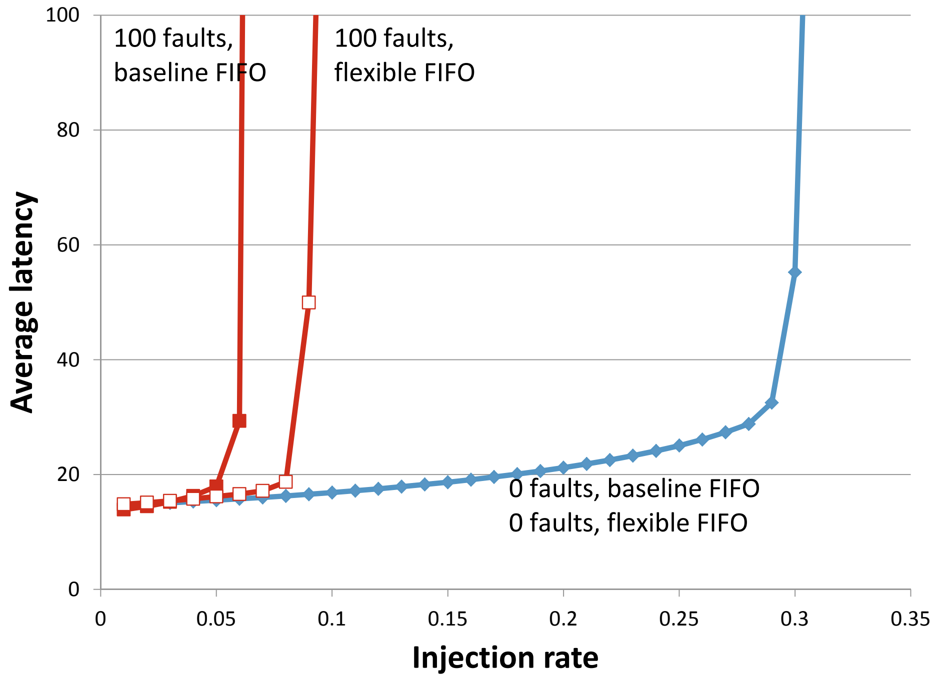


**Figure 5.15 Impact of flexible FIFO design** on Vicis' performance in an 8x8 torus network. The plot indicates that the use of flexible FIFOs allows a Vicis-equipped network to hit the latency wall at higher injection rates.

We examined the effectiveness of the flexible FIFO feature, which primarily affects performance of the network, measuring network latency under different uniform random traffic densities. In these experiments, we simulated an 8x8 torus network with the C++ model. A portion of the faults proportional to the area of the FIFOs were injected directly into the FIFOs, resulting in effectively smaller FIFO buffers after reconfiguration. We then ran simulations injecting 100,000 packets of uniform random traffic. 1,000 different faulty topologies were used for each datapoint. Figure 5.15 shows the average latency curves for the faulty topologies with 100 injected faults. The no-fault case is also shown for reference. First, the chart shows that with no injected faults, the flexible FIFO provides exactly the same performance as the baseline FIFO. As we did earlier, we again note that the fault-injected topology reaches the latency wall sooner. However, with flexible FIFOs, this effect can be mitigated, reaching the latency wall at an injection rate of approximately 0.1, compared to a previous 0.07.

### Area Overhead

The Vicis router physical design was done using automated place and route after synthesizing with Synopsys Design Compiler, targeting a 45nm technology. The baseline router was also designed in this fashion. The resulting Vicis reliable router with 32-flit FIFOs comprised 74,805$\mu m^2$, for an area overhead of 51% compared to the baseline router. This includes both the hardware to implement the routing algorithm, as well as all the architectural features. Table 5.3 shows the overhead of each component in the Vicis router in a 3x3 network. Among the reliability components, the BIST logic is the largest.

| **router component** | **Vicis area ($\mu m^2$)** | **baseline area ($\mu m^2$)** |
|---|---:|---:|
| crossbar | 2,657 | 1,487 |
| decoder | 1,350 | 395 |
| flexible FIFO buffers (32-flit) | 54,303 | 46,706 |
| output logic | 925 | 644 |
| routing table | 585 | 416 |
| misc | 854 | 28 |
| BIST and reconfig. logic | 5,082 | - |
| bypass bus | 201 | - |
| ECC | 1,544 | - |
| port swapper | 603 | - |
| **total** | **74,805$\mu m^2$** | **49,676$\mu m^2$** |
| **overhead** | **51%** | |

**Table 5.3** Area breakdown of Vicis router by component.

By leveraging the redundancy inherent in networks-on-chip, and NoC routers, Vicis can maintain high reliability, while incurring a 51% overhead. A built-in self test at each router diagnoses the number and locations of hard faults. Architecture features including ECC, a crossbar bypass bus and port swapping are then deployed to work around faults. We show that Vicis is able to provide significant area and reliability advantages over TMR, tolerating fault rates of over 1 in 2,000 gates. The Vicis architecture is able to work with a reliable routing algorithm, such as the algorithm described in the next section.

## 5.4 Network Reconfiguration — Ariadne

Permanent transistor faults may cause link and node failures that modify the topology of a NoC. When Vicis (previous section) is unable to mitigate faults within the router architecture, they must be addressed at the network level. Though the initial topology of a NoC is usually regular, after a number of link faults, nodes will be connected through a random irregular topology. Ariadne [8] is a reconfiguration algorithm that is invoked upon a permanent link failure. The name originates from princess Ariadne from Greek mythology, who gave Theseus a ball of thread to help him find his way in the Minotaur's labyrinth. Similarly, our algorithm helps packets find their way in the labyrinth-like topology of a faulty network.

Ariadne is agnostic to the topology, since it includes a discovery phase of the underlying network to update the routing tables with new deadlock-free paths. In a network of N nodes, the reconfiguration procedure consists of N broadcasts, taking up to $N^2$ cycles. The procedure may run in a partially or fully connected network, and guarantees that after $N^2$ cycles every node will know the output port(s) to use when routing to any connected destination. Thereafter, network operation resumes normally.

Ariadne leverages up*/down* routing, a deadlock-free algorithm that can operate on any irregular topology [125]. Up*/down* requires each link to be assigned a direction: *up* or *down*. It then disallows those paths that include traversing a *down* link followed by an *up* link. This way, all cyclic dependencies are broken. Our distributed reconfiguration algorithm assigns a direction to each link, thus allowing up*/down* routing to be applied after reconfiguration. The algorithm then explores the new topology and fills the routing tables with resilient paths connecting all surviving nodes. A key cornerstone of our reconfiguration algorithm is that it is fully distributed, relying on a single atomic broadcast by each node to assign all link directions and to explore the underlying topology. This simple broadcast message scheme makes for a very lightweight hardware implementation, which works in conjunction with the previously described architecture.

### 5.4.1 Historical Approaches to Reliable Routing

While a large number of available reliable routing algorithms suggest fault-free communication is a solved problem, many incur significant restrictions, such as limiting the number and location of faults.

#### Routing algorithms with restricted number of faults

Reliable routing algorithms are capable of routing data packets around faults. However, some limit the number of faults that they can tolerate. For instance, an early work in this area is [35]'s reliable router, which can handle a single node or link failure anywhere in the network. [59, 46] can handle $(n-1)$ faults in an $n$-dimensional mesh and [60] tolerates up to five faults using additional virtual channels. The work in [85] can potentially sustain several faults: the authors provide a backup path around each failed router. As faults accumulate, backup paths form a ring topology. However, the solution fails when additional faults affect the ring network. Other works that tolerate a limited number of faults through the use of additional virtual channels include [64] and [137].

#### Routing algorithms with restricted location of faults

Other routing algorithms are able to accommodate more faults, but restrict their location to specific types of "fault regions". A fault region is a subnetwork of a restricted shape that contains faults, and oftentimes correctly functioning nodes must be disabled to satisfy the constraints of the region. The shape of fault regions may be convex [31, 89, 153, 158, 159, 157], or rectangular [21, 141, 138], and sometimes it is also restricted from including the network boundary [139]. Other solutions require fault regions that are polygons [80], **+**, **L** or **T** shapes [128, 28], or contain no holes [118, 53]. Finally, faults may be limited to datapath components, excluding control logic [91], or to links and crossbars [82].

#### Routing algorithms with unrestricted faults

A number of on-chip proposals tackle the problem of unconstrained faults. uLBDR [119] handles routing for any 2D-mesh topology without the need for routing tables. It adds logic to each input port, which facilitate routing around faulty links. However, this approach requires virtual cut-though routing, requiring the entire packet to be buffered at each router. Other solutions address permanent faults by flooding the network to overcome lost network connections, which incurs high performance overhead [122, 113]. Stochastic approaches [20, 131] provide tolerance to permanent and transient faults by means of a probabilistic broadcast mechanism. Immunet [116] routes packets adaptively towards their destinations, based on buffer availability. If necessary, packets switch to a reserved, escape, virtual channel that guarantees that they will reach their destination and avoid faulty links. This channel is aware of the fault locations and routes deterministically in a ring through every node. Upon reconfiguration, a new ring that connects all surviving nodes is formed with a single broadcast, and all in-transit packets drain out via this ring, before updating the routing tables. While the ring guarantees delivery, it dramatically increases latency, since it must remain active during normal operation to ensure deadlock freedom. Additionally, the design requires three routing tables per router, resulting in high area overhead.

#### Centralized off-chip routing

Off-chip networks, such as clusters, were the first to address the reliability challenge of unconstrained faults. These resilient routing algorithms can be applied to any irregular topology, and include up*/down* (introduced in Autonet) [125], segment-based routing [101], FX routing [121], L-turn [84], and smart-routing [30]. With these approaches, a central node which runs the reconfiguration algorithm in software. First, the surviving topology is communicated to this central location, which can then use this global knowledge of the functional links to compute new routing tables. Finally, the new routing tables are communicated back to each node. While these centralized reconfiguration algorithms can perform powerful optimizations, communicating the global view of the surviving topology to a central node requires expensive, dedicated hardware. By contrast, on-chip solutions must be designed to meet tight on-chip area budgets.

### 5.4.2 Routing Algorithm

The goal of our routing algorithm to avoid nodes and links that are no longer function, reconfiguring the network as faults are discovered at runtime. The reconfiguration algorithm works as follows: each node, in turn, broadcasts a 1-bit reconfiguration flag to all nodes. The first node to broadcast is the node that detected the fault in the network, and it becomes the initiator (root node) of the reconfiguration process. Upon receiving the reconfiguration flag broadcasted from the root node, each node performs the following actions (illustrated in Figure 5.16):

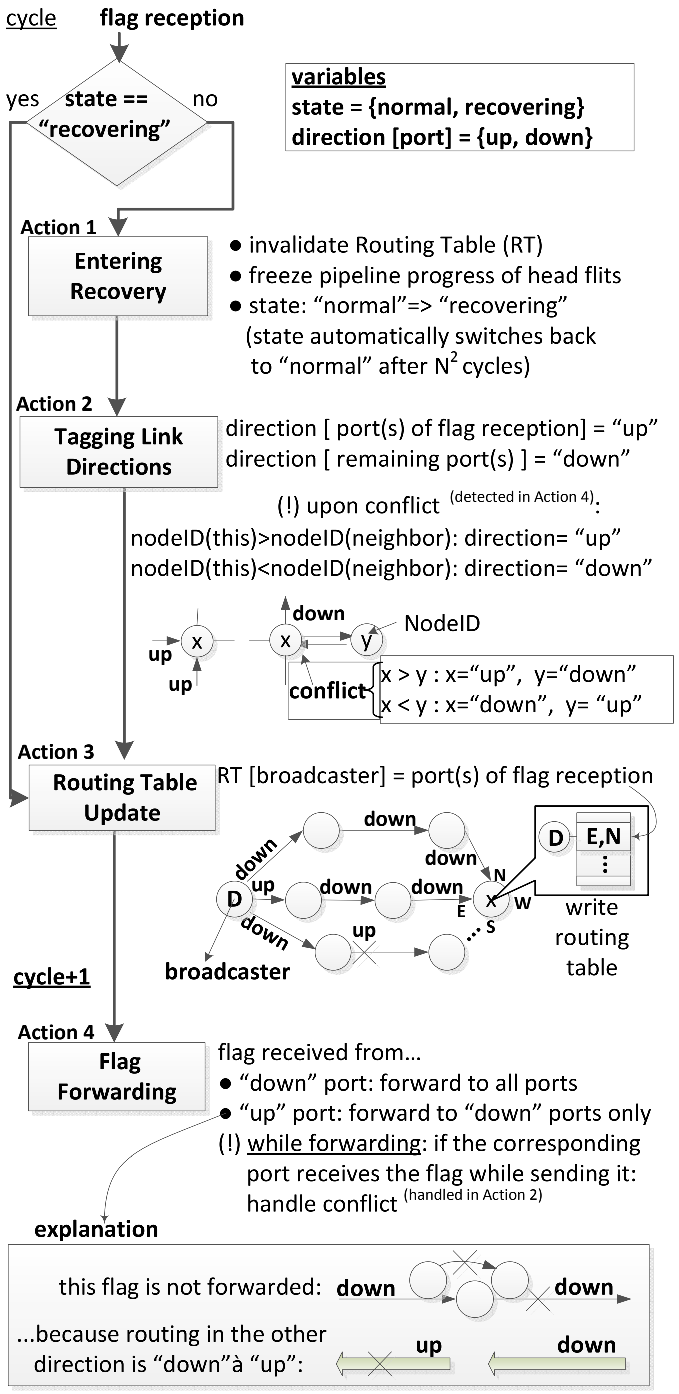

cycle
flag reception
state == "recovering"
yes
no
variables
state = {normal, recovering}
direction [port] = {up, down}
Action 1
Entering Recovery
● invalidate Routing Table (RT)
● freeze pipeline progress of head flits
● state: "normal"=> "recovering"
(state automatically switches back to "normal" after N2 cycles)
Action 2
Tagging Link Directions
direction [ port(s) of flag reception] = "up"
direction [ remaining port(s) ] = "down"
(!) upon conflict (detected in Action 4):
nodeID(this)>nodeID(neighbor): direction= "up"
nodeID(this)<nodeID(neighbor): direction= "down"
up
up
down
NodeID
conflict
x > y : x="up", y="down"
x < y : x="down", y= "up"
Action 3
Routing Table Update
RT [broadcaster] = port(s) of flag reception
down
down
up
down
down
down
up
broadcaster
D
E,N
write routing table
cycle+1
Action 4
Flag Forwarding
flag received from…
● "down" port: forward to all ports
● "up" port: forward to "down" ports only
(!) while forwarding: if the corresponding port receives the flag while sending it: handle conflict (handled in Action 2)
explanation
this flag is not forwarded: down
down
...because routing in the other direction is "down"à "up":
up
down

**Figure 5.16 Ariadne reconfiguration algorithm**: Actions performed upon reception of the reconfiguration flag.

- *Action1. Entering Recovery*: enters recovery mode, invalidates the old routing paths, and freezes head flits.
- *Action2. Tagging Link Directions*: marks all adjacent links as *up* or *down*.

- *Action3. Routing Table Update*: informs the routing table of which port(s) can be used to reach the node that initiated the broadcast.
- *Action4. Flag Forwarding*: forwards the reconfiguration flag to its neighbors.

After the root node has completed its first broadcast, the remaining N-1 nodes in the N-node network broadcast one-by-one. However, during these N-1 broadcasts, the flag recipients will only perform the last two actions described (Action3 and Action4). Each action is described in detail below.

**Action1: Entering Recovery**

Upon receiving the flag for the first time (initial broadcast from the root node), each node invalidates its routing table, freezes the pipeline progress of head flits, and sets its state as "recovering". The state will automatically switch back to "normal" in exactly $N^2$ cycles, since the reconfiguration process is guaranteed to have completed by then. Once the node gets into recovery mode, each subsequent incoming flag will only invoke Actions 3 and 4.

**Action2: Tagging Link Directions**

Once a node gets into recovery mode, up*/down* routing restrictions have to be applied. In up*/down*, links towards the root node (connecting to a node which is closer to the root) are *up* links, while links away from the root are *down* links. Links to a node of equal distance to the root (as the current node) can be either. During the initial broadcast by the root, a port that receives a flag connects to a node that is closer to the root (since the flag arrived there first), thus it is marked as *up*. Similarly, a port that forwards/sends a flag on is marked as *down*. The only conflict occurs when neighboring nodes with equal distance to the root node attempt to send the flag to each other in the same cycle. In this case, each node will receive the flag from a port while trying to send it to the same port. When this happens, up*/down* suggests that the direction of the corresponding link can be either *up* or *down*, so we set it based on the statically assigned nodeIDs: the node with the higher nodeID will mark the link as *up*; the other node will mark the link as *down* (shown in the Action2 diagram of Figure 5.16).

After this assignment, all the *down*$\rightarrow$*up* turns are disabled. This restriction inherits the deadlock freedom of up*/down*. Though a number of paths are disabled, there is always at least one deadlock-free path that connects all nodes reachable from the root node. That is because the minimal route from any node to the root (*up*) and from there to any destination node (*down*) is always available.

### Action3: Routing Table Update

During each broadcast, the broadcasting node communicates to other nodes how it can be reached. When nodes receive the broadcasted flag, they record the ports where the flag was received from in their own routing table, to learn how the broadcasting node can be reached. This requires the broadcasts to spread via enabled turns only, so that the recipient of the flag can follow the opposite path to reach the broadcasting node. The third Action of Figure 5.16 shows that the flag from node D's broadcast arrives to the current node via its North and East ports, thus the node marks in its routing table that these ports should be taken to route to D.

### Action4: Flag Forwarding

In the next cycle, the node broadcasts the flag only to those port(s) from which it did not receive a flag earlier and that correspond to enabled turns (a flag received from *up* link(s) is never broadcasted to *up* links, because this will enable a routing path with a *down*$\rightarrow$*up* turn, as shown in the last diagram of Figure 5.16). Since forwarding a flag takes a single cycle, each broadcast will deterministically complete in at most N cycles. Each broadcast is bounded to N cycles. The worst case scenario occurs when all nodes are connected in an open ring, and the longest broadcast from one end to the other requires N-1 cycles.

### Completion of Reconfiguration

Reconfiguration is deterministically completed within $N^2$ cycles since each node broadcasts once, and each broadcast takes at most N cycles. During this time, all routing tables are updated with resilient paths to any connected node, thus communication may be resumed and all nodes set their state back to "normal". After this point, any node can initiate a new broadcast upon detection of a link failure and invoke the reconfiguration process again. The head flits can now proceed in the pipeline, but since routes have changed, they must restart from the route compute stage to find an alternative port that leads to the desired destination.

### 5.4.3 Timing and Synchronization

We now detail the timing of the reconfiguration: How does each node know when to broadcast so that there is no overlap between broadcasts? How does the recipient of a broadcast know the broadcasting node, since the only data broadcasted is a 1-bit flag? How do nodes know when the reconfiguration is completed? If two nodes concurrently detect a new fault, can they both become roots and initiate a broadcast? This section deals with these timing issues by introducing the notion of atomic broadcasts, where the cycle number indicates the ID of the broadcasting node.

#### Atomic Broadcasts

The idea of atomic broadcasts is to correlate the cycle number at which a broadcast is initiated to the broadcasting node's nodeID. Using the cycle number as a common reference point, all nodes are assigned different cycles for broadcasting, during which the remaining nodes are prevented from broadcasting for a window of N cycles (every broadcast is guaranteed to complete in N cycles, where N is the number of nodes). Each node will have to wait for its slot to broadcast, so it is guaranteed that during that slot no other node would be broadcasting, causing a collision. N slots need to be provided for all N nodes to broadcast, with slots looping around throughout execution. In other words, node(X)'s Nth-cycle slot is always followed by node((X+1)modN)'s Nth-cycle slot. The idea is similar to time-division multiplexing, where a number of signals physically take turns to transfer data on the same communication channel. Once the root node broadcasts, every node will deterministically broadcast during its first available slot; assuming that R is the root, nodes will broadcast in the following order: R, R+1, ..., N-1, 0, 1, ..., R-1. Since N slots of N-cycles are required for the reconfiguration to complete, reconfiguration requires precisely $N^2$ cycles. Thus, all nodes can resume operation $N^2$ cycles after receiving the initial broadcast, since by that time the process is guaranteed to have completed.

We note that if more than one node concurrently detects a fault, only the one who first receives a broadcast slot will become the root. The others will resign from becoming the root once they receive the root's broadcast, since they become aware that reconfiguration is already taking place. Our atomic synchronization ensures that no two nodes will ever simultaneously become the root. Once reconfiguration has been initiated, Ariadne will consider all faults in the network (including faults that other nodes have detected), independently of the node that managed to become root. Also, if a node is disconnected, it will not receive the root's broadcast, and will thus remain silent during its broadcast slot. Other nodes will not fill the corresponding routing table entry, marking this node as unreachable.

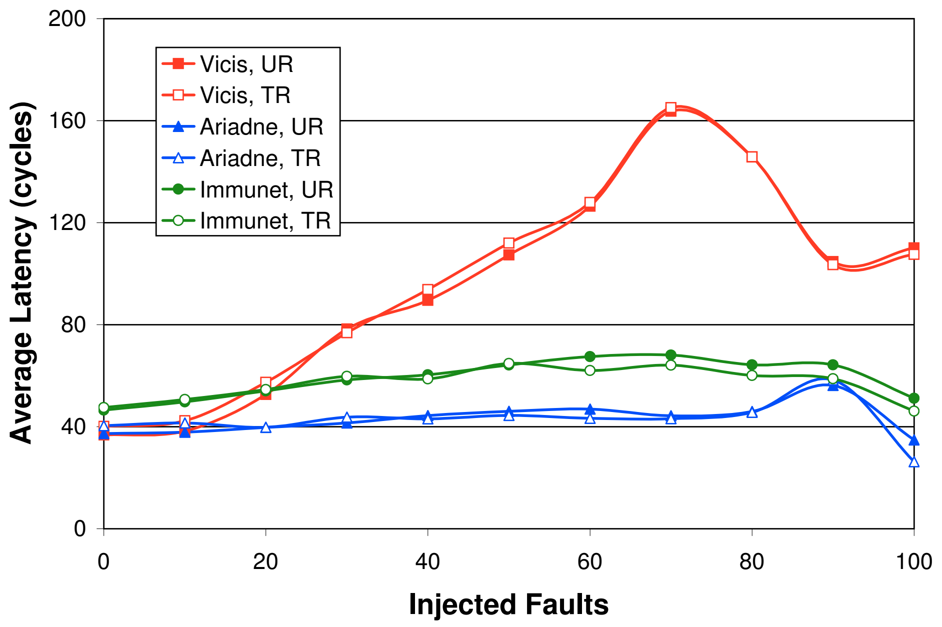


**Figure 5.17 Zero load latency with synthetic traffic.** The average latency with a fault injection rate of 0.01, plotted for varying number of faults injected. Ariadne is consistently the lowest.

### 5.4.4 Evaluation

We consider two metrics to measure the performance of Ariadne on a faulty network: average latency and throughput. Latency is defined as the delay experienced by a packet from source to destination, while throughput demonstrates the rate of packets delivered (per cycle) in the entire network. First, we look at the zero-load latency for each of the three routing algorithms (Vicis [51], Immunet [116], and Ariadne), reflecting the steady-state latency of a lightly loaded network (0.01 flits injected per cycle per node, well below saturation). Each data point in Figure 5.17 is the average zero-load latency of 100 different topologies with the same number of faults.

We note that Ariadne's latency is consistently the lowest, at 43 cycles on average, compared to 58 cycles for Immunet and 97 cycles for Vicis. At 50 faults, the difference increases, with Ariadne showing a 43% latency improvement over Immunet and a 142% improvement over Vicis. Moreover, we note that the latency trend is strongly dependent on the algorithm, but not greatly affected by the traffic type, as shown by the closeness of the lines for each given algorithm. Vicis shows some interesting behavior here, increasing in latency as the number of faults increases. Upon further investigation, we found that this was caused by the occasional deadlocks encountered by the Vicis algorithm, which in turn trigger a 1,000-cycle timeout before dropping the deadlocked packets. Ariadne maintains a reasonable constant latency, outperforming Immunet, since all of its virtual channels can route adaptively. In contrast, Immunet has one escape virtual channel restricted to route de-

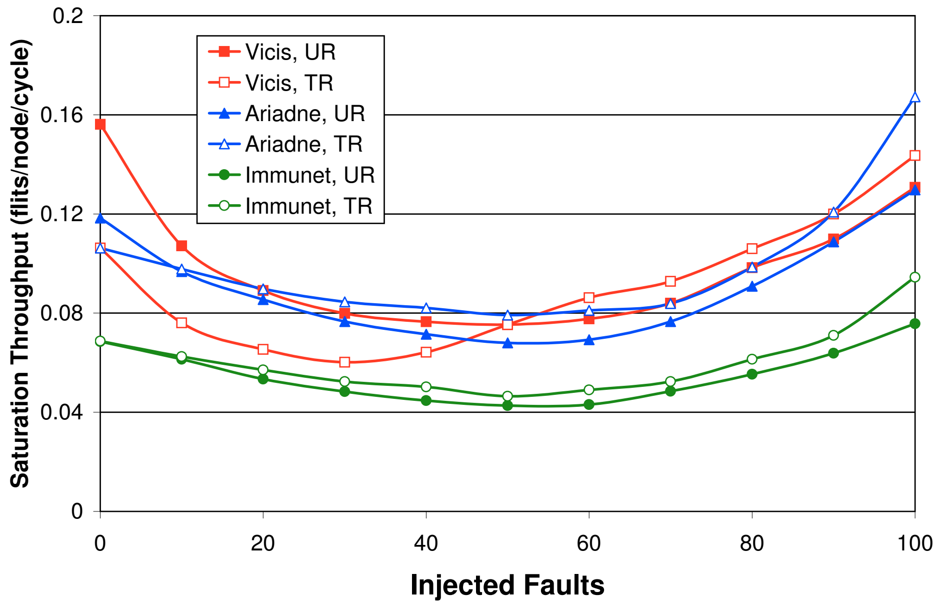


**Figure 5.18 Saturation Throughput with synthetic traffic** (flits/node/cycle) reaches a minimum at approximately 50 faults for all three algorithms. Then, it increases since more packets must be dropped as more node pairs become disconnected.

terministically in a high-latency ring that goes through all surviving nodes two times (both directions) on average, independently of a packet's destination.

Figure 5.18 plots the saturation throughput as a function of faults. We used numerical analysis to find the throughput value, from simulation results of various injection rates, within a precision of 0.01. For each number of faults, we performed this calculation for 100 different configurations. Notice that for fault rates up to about 50, saturation throughput decreases as the number of faults increases, as it can be expected, since the number of available paths decreases. This effect changes as the number of faults increases past 50, when throughput begins to increase. Upon closer investigation, we found that this was due to an increasing number of dropped packets; when the network is not fully connected, packets destined to disconnected destinations must be dropped. With the network being partitioned, packets are either dropped or routed a few hops within small subnetwork partitions, so overall throughput actually improves. For the same reason, the type of traffic does not critically affect the saturation throughput at high number of faults, since packets are restricted to route only within these subnetworks. Note that the impact of traffic type on saturation throughput is not strong for Ariadne and Immunet. However, since Vicis is based on the turn model, which has naturally a higher saturation point, it more evenly spreads random traffic, particularly in few-faults situations, where the routing algorithm is closer to the turn model. We note that although based on the turn model, Vicis is deterministic, and uses a heuristic that chooses a minimal subset of available turns to reduce the probability of deadlock occurrence.

Ariadne, an agnostic reconfiguration algorithm for NoCs, is capable of circumventing large numbers of simultaneous faults, and able to handle unreliable future silicon technologies. It guarantees that if a path between two nodes exists, the reconfiguration algorithm will enable at least one deadlock-free path between them. Ariadne is implemented in a fully distributed mode, since nodes coordinate to explore the surviving topology, resulting in very simple hardware and low complexity. At 1.97% area overhead, Ariadne is a parsimonious solution for many-core processor designs of the future, enabling a trade-off between performance and reliable functionality on unreliable silicon.

## 5.5 Data Recovery — Drain

Following network routing protocol reconfiguration, a reliable system must recover the system state before computation can resume on the reconfigured system. In heavily fault-injected topologies, some nodes may become disconnected as a result of faults. The data from these nodes should be recovered following the occurrence of each fault.

This portion of runtime solution presents a **D**istributed **R**ecovery **A**rchitecture for **I**naccessible **N**odes (Drain [39]) which uses dedicated cache-to-cache emergency links to recover data from disconnected nodes. Drain guarantees that processor architectural state and dirty cache data can be safely sent to memory via dedicated cache-to-cache emergency links, tolerating any number of disconnected nodes. This feature, in combination with a resilient NoC, guarantees full system recovery in the face of unlimited network faults. Unlike checkpointing approaches, Drain does not involve any runtime performance overhead during normal operation, while the additional recovery time upon a network failure is only a few milliseconds (assuming 1GHz clock). Our solution is realized with minimal hardware modifications, resulting in an area overhead of a few thousand gates. It is also flexible and can work with any underlying architecture, including homogeneous and heterogeneous chip multiprocessors, multiple shared or private levels of caching, any network topology, any resilient routing algorithm that guarantees connectivity upon link failures and any coherence protocol.

Modern designs already implement a variety of resiliency mechanisms to protect individual caches, for example error correcting codes (ECC). ECC is able to tolerate bit-errors, often a single bit, in datapath elements. Reconfigurable structures [86] are another common solution, where extra cache lines are added at design time. Post fabrication, these extra lines are configured as substitutes for faulty lines. Another reliability solution is triple modular

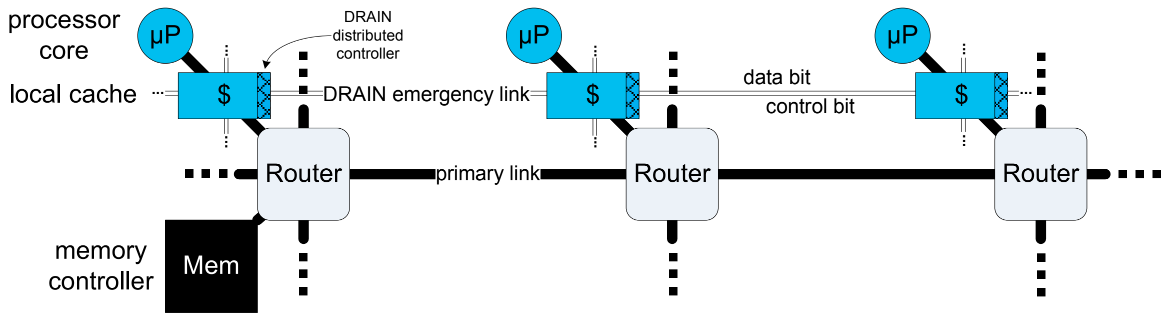


**Figure 5.19 Drain-enabled system**. An existing CMP is augmented with emergency links and small controllers to transfer the data from caches that have become disconnected.

redundancy, typically used to protect control logic by triplicating it and voting among the three outputs. This solution is very expensive in terms of area and power overhead, and furthermore provides only probabilistic reliability guarantees. While current cache reliability approaches preserve the correctness of data, they are localized solutions and can not handle faults due to lack of connectivity between multiple caches in CMPs.

### 5.5.1 Recovery Hardware

Our solution augments a CMP with a set of distributed controllers and dedicated “emergency links”, network links connecting nearby caches to each other. When an error is detected, Drain suspends execution, flushes dirty cache data and architectural state to memory, and then allows the OS to re-map the address space. At this point, normal operation can resume, with all data prior to the error recovered.

Figure 5.19 shows the high-level modifications to a CMP architecture that are required to implement Drain. The baseline system consists of nodes, each with a processor core, local cache and router, with some nodes also connected to memory controllers. Nodes communicate through a flexible interconnect, in this case, a network on chip. Drain adds 2-bit network links that connect neighboring caches together, with one bit for data and one for control. The emergency links are used when a node or a subnetwork becomes disconnected, in order to recover cache data and architectural state that would otherwise be lost. The emergency links (thin double lines in Figure 5.19) are used only during recovery, while normal operation progresses through the primary NoC links (thick lines in Figure 5.19).

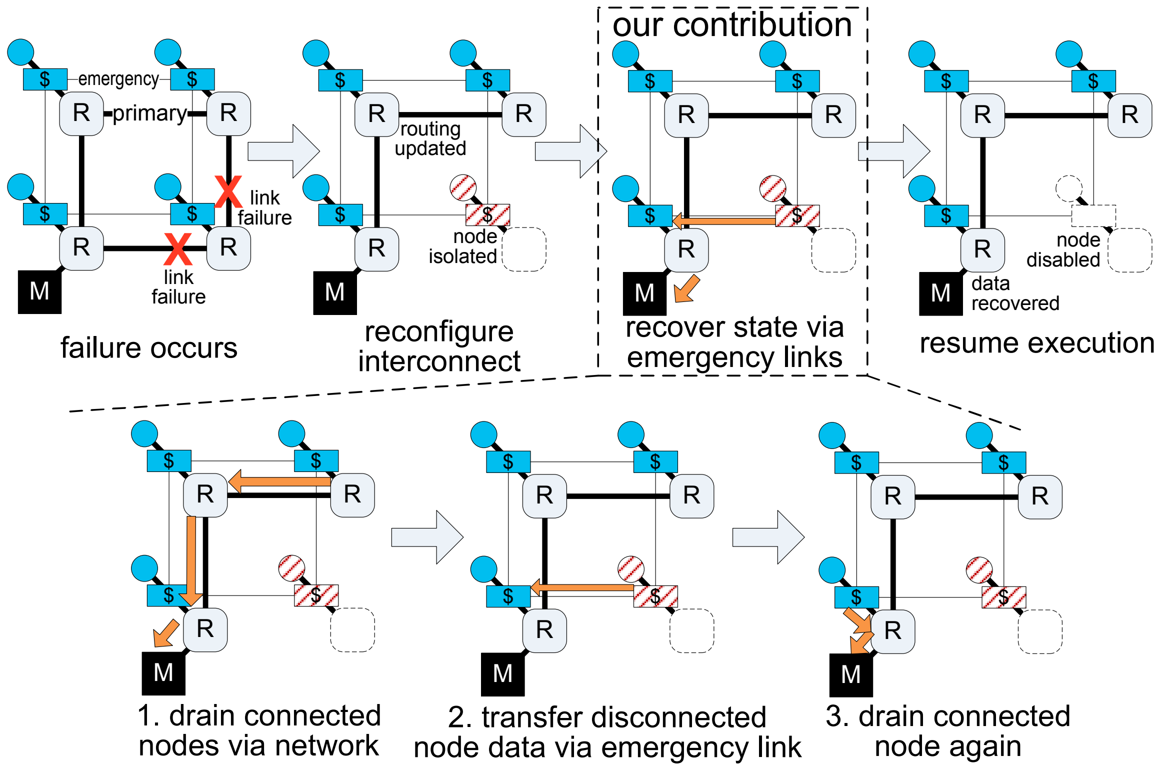


**Figure 5.20** **Drain system operation** recovers state when an error occurs, allowing the system to be reconfigured and resumed without losing information. The Drain algorithm operates in three major steps, first draining connected nodes via the existing interconnect. Next, disconnected nodes' data is transferred to a nearby connected node, and from there transferred to main memory.

## Recovery Overview

The error recovery process begins with the detection of an error, as shown in Figure 5.20. The fault can be detected by either hardware or software, typically handled by the underlying reliable network. Error detection has been extensively researched [19, 33, 66] and it is not the focus of this work. When an error is detected that renders all links to a node inoperable, a special interrupt designed for Drain is issued to the processors, which causes the processors to save all state required to resume the running processes: typically stored in a Process Control Block (PCB). The PCB includes the PID, architectural registers (including the program counter, load/store queue, stack, *etc*.), address space, I/O port permissions, stack pointers, *etc*. During reconfiguration, each processor sends the PCB of its running process to memory (either via the primary or emergency links).

Next, the network reconfigures, reestablishing communication among the processing elements that are still connected. A variety of schemes are possible here, for example [52, 101, 125], as long as the interconnect enables the communication of functioning units and avoids the loss of in-transit packets. The newly reconfigured network is reflected in the figure by the disabled primary links, which leads to the occurrence of a newly isolated node, shown by the missing router (dashed) connected to a local cache and processor. While reconfiguration allows a communication subsystem to resume message transfers, it does not guarantee that disconnected processors and caches will be able to recover their state.

### Emergency link transfer

Next, Drain transfers data via the emergency links from isolated nodes to a nearby functioning cache: the details of this process are shown in the bottom row of Figure 5.20. First, the nodes that remain connected to main memory following network reconfiguration are informed by the reliable network of their connectivity via the emergency link control bit. These nodes drain their state via the main network (step 1 in the bottom row of Figure 5.20). Drain interacts with the system's coherence mechanism to transfer the address and data of only those cache lines that are dirty to main memory. For example, in the write-invalidate cache coherence protocol with `MOESI` states [34], lines in the `M` (modified) or `O` (owned) state may be dirty, including lines in a transition state. When a destination cache receives a cache line, it writes the line to the appropriate address, marking it as dirty in the destination cache. The processor's architectural state is then transferred to a (now evicted) cache line, enabling Drain to reuse the emergency links to transfer the state. The registers, PC, and store buffers are included in the transferred state, while speculative state such as branch tables are not included. Additionally, all flags and state bits that will be necessary to recover the state of the processor are stored. Finally, this data is drained via the emergency links.

Once all the connected nodes have transferred their architectural state and emptied their caches, they advertise to each neighbor in turn that they are ready to receive data over the emergency link control bit. In step 2, cache data and state from the disconnected node are transferred one bit at a time over the emergency link, facilitated by the distributed Drain controllers (shown by the hashed portion of the cache in Figure 5.19). A target node, which accepts a transfer from the disconnected node, ceases to advertise that it is ready to receive as the transfer begins. Upon completing the transfer, the target node drains its cache contents again to main memory (step 3 of Figure 5.20). At this point, all dirty cache data and architectural state in the entire CMP system has been transferred to main memory, allowing the operating system to remap addresses and processes (PCBs) to the surviving processor nodes. The reliable network signals the processors to wake up. A processor waking from a Drain interrupt will generate a memory request to retrieve a PCB and resume normal operation. This enables the OS to flexibly re-assign PCBs upon resume. The resume process proceeds in a similar fashion as a processor switching to a new process during a context switch. Finally, normal execution can resume on the newly recovered system.

### 5.5.2 Recovery Algorithm

Triggered by a detected fault, Drain's distributed recovery algorithm ensures that all cache data and architectural state reaches main memory. CMP nodes or subnetworks that have become disconnected as a result of a fault are drained using a set of emergency links, one bit links that connect nodes to their nearest neighbors. A disconnected node's cache contents and architectural state are transferred to the nearest connected node via these emergency links. At this time, the target connected node, which contains the data of the disconnected node's cache, transfers it to main memory via the primary network. Leveraging both the emergency links as well as the correctly functioning portion of the primary interconnect, the algorithm finds the most efficient combination of both to deliver all data and state to memory.

To enable the operating system to remap the workload onto the reduced system resources, all caches in the system must be drained, even those that remain functional and connected. The first step in the Drain algorithm is to transfer data in the nodes connected to memory via primary links. Dirty cache lines are first written back, followed by architectural state. For architectural state transfer, each register or state element is first copied to a cache line in the recently drained cache. Finally, the drained cache advertises that it has completed this step via the control bit of the emergency link. Thus, other caches become able to use this cache space as target node in transferring the contents of a disconnected node.

If the node in question is not connected to main memory via a route along the primary links, emergency links are used to recover cache contents and architectural state. The emergency links operate by copying the lines of the disconnected cache to a nearby neighbor, as detailed in Figure 5.21. First, the disconnected node scans the control bits of the emergency links to its neighbors (Figure 5.21 lines 2-5), attempting to find a neighbor that has a route to main memory. If available, it selects the first such neighbor that advertises an empty cache, using it as the target for transferring exclusive cache data and architectural state via the emergency links. If no such neighbor exists, as in the case of a large, disconnected subnetwork, then the emergency links transfer through intermediate nodes to reach one that is connected. In this case, a node will try to find a target cache that is connected to main memory and fail. The fallback procedure is to do a cache-to-cache transfer towards

```
1 drain_via_emergency_link(this_node)
2   while target_cache not found
3     for each neighbor
4       if neighbor is connected and empty
5         target_cache = neighbor; break;
6     if target_cache not found
7       for each neighbor
8         if neighbor is toward boundary and empty
9           target_cache = neighbor; break;

10  for each dirty cache_line
11    copy_line to target_cache
12  for each register/state_element
13    copy_register to target_cache
```

**Figure 5.21 Drain uses an emergency link** for disconnected nodes, scanning its neighbors for a connected node to which it transfers dirty cache lines and architectural state via the emergency link.

the outer boundary of the disconnected subnetwork, which is ascertained from the routing logic in the reliable network (Figure 5.21 lines 6-9). The transfer is described by lines 10-11 of Figure 5.21, where the emergency link controller iterates over each dirty cache line, transferring the dirty data, along with its address. Then, architectural state is transferred to the target cache as shown in lines 12-13. Finally, the target node, upon receiving the data, will be analyzed again to find the next best step towards main memory. If the target node has no direct route to memory, data will traverse several other nodes before reaching memory via primary and emergency links.

### 5.5.3 Evaluation

We evaluated an implementation of Drain on a 64-node architectural simulator, injecting a variety of faults in the underlying network-on-chip. We used the SPLASH2 benchmarks as our workloads to evaluate performance during a system recovery.

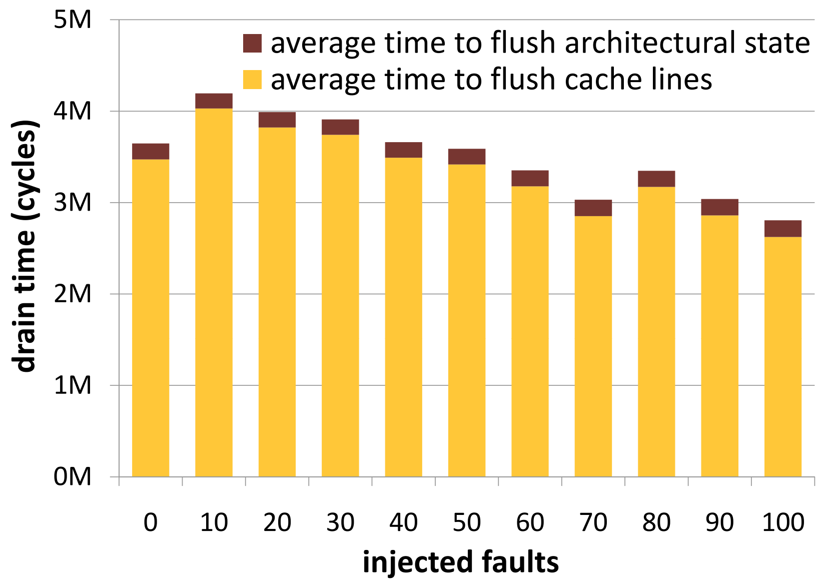
We implemented Drain in the memory model (Ruby) of the Wisconsin Multifacet GEMS simulator [95]. We simulated full system recovery by combining Drain with an on-chip implementation of the up*/down* [125] resilient routing algorithm, which was implemented within the Garnet network model [7]. The parameters of our experimental setup are shown in Table 5.4. To evaluate our scheme under real workloads, we used SPLASH2 benchmark [130] traces, injecting an additional fault and triggering a recovery after one million instructions. We evaluated full system recovery for both fully connected and partially connected networks.

| | |
|---|---|
| network topology | 8x8 2D mesh, 5-stage wormhole routers |
| L1 configuration | 4-way, 64KB, 64-byte block, 2-3 cycle latency |
| L1 functionality | private, unified, write-back |
| L2 configuration | 4-way, 1MB, 64-byte block, 4-6 cycle latency |
| L2 functionality | private, unified, inclusive, write-back |
| memory | 4GB, 160-cycle access latency |
| coherence protocol | MOESI states, directory |

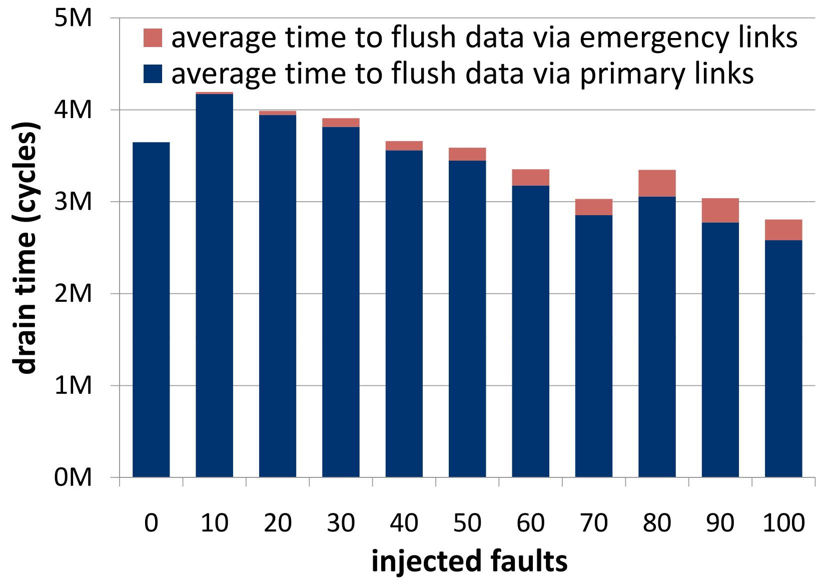
**Table 5.4 Experimental setup** for a 64-node system.

To evaluate our solution with a variable number of disconnected nodes, we generated 100 faulty topologies, corresponding to 100 different random seeds, for various numbers of network faults (0, 10, 20, ..., 100) for a total of 1,100 topologies. We used a fault model that uniformly injects gate-level faults in the router logic, similar to the fault model of Vicis [52]. Faults were not injected in the emergency links, since these are power-gated off during normal operation, allowing them to avoid wear-out related faults. We then identified the subnetwork connected to the main memory and marked all of its nodes as connected; the remaining nodes were marked as disconnected. Next, we mapped the address space to the connected nodes and simulated our benchmarks. After 1 million instructions, when the system is warmed up, we injected an additional fault, resulting in a Drain invocation. This single fault models the expected real-world scenario, since it is unlikely that more than one permanent fault occurs at the same time: the frequency of fault occurrence is typically measured in the order of days or months. Consequently, our performance measurements correspond to the worst case scenario: applications penalized for the rare occurrence of a permanent fault. We note that some fault-injected topologies result in disconnected memory controllers, making it impossible for the cores to reach main memory. The probability of these cases ranged from less than 1% with 10 faults, to almost 88% with 100 faults. Since topologies without a connection to memory are not useful even when reconfigured, we considered only those topologies that remained connected to main memory.

Figures 5.22a and 5.22b show the time required for Drain to flush all connected caches with an increasing number of faults. Each datapoint is an average over all SPLASH benchmarks, where each benchmark was evaluated over 100 randomly generated faulty topologies. We observed that for most topologies, the time to recover ranges from 3 to 4 million cycles (just a few milliseconds at 1GHz clock), a reasonable penalty for a rare event. The drain time can be partitioned based on the data that is drained (cache lines or architectural state, Figure 5.22a), or the communication medium (emergency links or on-chip network, Figure 5.22b). Figure 5.22b shows that the majority of the time is consumed in



**(a)** Drain time by cache and architectural state



**(b)** Drain time by emergency and primary links

**Figure 5.22 Cycles to drain the entire network**, averaged over all benchmarks. The time to drain architectural state is a function only of the number of cores. The drain time decreases as the number of faults increases, reflecting the decreasing surviving network size due to faults.

the on-chip network, because upon recovery, all caches flush their data concurrently, resulting in high network traffic and congestion. In the same figure, we observe that, although the total time to drain data decreases with increasing faults, the time to flush data via the emergency links increases. Upon further investigation, we found that for networks with a large number of faults, an additional fault is likely to disconnect more than one node and/or break the network down to several partitions; consequently, more lines have to be copied from disconnected nodes to the connected subnetwork using emergency links.

Figures 5.22a and 5.22b both reflect that as faults increase, the number of nodes connected before the drain occurs decreases. During the drain, both connected and newly disconnected caches are drained concurrently.

### Area Overhead

Drain requires minimal hardware modifications. The basic hardware structures that need to be added to each cache to implement the emergency links are a few counters and multiplexers. Table 5.5 shows the logic to implement these structures, on a sample L1-L2 private cache system, together with their 2-input AND equivalent gate count. The total overhead per node adds to 4,952 gates (2,466 gates for L1 and 2,486 gates for L2), negligible compared to a core's gate count, which is on the order of hundreds of millions of gates.

| | **L1 cache** (64KB 4-way) | **L2 cache** (1MB 4-way) |
|---|---|---|
| **row counter** | 40 gates | 60 gates |
| **way counter** | 10 gates | 10 gates |
| **data multiplex** | 1,632 gates | 1,632 gates |
| **index multiplex** | 24 gates | 36 gates |
| **way multiplex** | 6 gates | 6 gates |
| **tag multiplex** | 54 gates | 42 gates |
| **serial-to-parallel** | 350 gates | 350 gates |
| **parallel-to-serial** | 350 gates | 350 gates |
| **TOTAL** | 2,466 gates | 2,486 gates |
| | **4,952** gates grand total | |

**Table 5.5 Additional gates** required at each node to implement Drain. Each node requires a total of 4,952 additional gates.

Drain is a recovery mechanism targeting large scale CMPs. Drain augments a CMP interconnect architecture with emergency links that facilitate the recovery of dirty cache data and architectural state in the event that a node or subnetwork becomes disconnected. Our experimental results show that Drain is able to recover data from disconnected nodes in any faulty network configuration, even those where aggressive faults cause network partitioning. It is able to provide complete state recovery for an entire 64-node CMP within several milliseconds and it incurs very low area overhead of 4,952 gates at each node. Thus, we demonstrate that Drain is an effective recovery solution for large scale CMP systems.

## 5.6 Summary

We have presented a family of compatible solutions to address transistor faults that occur at runtime due to wear-out. In addition to covering hard-faults, our runtime approach can also be applied to electrical failures occurring at runtime. Electrical failures, which cause a circuit to fail to meet specification under certain operating conditions, are addressed in a similar manner to hard faults. Our solution has four parts: fault detection, diagnosis, reconfiguration and recovery.

**Detection** is the first step to a fault-tolerance. We reuse the flexible hardware provided by BiPeD to monitor internal router protocols for faults. When a fault is detected, it triggers the diagnosis mechanism.

**Diagnosis** is carried out by Vicis, a reliable network-on-chip that is able to tolerate many faults in both the router components and the reliability components themselves. It maintains correct operation in the face of faults, trading off performance for correctness. As the number of failures increases, Vicis mitigates errors by reconfiguring the router architecture. By leveraging the redundancy inherent in networks-on-chip, and NoC routers, Vicis tolerates fault rates of over 1 in 2,000 gates.

**Reconfiguration** of the communication interconnect routing is handled by Ariadne, which circumvents large numbers of simultaneous faults. It utilizes up*/down* for high performance and deadlock-free routing in irregular network topologies that result from large numbers of faults, and offers performance gains ranging from 40% to 140% (for 50 faults) during normal operation, compared to state-of-the-art fault tolerant solutions. It guarantees that if a path between two nodes exists, the reconfiguration algorithm will enable at least one deadlock-free path between them. Ariadne is implemented in a fully distributed mode, since nodes coordinate to explore the surviving topology, resulting in very simple hardware and low complexity.

**Recovery** of lost data is provided by Drain, a recovery mechanism targeting large scale CMPs. Drain augments a CMP interconnect architecture with emergency links that facilitate the recovery of dirty cache data and architectural state in the event that a node or subnetwork becomes disconnected. We show that Drain is able to recover data from disconnected nodes in any faulty network configuration, even those where aggressive failures cause network partitioning. Taken together, this family of solutions provides resilience to transistor faults and electrical failures that occur in the field.

# Chapter 6

# BiPeD: Bridging the Phases of Verification

Pre-silicon verification, post-silicon validation and runtime verification methodologies are very different, traditionally sharing little information or hardware between them. As a result, the diagnosis and debugging of post-silicon failures is very much an ad-hoc and time-consuming task that is largely unable to leverage the vast body of design knowledge available in pre-silicon. Furthermore, runtime solutions require significant hardware overheads, and are typically unable to make use of post-silicon validation hardware.

Previous chapters have described how the solutions in this dissertation fit into the BiPeD framework. This chapter presents a holistic view of BiPeD, applying the pre-silicon verification techniques of Inferno (Section 3.2) to understand the design, uses this information to find bugs during post-silicon validation, and provides flexible hardware that can be leveraged for runtime verification. We also quantify the effectiveness of BiPeD in detecting bugs.

## 6.1 BiPeD Operation

During pre-silicon verification, BiPeD learns the correct behavior of a design's communication patterns. In post-silicon, this knowledge is used to detect errors by means of a flexible hardware unit. When an error is detected, bug reproduction is not necessary: a diagnosis software algorithm analyzes information stored in the hardware unit to provide a wide range of debugging information. Finally, BiPeD is implemented with flexible hardware that can be reused at runtime.

BiPeD identifies the exact time and location of bugs. Our approach accelerates debugging by providing a broad set of debugging information: modules and signals involved and a history of the activity that preceded the failure, presented as intuitive, high-level transactions.

We address failures that occur in the behavioral protocols governing the communication among a system's components via its interfaces. The root causes of the errors that we strive to detect and diagnose can be functional bugs, electrical failures and/or manufacturing faults.

The goal of the framework presented in this chapter is to learn the correct behavior of a system's protocols during high-observability, low-speed pre-silicon verification, and then detect violations of these protocols during high-speed, low-observability post-silicon validation. Our work targets difficult post-silicon bugs that manifest in the inter-block interactions of a complex chip, automatically determining their time of manifestation and providing a complete and detailed set of intuitive of debugging information related to the system's activity preceding and leading up to the failure. We completed the dissertation work with a novel solution that bridges pre- and post-silicon verification by leveraging the high observability typical of pre-silicon verification to learn the protocols that define the interactions among hardware blocks. During post-silicon validation, these protocols are loaded into a small and flexible hardware unit, which monitors the interface(s) at runtime on the silicon prototype. When an error manifests, the hardware detector provides the recent history of the protocol-level activity observed on the interface that flagged the bug to a companion software algorithm. This, in turn, organizes the data as a series of intuitive transactions representing the interface's activity leading up to the failure.

BiPeD eases the debugging process by locating the bug manifestation time and location, tolerating noisy, non-deterministic post-silicon environments without requiring failure reproduction. It provides a rich set of debugging information, including critical signals involved, modules, event, transaction, and a history of recent activity. Additionally, BiPeD incurs zero performance overhead during regular post-silicon validation, and requiring off-chip data transfer only at the occurrence of a bug. Finally, it bridges pre-silicon, post-silicon and runtime verification through the abstraction of protocols and flexible, reusable hardware.

## Pre-silicon Verification: Learning Protocols

During pre-silicon verification, BiPeD learns the semantics of a design's protocols with Inferno's protocol extraction software (Section 3.2). Later, these semantics are checked at-speed by flexible protocol detection hardware during post-silicon validation. When a check fails, the history of the activity observed on the failed interface is transferred off-chip for analysis by an off-line software algorithm. The result is a rich set of debugging information, which includes a trace of intuitive, high-level descriptions of the behavior leading up to the failure. Section 3.3.1 describes this process.

Leveraging the full observability of the design during pre-silicon verification, the protocols for a design's interfaces are generated during pre-silicon verification. First, interfaces are identified by designers: each interface is defined by a set of the design's signals, usually control signals. Passing testcases are then run on the system, generating traces for protocol extraction. The end result is a protocol representing the expected behavior of the interface, saved to a "protocol database" for use during post-silicon validation. This process is repeated for each interface selected.

#### Post-silicon Validation: Failure Detection

BiPeD leverages the abstraction of protocols for post-silicon bug diagnosis. The fast execution speeds of post-silicon validation enable high coverage, thus the protocols learned during pre-silicon verification can now be stressed with heavy testing. Our in-hardware solution monitors a number of interfaces simultaneously, confirming that the observed events and transitions conform to the protocol. When a mismatch is detected, our solution considers the past history of events and uses it to diagnose the bug, identifying the time of the failure, the errant transaction, the modules and signals involved, etc.

In order to monitor the interfaces of interest and check them against their corresponding communication protocol, the design is augmented with flexible hardware protocol detectors. During post-silicon validation, a number of protocols are programmed into "detector" hardware blocks, one block for each monitored interface. At runtime, the detector hardware units monitor the interfaces' activity to check that it conforms to the known protocol. The details of post-silicon bug detection are in Section 3.3.2.

#### Runtime Reliability: Fault Detection

BiPeD's hardware protocol detector has the advantage of flexibility. After post-silicon validation is complete, its programmable detectors can be repurposed for runtime failure detection. When coupled with the runtime solutions outlined in this dissertation, BiPeD can be leveraged as an effective detection mechanism during runtime verification. The programmable, multi-use hardware provided by BiPeD's protocol detectors make it an effective use of silicon area. Sections 5.2 and 3.5.4 discuss runtime applications of BiPeD.

## 6.2 Case Study

We illustrate BiPeD's capabilities on the OpenSPARC T2 microprocessor, demonstrating that it is able to accurately detect bugs and provide intuitive debugging information. We identified 10 interfaces within the design, creating a protocol for each. In this case study, we focus on the TLU interface 3.8, which monitors signals that connect the trap logic unit (TLU) and the load store unit (LSU). The interface contains 5 signals: `protect`, indicating protected memory; `thread sync` occurs with high latency LSU operations, e.g., a D-cache miss. `TLB bypass` indicates that the instruction in the bypass stage bypassed the TLB. `ASI reload` indicates the ASI (address space identifier) reload is enabled, and `flush` monitors whether the instruction in the bypass stage is being flushed.

We first built the protocol database containing all 10 interfaces by running 10 different test cases, each with 100 different random seeds. More details on the protocols can be found in Section 6.3, Table 6.2. We then ran a buggy version of the design, with hardware protocol detectors programmed to monitor each interface. The buggy version contained an error in the LSU's `protect` signal, injected after 10,000 execution cycles.

The protocol detector monitoring the TLU interface quickly identified a mismatch at cycle 10,016, detecting an erroneous transition. A history of events from the circular buffer in the protocol detector then underwent transaction extraction, and 70 transactions were identified. Figure 6.1 shows a subset of these transactions: four correct, and one buggy transaction. The transactions shown in the Figure begin with a thread sync, where thread synchronization occurs after the TLB is bypassed. Next, a burst TLB bypass with sync, was observed, a sequence of two TLB bypasses: the first bypass triggers synchronization and the second one without synchronization. The last two correct transactions were TLB bypasses, single bypasses with no synchronization.

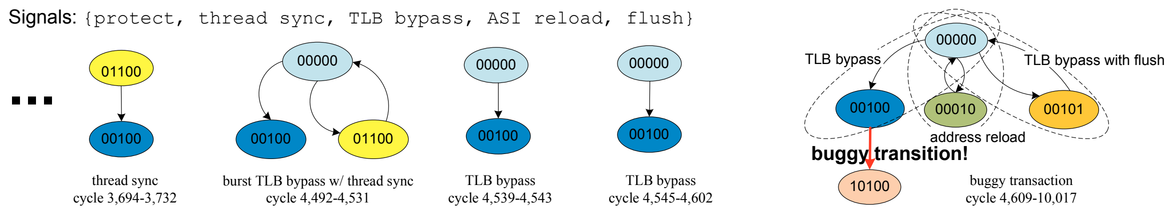


**Figure 6.1 Transaction history example.** Transactions extraction was performed for a bug in the OpenSPARC T2 TLU interface: the buggy transaction is shown at the right, preceded by the four transactions that led to it. Bit vectors in each state represent signal values. As shown in the figure, the buggy transaction contains a transition edge not included in the approved protocol diagram. The dotted ovals indicate behaviors that appear within the buggy transaction.

The rightmost transaction in Figure 6.1 contains the errant transition, detected by a protocol detector mismatch. This transaction has some similarity to previously observed

| **5-stage pipeline** | | |
|---|---|---|
| **test case** | **description** | **length (cycles)** |
| bubblesort | bubble sort | 569,237 |
| combRec | recursive combinations | 4,609,206 |
| fib | Fibonacci numbers | 442,233 |
| hanoi | tower of Hanoi puzzle | 11,593,567 |
| insert | insertion sort | 229,429 |
| knapsack | knapsack problem | 1,597,497 |
| matmult | matrix multiplication | 1,723,423 |
| merge | merge sort | 548,172 |
| quick | quick sort | 277,503 |
| saxpy | scalar alpha x plus y | 60,024 |
| **OpenSPARC T2** | | |
| **test case** | **description** | **length (cycles)** |
| blimp_rand | hypervisor test | 251,480 |
| fp_addsub | floating point add/subt | 913,093 |
| fp_muldiv | floating point mult/div | 238,343 |
| isa2_basic | constrained-random | 452,009 |
| isa3_asr_pr | constrained-random | 1,178,151 |
| isa3_window | constrained-random | 1,282,348 |
| mpgen_smc | constrained-random | 135,251 |
| ldst_sync | thread sync. instrs. | 64,570 |
| n2_lsu_asi | load/store unit test | 62,523 |
| tlu_rand | trap logic unit test | 591,434 |

**Table 6.1 Workloads used for evaluation.** The OpenSPARC testcases are subset of those that ship with OpenSPARC T2.

correct transactions: first, it contains a single TLB bypass (shown with dashed circle). Additional components in this complex transactions are a burst address reload, a burst TLB bypass with flush, as well as a TLB bypass. The transition from the single TLB bypass to data access protection is not included in the approved protocol diagram, thus an error is flagged. After detecting the bug, BiPeD provides accurate and intuitive debugging information: the relevant interface (TLU), modules (load-store unit and trap logic unit), the exact buggy transaction, the buggy signal (`protect`) and the detection cycle (10,016).

## 6.3 Evaluation

We employed BiPeD to locate failures in two hardware designs: a simple 5-stage pipeline, and the OpenSPARC T2 design. We simulated the OpenSPARC T2 design in its `cmp1` configuration, which included a SPARC core, cache, memory and crossbar. The 5-stage

| **5-stage Pipeline** | | | | | |
|---|---|---|---|---|---|
| **interface** | **description** | **signals** | **bits** | **transitions** | **events** |
| branch | branch predictor | 3 | 3 | 10 | 6 |
| decode | decode stage | 8 | 8 | 33 | 9 |
| fetch | fetch stage | 5 | 5 | 33 | 13 |
| illegal | illegal instr. logic | 6 | 6 | 7 | 6 |
| mem stage | memory stage | 3 | 4 | 6 | 4 |
| mem ctrl | memory controller | 2 | 6 | 10 | 5 |
| execute | execute stage | 4 | 8 | 51 | 20 |
| forward | forwarding logic | 6 | 6 | 78 | 32 |
| pipeline | pipeline registers | 6 | 6 | 31 | 13 |
| writeback | writeback stage | 3 | 3 | 14 | 6 |
| **OpenSPARC T2** | | | | | |
| **interface** | **description** | **signals** | **bits** | **transitions** | **events** |
| CPX | cache to processor | 5 | 33 | 188 | 18 |
| branch | EX branch logic | 5 | 5 | 222 | 16 |
| CCX | cache Xbar | 6 | 20 | 215 | 23 |
| memory | memory control unit | 12 | 12 | 135 | 21 |
| execute | execute unit | 5 | 7 | 107 | 16 |
| FPU | floating point unit | 10 | 10 | 622 | 62 |
| fetch | fetch unit | 6 | 6 | 101 | 16 |
| perf | performance monitor | 3 | 5 | 23 | 6 |
| TLU | trap logic unit | 5 | 5 | 161 | 16 |
| PCX | processor to cache | 4 | 4 | 12 | 6 |

**Table 6.2 Monitored interfaces.** We instrumented each design to monitor 10 interfaces during program execution. The 5-stage pipeline's interfaces are smaller than OpenSPARC's, resulting in fewer events and transitions.

in-order pipeline implemented a subset of the Alpha ISA, and comprised approximately 5,000 lines of Verilog HDL code. Each design was simulated in behavioral Verilog, and equipped to monitor the protocols of 10 simultaneous interfaces. Table 6.2 describes the interfaces, showing the number of signals in each, as well as the number of bits, as some of the signals may be more than 1 bit wide.

First, we ran each design free of bugs, training the protocol detector on 10 testcases (Table 6.1), ranging from 60,000 cycles to almost 12 million cycles in length. 100 variations of each testcase were run, using different random seeds to introduce execution variations with variable and random communication latencies. The number of events and transitions observed is shown in Table 6.2. The table shows that the interfaces in the 5-stage design are smaller, comprising fewer signals. Thus, each protocol has a smaller number of events and transitions.

### 6.3.1 Protocol Detection

After building the protocol database with bug-free testcases, we employed the protocol detection hardware to detect a set of 10 bugs injected into each design, described in Table 6.3. Each buggy execution contained one bug, which was injected after 5,000 cycles in the 5-stage design, and after 10,000 cycles in OpenSPARC. First, the detectors were programmed with the protocols described in Table 6.2. Each bug/testcase combination was then run with protocol detection hardware monitoring the 10 protocols, and the latency from bug injection to bug detection was recorded for each protocol. Each buggy execution was simulated with different random variations (random seeds) compared to protocol extraction: thus, no buggy execution matched any training execution.

Table 6.4 reports the latency (cycles) from bug injection to bug detection for each bug/testcase combination. We note that BiPeD reports the exact cycle of a protocol mismatch, while the table measures the time from bug injection to detection. For each bug

| **OpenSPARC T2** | |
|---|---|
| **bug** | **description** |
| branch | failure in branch to fetch communication |
| decode halt | decode error of halt instr. |
| EX valid instr. | erroneous invalid instruction |
| stall fetch | fetch stalled and instr. lost |
| register index | incorrect destination register index |
| source operand | incorrect source operand index |
| mem response | memory-to-processor communication failure |
| EX operand | select wrong operand for execution |
| pipeline valid | erroneous valid instruction |
| WB enable | writeback-register file communication error |
| **5-stage Pipeline** | |
| **bug** | **description** |
| branch | failure in branch to fetch communication |
| EX valid instr. | execution unit error |
| cache-proc req | erroneous cache-to-processor request |
| MEM read ack | erroneous memory load acknowledgment |
| FPU exception | floating point exception error |
| fetch thread id | LD/ST to fetch communication error |
| LSU data access | incorrect LSU access |
| table walk req | page table walk request |
| PCX stall | processor-to-cache communication stall |
| CCX/PCX req | processor/cache communication error |

**Table 6.3 Bugs injected,** one at a time, after 5,000 cycles in the 5-stage design and after 10,000 cycles in OpenSPARC.

| | 5-stage Bug | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Interface** | branch | decode halt | EX valid instr. | stall fetch | register index | source operand | mem response | EX operand | pipeline valid | WB enable |
| branch | 4 | | | | | | | | | |
| decode | 88 | 4 | | 17 | | | 3M | | | |
| fetch | 4 | | 2k | 5 | | | | | | |
| illegal | 227 | 88 | | 19 | | | 3M | | | |
| mem stage | | | 2k | | | | | | | |
| mem ctrl | | | | 7 | | | 4 | | | |
| execute | 5 | | 2k | | | 4 | | 17 | | |
| forward | 5 | 158 | 4 | 8 | 124 | | 3M | | 4 | 4 |
| pipeline | 4 | | 356 | | | | | | | |
| writeback | 17 | | 156 | 91 | 124 | | | | 291 | 4 |

| | OpenSPARC Bug | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Interface** | branch | EX valid instr. | cache-proc req | MEM read ack | FPU exception | fetch thread id | LSU data access | table walk req | PCX stall | CCX/PCX req |
| CPX | 1,719 | | 16 | | | | | | | |
| branch | 242 | | | | | | | | | |
| CCX | 16k | 39 | 16 | | | | | | | 742 |
| memory | | | | 223 | | | | | | |
| execute | | 16 | | | | | | f.n. | | |
| FPU | | f.p. | 22k | 48k | 739 | 48k | | | 22k | |
| fetch | | | | | | 47 | | | | |
| perf. | | | | | | | | | | |
| TLU | | | | | | | 16 | | | |
| PCX | | | | | | | | | 767 | 764 |

**Table 6.4 Bug detection latency** (cycles) from bug injection to detection. Each bug was first detected after 22 cycles, on average, in the 5-stage design, and after 281 cycles in OpenSPARC. Additionally, most bugs were detected by one interface earlier than the others, demonstrating precise bug localization.

in OpenSPARC, the first interface to detect the error flagged it error within 281 cycles, on average, while the bugs were first detected after 22 cycles on average in the 5-stage design. The small size of the 5-stage design caused the effects of bugs to spread rapidly through the design, enabling them to be caught more quickly. The first interface to identify the bug is marked in green (light shading). While most OpenSPARC bugs were detected by one interface many cycles before all others, the `cache-proc req` bug was detected by two interfaces simultaneously. In this case, two closely related interfaces caught the bug: the cache-to-processor crossbar (`CPX`) and the cache crossbar (`CCX`). We observed one false positive (red/dark shading, marked "f.p.") with the `EX valid instr.` bug, due to noise introduced by the new random variations, which had not been observed during training. One bug evaded detection (`table walk req`, which resulted in false negatives ("f.n.") marked with orange/medium shading. In this case, the bug signal was not part of any interface and the bug did not cause wider system effects detectable by other interfaces.

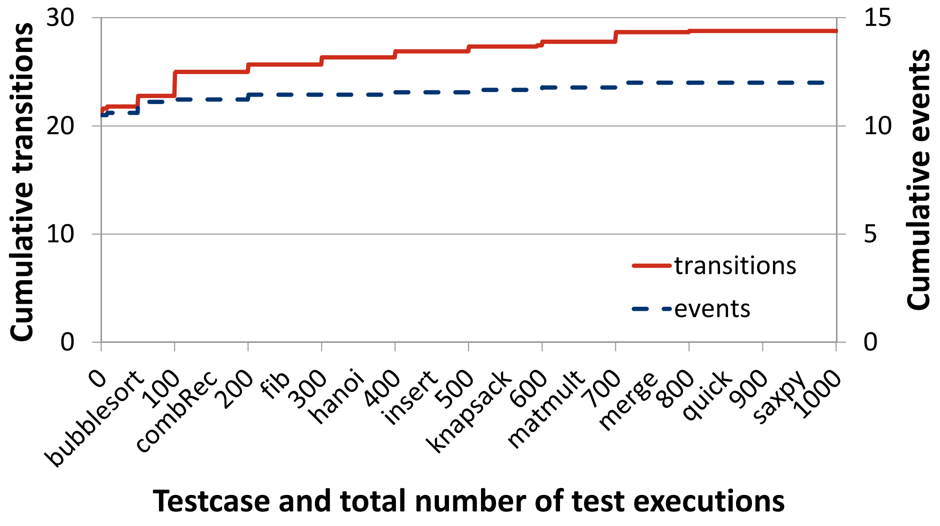
### 6.3.2 Protocol Extraction

We also evaluated the effectiveness of pre-silicon protocol detection, using the 10 testcases each with 100 random variations, for a total of 1,000 training tests. With each test execution, new events and transitions were added to the protocol database. Figure 6.2 shows the number of events and transitions in each interface, on average. We observed that, as the volume of training data increased, the number events and transitions increased too, quickly with the first few tests, and then leveling off.

In addition to the discovery of events and transitions, protocol extraction impacts the number of false positives encountered by the detection phase. We applied leave-one-out-cross-validation to determine the effect of differing sets of training data. Here, 10 different protocol databases were used: each trained on 9 of the 10 available testcases. With the OpenSPARC T2 design, we found that leaving out a testcase had the effect of increasing the number of false positives, as shown in Figure 6.3. We found that excluding the `blimp_rand` testcase resulted in a 24% false positive rate among all bug/testcase combinations, underscoring its importance as a training test. Additionally, the coverage of the testcases for the small 5-stage design is very good, and as a result, we observed no false positives in this design (Table 6.4), even with leave-one-out-cross-validation.

### 6.3.3 Transaction Extraction

We then explored the effect of circular buffer size on the number of transactions extracted. Figure 6.4 reports the total number of transactions, as well as unique transactions, for different buffer sizes. We observed that the total transactions scaled with the size of the buffer (on average), while unique transactions leveled off. As more unique transactions are discovered, the observance of repeated transactions increases, indicating that high-quality transaction are being extracted.



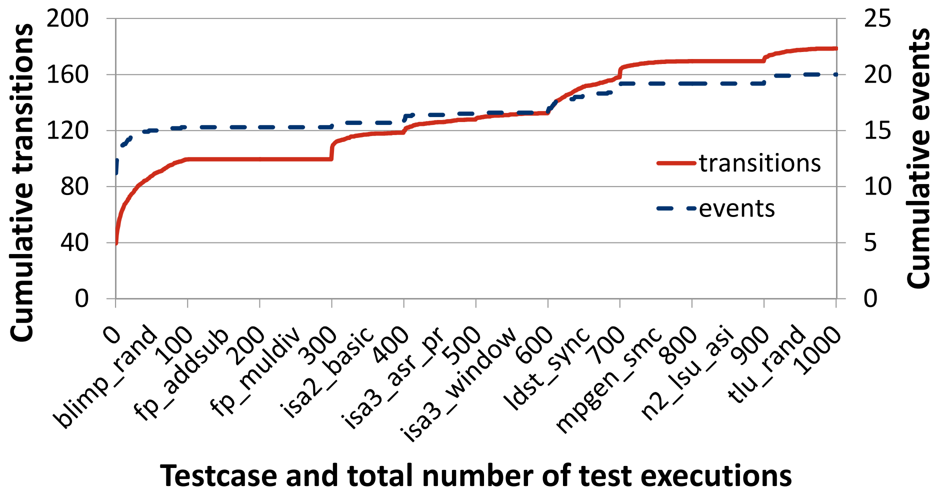


**Figure 6.2 Protocol extraction.** The plot reports the number of events and transitions in a protocol, on average. 10 testcases were used with 100 random seeds each for a total of 1000 training tests per design, reflected on the X-axis. As the number of training tests accumulates, the size of the protocol approaches a consistent value. Steps in the graph represent the transition from one testcase to another.

### 6.3.4 Area Overhead

We evaluated the area overhead of an implementation of the protocol detection hardware in Verilog HDL, synthesized with a 65nm TSMC target library. The storage dominates the area:

$$\begin{array}{ll} \multicolumn{2}{l}{total\ storage =} \\ ifc\_bits \cdot num\_events + & \text{event storage} \\ 2\log_2(ifc\_bits) \cdot num\_transitions + & \text{transition storage} \\ \log_2(ifc\_bits) \cdot buffer\_size & \text{circular buffer} \end{array}$$

Despite the complexity of the OpenSPARC T2 design, we found that protocols were

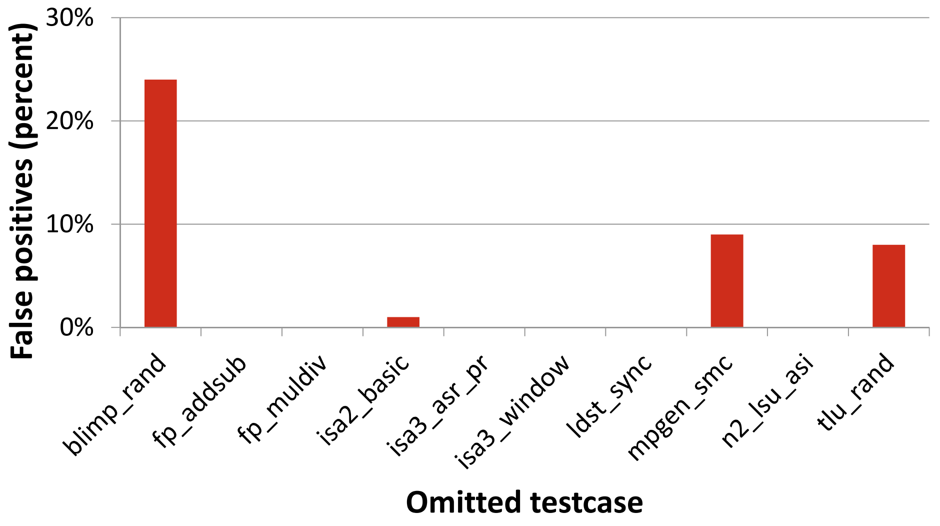


**Figure 6.3 Effect of leave-one-out-cross-validation.** The percentage of false positives with 9 training testcases and 1 testing in the OpenSPARC T2 design. No false positives were observed in the 5-stage design during cross-validation.

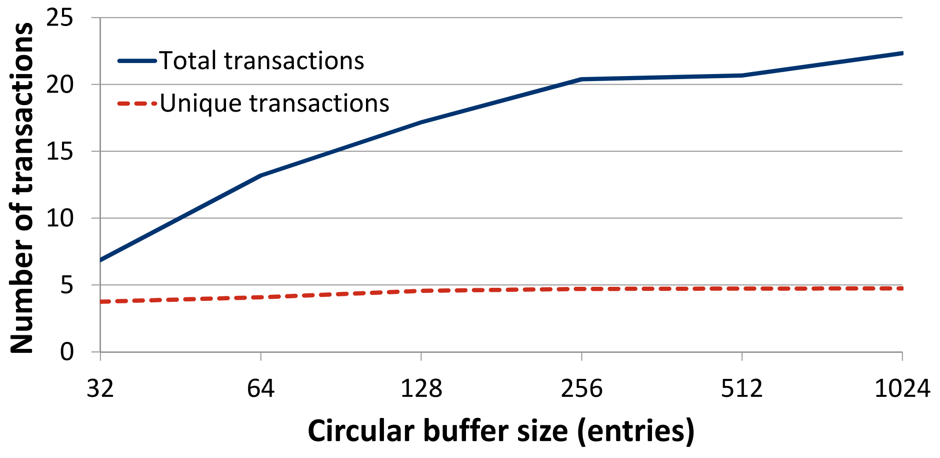
limited in size. A detector sized to handle the largest OpenSPARC T2 interface can handle 62 events, each 33 bits wide, and 622 transitions. With this configuration, the resulting protocol detector required 15.3KB of storage and comprised 0.251 mm$^2$ in our 65nm library. When compared to the total area of the OpenSPARC T2 chip (342 mm$^2$[45]), the area overhead of 10 detectors (one for each monitored interfaces), is 0.7%. While this configuration is adequate for the OpenSPARC design, the area of other configurations are shown in Table 6.5.

| buffer entries | interface bits | num. events | num. transitions | storage | area |
|---|---|---|---|---|---|
| 1,024 | 8 | 16 | 32 | 3.3 KB | 67,839 $\mu m^2$ |
| 1,024 | 16 | 32 | 64 | 5.0 KB | 95,129 $\mu m^2$ |
| 1,024 | 32 | 64 | 128 | 8.3 KB | 149,673 $\mu m^2$ |
| 1,024 | 64 | 128 | 256 | 17.0 KB | 296,236 $\mu m^2$ |
| 1,024 | 128 | 256 | 512 | 46.0 KB | 781,095 $\mu m^2$ |

**Table 6.5 Area comprised by one protocol detector**, for several configurations. Each configuration has a maximum number of circular buffer entries, number of bits supported by the monitor interface, number of stored events, and number of stored transitions.

## 6.4 Summary

BiPeD is a verification framework that bridges pre-silicon verification with post-silicon validation, leveraging flexible hardware that is later applied to runtime verification. It provides



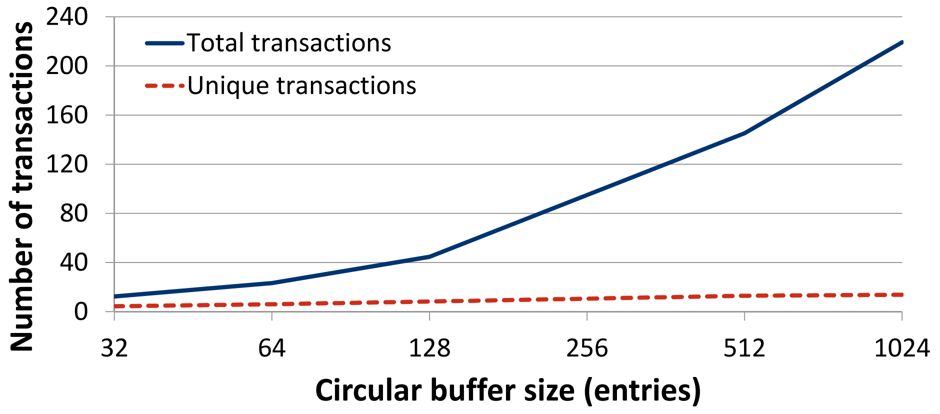


**Figure 6.4 Transactions extracted from the circular buffer** as the number of buffer entries changes. The plot shows both total, as well as the number of unique transactions. While the total increases with buffer size, unique transactions approach a constant.

a framework for the integration of the complementary solutions outlined in this dissertation.

By making use of the high-level, compact, intuitive transactions and protocols of Inferno, BiPeD is able to learn the behavior of a design's interfaces during pre-silicon verification and enforce it during post-silicon and runtime validation. Our framework is capable of detecting bugs in the industrial-size OpenSPARC T2 design and able to accelerate post-silicon validation with intuitive debugging information. It enables a number of complementary solutions during pre-silicon, post-silicon and runtime verification, exploiting the synergies among the different verification phases.

# Chapter 7
# Conclusions

With this dissertation, we have presented a comprehensive solution to ensure the correct operation of the communication system in a multi-core chip. Multi-core communication is critical to the operation of a chip in which many processors work concurrently. However, this system-wide component is prone to failure due to its size and complexity. Furthermore, its constituent transistors are increasingly likely to fail as dimensions shrink.

We provide a taxonomy for categorizing the problem into three sub-problems: functional bugs, electrical failures, and transistor faults. In addition to dividing the problem space, we also divide the solution space into the three categories: pre-silicon verification, post-silicon validation and runtime verification. We explore BiPeD, a framework that addresses the correctness problem, synergizing the phases of verification to provide a comprehensive solution that draws on the strengths of each phase. Figure 7.1 reviews our solution framework.

## 7.1 Bridging Verification Phases

**BiPeD** bridges the different verification phases, identifying synergies among them to improve the verification process. **Inferno** is an integral part of the framework, defining the level of abstraction by generating high-level, compact, intuitive transactions and protocols during pre-silicon verification. These transactions ease the understanding of complex communication protocols during pre-silicon verification. They also leverage the high observability of pre-silicon verification to learn the correct behavior of a design's interfaces.

After pre-silicon verification, BiPeD's flexible hardware protocol detectors quickly detect bugs during high-speed post-silicon validation. The detectors enforce the correct behavior learned during the previous phase. When a mismatch occurs **Dacota** and **BPS** are deployed to narrow down the bug to its root cause.

Finally, the programmable hardware provided by the protocol detectors is re-purposed for runtime verification. By tracking internal router control signals, BiPeD's detectors

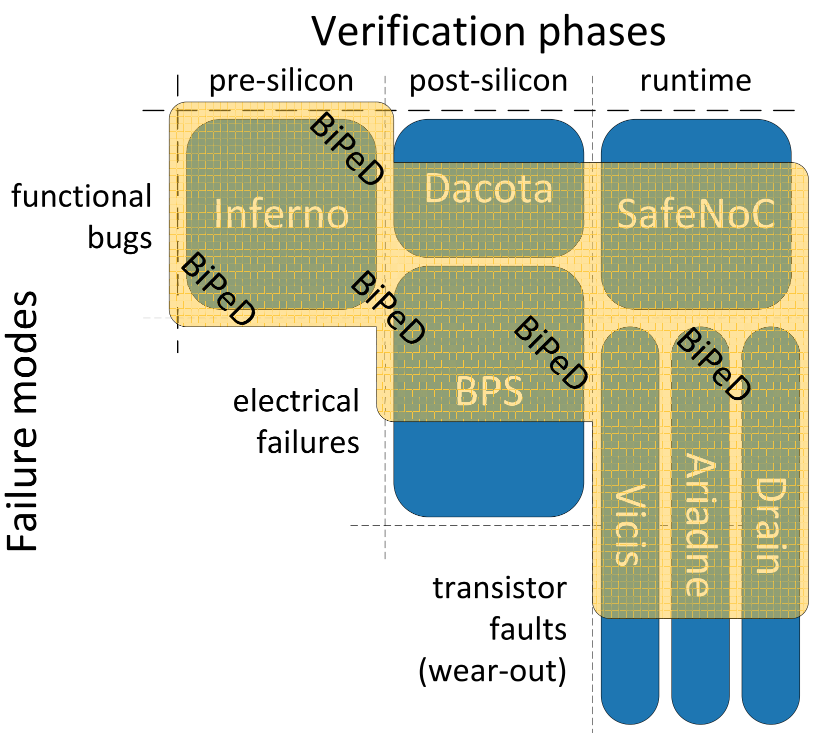


**Figure 7.1 Review of the solution framework proposed by this dissertation**, which synergizes the different phases of verification in the BiPeD framework (shaded region). It works with integral component solutions (completely shaded) and complementary solutions (partially shaded) to address errors and ensure correctness.

identify errant control paths. When an error is detected, it triggers the process of diagnosis (**Vicis**), reconfiguration (**Ariadne**), and recovery (**Drain**). The detectors are also enhance the coverage of end-to-end correctness guaranteed by **SafeNoC**.

## 7.2 Functional Bugs

Functional bugs may be present in the design starting from its inception during pre-silicon verification, and persist through post-silicon verification and into runtime. Our framework leverages the high-observability of pre-silicon verification to learn correct design behavior with Inferno. This correct behavior is then enforced during post-silicon validation and runtime verification with BiPeD's flexible hardware detectors. When an error is detected, Dacota helps narrow down the source of a common class functional bugs. Finally, SafeNoC guarantees end-to-end correctness at runtime.

**Inferno** is software tool that is used to understand and validate complex communication protocols during pre-silicon verification, learning the correct behavior of the design. It automatically extracts transactions from a simulation trace, that is, high level descriptions of a design's behavior. Transactions are presented to the user through simple and intuitive diagrams for which we have developed a number of specialized visualization enhancements. Complex, repetitive design simulations are distilled to a compact set of transactions which describe the semantic behavior of the system in a compact, high level format. We proposed a new verification methodology enabled by Inferno, called Transactional Verification, which greatly reduces the verification effort required to determine the correct behavior of a system. Our methodology is based on a closed-loop approach where transactions are extracted from a simulation and displayed to the user for approval. Inferno is at once practical and effective at extracting high-level design behavior from low-level, lengthy simulation data.

**BiPeD** provides hardware protocol detectors that monitor the communication protocols during post-silicon test executions. They enforce the correct behavior learned during high-observability pre-silicon verification during high-speed post-silicon validation. When an error is detected, a complementary solution is deployed to narrow down the source of the failure.

**Dacota** verifies a common class of functional bugs that occur in the memory subsystem. It is deployed after BiPeD's protocol detectors flag an error, when Dacota commences high-coverage post-silicon validation of memory ordering. When enabled by the verification team, Dacota stores sequence information about issued memory operations, periodically aggregating this information to perform a software-based policy validation. The validation algorithm is implemented purely in software to minimize the area impact of our solution and executes on existing processor resources. Leveraging approximately 6 orders of magnitude performance advantage over pre-silicon simulation, Dacota's post-silicon approach is able to offer significantly higher coverage compared to pre-silicon approaches. We found that Dacota is effective in detecting subtle consistency and coherence bugs, showing its promise as a solution to the problem of validating the order of memory operations in CMP systems. Furthermore, Dacota enables post-silicon debugging support, providing invaluable information to the validation team.

**SafeNoC** is a runtime end-to-end error detection and recovery technique that guarantees the functional correctness of CMP interconnects. Its coverage is improved by enhancements provided by BiPeD hardware. SafeNoC augments the interconnect with a lightweight and simple checker network and it detects functional errors by comparing the signature of every received data packet with its look-ahead signature that was delivered through the checker network. In case of mismatches, we use a novel recovery approach during which blocked packets and stray flits are collected from the primary network and are distributed over the checker network to all processor cores, where our reconstruction algorithm reassembles them. SafeNoC can detect and recover from a broad range of functional design errors, while incurring a low performance impact.

## 7.3 Electrical Failures

Electrical failures may occur beginning with the first chip prototypes, and are a common source of errors during post-silicon validation. Here, circuit timing failures may be intermittent, occurring only in certain on-chip conditions. Similar to functional bugs, the **BiPeD** protocol detectors are used to monitor execution on post-silicon prototypes. When an error is detected, BPS is applied to narrow down the root cause of post-silicon electrical failures.

**BPS** is a solution for locating the most challenging bugs during post-silicon validation: those bugs with inconsistent program outcomes. While it is able to localize electrical failures, it is also effective for functional and manufacturing bugs as well. BPS has two components: hardware structures that log a compact encoding of observed signal activity and companion post-analysis software. BPS can localize bugs in time and space while tolerating non-repeatable executions of the same test. It provides a fast solution, reducing off-chip data transfers with compact signatures and scales to industrial-size designs. BPS is effective in locating bugs under many different workloads, often to the exact signal.

During post-silicon validation, many electrical failures are fixed, and thus their effects are diminished in the final product. However, those that may slip through share many properties with transistor faults. While transistor faults are modeled as stuck-at errors on a signal in the design, electrical failures may be modeled in similar fashion, but with a shorter duration. Thus, the techniques used to addressed transistor faults in the field can be applied to electrical failures as well.

## 7.4 Transistor Faults

Once a chip has been shipped, transistor faults can be caused by a variety of silicon wear-out mechanisms. In order to protect against the correctness failures induced by malfunctioning transistors, a comprehensive runtime solution is necessary. Our solution works by first detecting the occurrence of faults with BiPeD, then diagnosing the affected hardware blocks with Vicis. Next, it reconfigures the architecture around the faults using a flexible routing algorithm called Ariadne. Finally, the system recovers any data that may be lost in the reconfiguration process with Drain.

**BiPeD** protocol detectors are used to detect that a fault has occurred at runtime. By monitoring internal router control signals, it ensures that correct control paths are followed by the routing logic. When a fault is detected, it triggers the diagnosis mechanism.

**Vicis** diagnoses hardware faults in both network-on-chip router components as well as the reliability components themselves. It maintains correct operation in the face of faults, trading off performance for correctness. As the number of failures increases, Vicis mitigates errors by reconfiguring the router architecture. By leveraging the redundancy inherent in networks-on-chip, and NoC routers, Vicis can maintain high reliability.

**Ariadne** reconfigures an NoC around failed routers and cores, providing an algorithm capable of circumventing large numbers of simultaneous faults, and able to handle unreliable future silicon technologies. Ariadne utilizes up*/down* for high performance and deadlock-free routing in irregular network topologies that result from large numbers of faults, and offers performance gains ranging from 40% to 140% (for 50 faults) during normal operation, compared to state-of-the-art fault tolerant solutions. It guarantees that if a path between two nodes exists, the reconfiguration algorithm will enable at least one deadlock-free path between them. Ariadne is implemented in a fully distributed mode, since nodes coordinate to explore the surviving topology, resulting in very simple hardware and low complexity. At 1.97% area overhead, Ariadne is a parsimonious solution for many-core processor designs of the future, enabling a trade-off between performance and reliable functionality on unreliable silicon.

**Drain** recovers data following reconfiguration. Drain augments a CMP interconnect architecture with emergency links that facilitate the recovery of dirty cache data and architectural state in the event that a node or subnetwork becomes disconnected. In the experimental results, we show that Drain is able to recover data from disconnected nodes in any faulty network configuration, even those where aggressive failures cause network partitioning. It is able to provide complete state recovery for an entire 64-node CMP within several milliseconds and it incurs very low area overhead of 4,952 gates at each node.

## 7.5 Summary

Throughout the verification process, the BiPeD framework unites solutions from pre-silicon, post-silicon and runtime verification. This new synergy among the verification phases enables more efficient and effective design verification. The end result is a more robust communication infrastructure that is resilient to errors and ensures correct operation throughout the lifetime of the chip.

# Bibliography

[1] Rawan Abdel-Khalek, Ritesh Parikh, Andrew DeOrio, and Valeria Bertacco. Functional correctness for CMP interconnects. In *Proc. ICCD*, 2011.

[2] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A reconfigurable design-for-debug infrastructure for SoCs. In *Proc. DAC*, 2006.

[3] Dennis Abts, Steve Scott, and David J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *Proc. IPDPS*, 2003.

[4] Mithun Acharya. Automatic generation and inference of interface properties from program source code. In *Proc. OOPSLA*, 2006.

[5] Allon Adir, Maxim Golubev, Shimon Landa, Amir Nahir, Gil Shurek, Vitali Sokhin, and Avi Ziv. Threadmill: A post-silicon exerciser for multi-threaded processors. In *Proc. DAC*, 2011.

[6] Advanced Micro Devices, Inc. *Revision Guide for AMD Athlon(TM) 64 and AMD Opteron(TM) Processors*, August 2005.

[7] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj K. Jha. Garnet: A detailed on-chip network model inside a full-system simulator. *Proc. ISPASS*, 2009.

[8] Konstantinos Aisopos, Andrew DeOrio, Li-Shiuan Peh, and Valeria Bertacco. ARIADNE: Agnostic reconfiguration in a disconnected network environment. In *Proc. PACT*, 2011.

[9] Muhammad A. Alam. A critical examination of the mechanics of dynamic NBTI for PMOSFETs. In *Proc. IDEM*, 2003.

[10] Glenn Ammons, Rastislav Bodik, and James Larus. Mining specifications. In *Proc. POPL*, 2002.

[11] K. V. Anjan and Timothy Mark Pinkston. An efficient, fully adaptive deadlock recovery scheme: DISHA. In *Proc. ISCA*, 1995.

[12] Sean Baartmans and Bryan White. *U.S. Patent no. 6438664: Customizable event creation logic for hardware monitoring*. Intel Corp., 2007.

[13] Keith Baker and Jos Van Beers. Shmoo plotting: The black art of ic testing. *IEEE Des. Test*, 14(3), 1997.

[14] Alphan Bayazit and Sharad Malik. Complementary use of runtime validation and model checking. In *Proc. ICCAD*, 2005.

[15] Shane Bell, Bruce Edwards, John Amann, Rich Conlin, Kevin Joyce, Vince Leung, John MacKay, Mike Reif, Liewei Bao, John Brown, Matthew Mattina, Chyi-Chang Miao, Carl Ramey, David Wentzlaff, Walker Anderson, Ethan Berger, Nat Fairbanks, Durlov Khan, Froilan Montenegro, Jay Stickney, and John Zook. TILE64 processor: A 64-core SoC with mesh interconnect. In *Proc. ISSCC*, 2008.

[16] Saddek Bensalem, Yassine Lakhnech, and Hassen Sadi. Powerful techniques for the automatic generation of invariants. In *Proc. CAV*, 1996.

[17] Davide Bertozzi, Luca Benini, and Giovanni De Micheli. Low power error resilient encoding for on-chip data buses. In *Proc. DATE*, 2002.

[18] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proc. PACT*, 2008.

[19] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38(1), 2006.

[20] Paul Bogdan, Tudor Dumitras, and Radu Marculescu. Stochastic communication: A new paradigm for fault-tolerant networks-on-chip. In *Hindawi Publishing Corporation Open Access Journal*, 2007.

[21] Rajendra V. Boppana and Suresh Chalasani. Fault-tolerant wormhole routing algorithms for mesh networks. *IEEE Trans. Computers*, 44(7), 1995.

[22] Shekhar Borkar. Microarchitecture and design challenges for gigascale integration. In *Proc. MICRO*, 2004.

[23] Shekhar Borkar, Norman P. Jouppi, and Per Stenstrom. Microprocessors in the era of terascale integration. In *Proc. DATE*, 2007.

[24] Dhanajay Brahme, Steven Cox, Jim Gallo, William Grundmann, C. Ip, William Paulsen, John Pierce, John Rose, Dean Shea, and Karl Whiting. The transaction-based verification methodology. Technical report, Cadence Design Systems, Inc., 2000. Technical Report No. CDNL-TR-2000-0825.

[25] Doug Burger and Todd Austin. The SimpleScalar toolset, version 3.0. `http://www.simplescalar.com`.

[26] Michael Bushnell and Vishwani Agrawal. *Essentials in Electronic Testing*. Springer, 2000.

[27] Harold W. Cain, Mikko H. Lipasti, and Ravi Nair. Constraint graph analysis of multithreaded programs. In *Proc. PACT*, 2003.

[28] S. Chalasani and R.V. Boppana. Communication in multicomputers with nonconvex faults. *IEEE Trans. Computers*, 46(5), 1997.

[29] Kaiyu Chen, Sharad Malik, and Priyadarsan Patra. Runtime validation of memory ordering using constraint graph checking. In *Proc. HPCA*, 2008.

[30] Ludmila Cherkasova, Vadim Kotov, and Tomas Rokicki. Fibre channel fabrics: Evaluation and design. In *International Conference on System Sciences*, 1995.

[31] Andrew A. Chien and Jae H. Kim. Planar-adaptive routing: Low-cost adaptive networks for multiprocessors. In *Proc. ISCA*, 1992.

[32] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *Proc. CAV*, 2002.

[33] Kypros Constantinides, Stephen Plaza, Jason Blome, Bin Zhang, Valeria Bertacco, Scott Mahlke, Todd Austin, and Michael Orshansky. Bulletproof: a defect-tolerant cmp switch architecture. In *Proc. HPCA*, 2006.

[34] David Culler, Anoop Gupta, and Jaswinder Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[35] William Dally, Larry Dennison, David Harris, Kinhong Kan, and Thucydides Xanthopoulos. The reliable router: A reliable and high-performance communication substrate for parallel computers. In *Proc. PCRCW*, 1994.

[36] William Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *Proc. DAC*, 2001.

[37] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, San Francisco, CA, USA, 2003.

[38] Flavio De Paula, Marcel Gort, Alan Hu, Steven Wilton, and Jin Yang. Backspace: formal analysis for post-silicon debug. In *Proc. FMCAD*, 2008.

[39] Andrew DeOrio, Kostantinos Aisopos, Valeria Bertacco, and Li-Shiuan Peh. DRAIN: Distributed recovery architecture for inaccessible nodes in multi-core chips. In *Proc. DAC*, 2011.

[40] Andrew DeOrio, Adam Bauserman, Valeria Bertacco, and Beth Isaksen. Inferno: streamlining verification with inferred semantics. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 28(5), 2009.

[41] Andrew DeOrio, David Fick, Valeria Bertacco, Dennis Sylvester, David Blaauw, Jin Hu, and Gregory Chen. A reliable routing architecture and algorithm for NoCs. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 31(5), 2012.

[42] Andrew DeOrio, Daya Shanker Khudia, and Valeria Bertacco. Post-silicon bug diagnosis with inconsistent executions. In *Proc. ICCAD*, 2011.

[43] Andrew DeOrio, Ilya Wagner, and Valeria Bertacco. DACOTA: Post-silicon validation of the memory subsystem in multi-core designs. In *Proc. HPCA*, 2009.

[44] David Dill, Andreas Drexler, Alan Hu, and C. Yang. Protocol verification as a hardware design aid. In *Proc. ICCD*, 1992.

[45] Xiangyu Dong and Yuan Xie. System-level cost analysis and design exploration for three-dimensional integrated circuits (3D ICs). In *Proc. ASPDAC*, 2009.

[46] José Duato. A theory of fault-tolerant routing in wormhole networks. *IEEE Trans. Parallel and Distributed Systems*, 8(8), 1997.

[47] Don Edenfeld, Andrew B. Kahng, Mike Rodgers, and Yervant Zorian. 2003 technology roadmap for semiconductors. *IEEE Computer*, 37(1), 2004.

[48] Michael Ernst. Verification for legacy programs. In *Proc. VSTTE (Verified Tools Theories, Tools, Experiments)*, 2005.

[49] Nicolas Falliere, Liam O Murchu, and Eric Chie. W32.stuxnet dossier. Technical report, Symantec, 2010.

[50] Görschwin Fey and Rolf Drechsler. Improving simulation-based verification by means of formal methods. In *Proc. ASPDAC*, 2004.

[51] David Fick, Andrew DeOrio, Valeria Bertacco, Dennis Sylvester, and David Blaauw. A highly resilient routing algorithm for fault-tolerant NoCs. In *Proc. DATE*, 2009.

[52] David Fick, Andrew DeOrio, Jin Hu, Valeria Bertacco, David Blaauw, and Dennis Sylvester. Vicis: a reliable network for unreliable silicon. In *Proc. DAC*, 2009.

[53] José Flich and José Duato. Logic-based distributed routing for NoCs. *Computer Architecture Letters*, 7(1), 2008.

[54] Robert A. Frohwerk. Signature analysis: A new digital field service method. *Hewlett-Packard Journal*, 1977.

[55] Steve Furber. Living with failure: Lessons from nature? In *Proc. ETS*, 2006.

[56] Jianliang Gao, Yinhe Han, and Xiaowei Li. A new post-silicon debug approach based on suspect window. In *Proc. VTS*, 2009.

[57] Steven German. Formal design of cache memory protocols in IBM. *Formal Methods in System Design*, 22(2), 2003.

[58] P. B. Ghate. Electromigration-induced failures in VLSI interconnects. In *Proc. Reliability Physics Symposium*, 1982.

[59] Christopher J. Glass and Lionel M. Ni. Fault-tolerant wormhole routing in meshes without virtual channels. *IEEE Trans. Parallel and Distributed Systems*, 7(6), 1996.

[60] M. E. Gómez, J. Duato, J. Flich, P. López, A. Robles, N. A. Nordbotten, O. Lysne, and T. Skeie. An efficient fault-tolerant routing methodology for meshes and tori. *IEEE Computer Architecture Letters*, 3(1), 2004.

[61] Shantanu Gupta, Shuguang Feng, Jason Blome, and Scott Mahlke. StageNet: A reconfigurable CMP fabric for resilient systems. In *Proc. Reconfigurable and Adaptive Architecture Workshop*, 2007.

[62] Sudheendra Hangal, Naveen Chandra, Sridhar Narayanan, and Sandeep Chakravorty. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *Proc. DAC*, 2005.

[63] Rongsen He and J.G Delgado-Frias. Fault tolerant interleaved switching fabrics for scalable high-performance routers. *IEEE Trans. Parallel and Distributed Systems*, 18(12), 2007.

[64] Ching-Tien Ho and Larry Stockmeyer. A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers. *IEEE Trans. Computers*, 53(4), 2004.

[65] Ted Hong, Yanjing Li, Sung-Boem Park, Diana Mui, David Lin, Helia Naeimi, Donald Gardner, Nagib Hakim, Ziya Khaleq, and Subhasish Mitra. QED: Quick error detection tests for effective post-silicon validation. In *Proc. ITC*, 2010.

[66] Mohammad Hosseinabady, Abbas Banaiyan, Mahdi Nazm Bojnordi, and Zainalabedin Navabi. A concurrent testing method for NoC switches. In *Proc. DATE*, 2006.

[67] Intel Corporation. *Intel(R) StrongARM(R) SA-1100 Microprocessor Specification Update*, February 2000.

[68] Intel Corporation. *Intel Core 2 Duo and Intel Core 2 Solo Processor for Intel Centrino Duo Processor Technology Specification Update*, September 2007.

[69] Intel Corporation. *Intel Core i7-900 Desktop Processor Series Specification Update*, July 2010.

[70] International Business Machines Corporation. *IBM PowerPC 750GX and 750GL RISC Microprocessor Errata Notice*, July 2005.

[71] Axel Jantsch, Robert Lauter, and Arseni Vitkowski. Power analysis of link level and end-to-end data protection in networks on chip. In *Proc. ISCAS*, 2005.

[72] Susmit Jha, Wenchao Li, and Sanjit A. Seshia. Localizing transient faults using dynamic bayesian networks. In *Proc. HLDVT*, 2009.

[73] Doug Josephson. The manic depression of microprocessor debug. In *Proc. ITC*, 2002.

[74] Doug Josephson. The good, the bad, and the ugly of silicon debug. In *Proc. DAC*, 2006.

[75] Frédéric Kastner. *Les Flammes Chantantes*. Dentu & Lacroix, Paris, 3rd edition, 1876.

[76] John Keane, Shrinivas Venkatraman, Paulo Butzen, and Chris H. Kim. An array-based test circuit for fully automated gate dielectric breakdown characterization. In *Proc. CICC*, 2008.

[77] Brian Keng, Sean Safarpour, and Andreas Veneris. Bounded model debugging. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 29(11), 2010.

[78] John Kim and Hanjoon Kim. Router microarchitecture and scalability of ring topology in on-chip networks. In *Proc. NoCArc*, 2009.

[79] Jongman Kim, Chrysostomos Nicopoulos, and Dongkook Park. A gracefully degrading and energy-efficient modular router architecture for on-chip networks. *ACM SIGARCH Computer Architecture News*, 34(2), 2006.

[80] Seong-Pyo Kim and Taisook Han. Fault-tolerant wormhole routing in mesh with overlapped solid fault regions. *Parallel Computing*, 23(13), 1997.

[81] Donald E. Knuth. *The Art of Computer Programming*. Four volumes. Addison-Wesley, 1968.

[82] Adán Kohler and Martin Radetzki. Fault-tolerant architecture and deflection routing for degradable NoC switches. In *Proc. NoCs*, 2009.

[83] Adán Kohler, Gert Schley, and Martin Radetzki. Fault tolerant network on chip switching with graceful performance degradation. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 29(6), 2010.

[84] Michihiro Koibuchi, Hiroki Matsutani, Hideharu Amano, and Timothy Mark Pinkston. L-turn routing: An adaptive routing in irregular networks. In *Proc. ICPP*, 2001.

[85] Michihiro Koibuchi, Hiroki Matsutani, Hideharu Amano, and Timothy Mark Pinkston. A lightweight fault-tolerant mechanism for network-on-chip. *Proc. NoCs*, 2008.

[86] Hyunjin Lee, Sangyeun Cho, and Bruce R. Childers. Performance of graceful degradation for cache faults. In *Proc. VLSI Symposium*, 2007.

[87] Ana Sonia Leon, Kenway W. Tam, Jinuk Luke Shin, David Weisner, and Francis Schumacher. A power-efficient high-throughput 32-thread SPARC processor. *IEEE Journal of Solid-State Circuits*, 42(1), 2007.

[88] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable specification mining for verification and diagnosis. In *Proc. DAC*, 2010.

[89] R. Libeskind-Hadas and E. Brandt. Origin-based fault-tolerant routing in the mesh. In *Proc. HPCA*, 1995.

[90] Benjamin Robert Liblit. *Cooperative bug isolation*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2004. AAI3183833.

[91] Cheng Liu, Lei Zhang, Yinhe Han, and Xiaowei Li. A resilient on-chip router design through data path salvaging. In *Proc. ASPDAC*, 2011.

[92] P. L. López, Juan Miguel Martínez, and Joeé Duato. A very efficient distributed deadlock detection mechanism for wormhole networks. In *Proc. HPCA*, 1998.

[93] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2), Feb 2002.

[94] John Markoff. Burned once, Intel prepares new chip fortified by constant tests. *New York Times*, November 2008.

[95] Milo Martin, Daniel Sorin, Bradford Beckmann, Michael Marty, Min Xu, Alaa Alameldeen, Kevin Moore, Mark Hill, and David Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4), 2005.

[96] Juan Miguel Martínez, P. L. López, José Duato, and Timothy Mark Pinkston. Software-based deadlock recovery technique for true fully adaptive routing in wormhole networks. In *Proc. ICPP*, 1997.

[97] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Nitin Vangal, Sriram Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC processor: the programmer's view. In *Proc. SC*, 2010.

[98] Rich McLaughlin, Srikanth Venkataraman, and Carlston Lim. Automated debug of speed path failures using functional tests. In *Proc. VTS*, 2009.

[99] Albert Meixner, Michael Bauer, and Daniel Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *International Symposium on Microarchitecture*, 2007.

[100] Albert Meixner and Daniel Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *Proc. DSN*, 2006.

[101] Andres Mejia, José Flich, José Duato, Sven-Arne Reinemo, and Tor Skeie. Segment-based routing: An efficient fault-tolerant routing algorithm for meshes and tori. In *Proc. IPDPS*, 2006.

[102] Giovanni De Micheli. Reliable communication in systems on chips. In *Proc. DAC*, 2004.

[103] Srinivasan Murali, Theocharis Theocharides, N. Vijaykrishnan, Mary Jane Irwin, Luca Benini, and Giovanni De Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design & Test*, 22(5), 2005.

[104] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proc. ISCA*, 2005.

[105] Lionel M. Ni and Philip K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 6(2), 1993.

[106] Sung-Jui Pan and Kwang-Ting Cheng. A framework for system reliability analysis considering both system error tolerance and component test quality. In *Proc. DATE*, 2007.

[107] Dongkook Park, Chrysostomos Nicopoulos, Jongman Kim, N. Vijaykrishnan, and Chita R. Das. Exploring fault-tolerant network-on-chip architectures. In *Proc. DSN*, 2006.

[108] Sung-Boem Park, A. Bracy, Hong Wang, and S. Mitra. BLoG: Post-silicon bug localization in processors using bug localization graphs. In *Proc. DAC*, 2010.

[109] Sung-Boem Park, T. Hong, and S. Mitra. Post-silicon bug localization in processors using instruction footprint recording and analysis (IFRA). *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 28(10), 2009.

[110] Priyadarsan Patra. On the cusp of a validation wall. *IEEE Design & Test*, 24(2), 2007.

[111] Andrea Pellegrini, Valeria Bertacco, and Todd Austin. Fault-based attack to RSA authentication. In *Proc. DATE*, 2010.

[112] Paul Dickinson Peter Dahlgren and Ishwar Parulkar. Latch divergency in microprocessor failure analysis. In *Proc. ITC*, 2003.

[113] M. Pirretti, G.M. Link, R.R. Brooks, N. Vijaykrishnan, M. Kandemir, and M.J. Irwin. Fault tolerant algorithms for network-on-chip interconnect. In *Proc. ISVLSI*, 2004.

[114] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1), 1997.

[115] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proc. ISCA*, 2002.

[116] V. Puente, J. A. Gregorio, F. Vallejo, and R. Beivide. Immunet: A cheap and robust fault-tolerant packet routing mechanism. *ACM SIGARCH Computer Architecture News*, 32(2), 2004.

[117] Bradley R. Quinton and Steven J. E. Wilton. Programmable logic core based post-silicon debug for SoCs. In *Proc. IEEE Silicon Debug and Diagnosis Workshop*, 2007.

[118] Samuel Rodrigo, José Flich, José Duato, and Mark Hummel. Efficient unicast and multicast support for CMPs. In *Proc. MICRO*, 2008.

[119] Samuel Rodrigo, José Flich, A. Roca, S. Medardoni, Davide Bertozzi, J. Camacho, F. Silla, and José Duato. Addressing manufacturing challenges with cost-efficient fault tolerant routing. In *Proc. NoCs*, 2010.

[120] Sean Safarpour and Andreas Veneris. Automated design debugging with abstraction and refinement. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 28(10), 2009.

[121] José Carlos Sancho, Antonio Robles, and José Duato. A flexible routing scheme for networks of workstations. In *Proc. HPCS*, 2000.

[122] Azeez Sanusi and Magdy A. Bayoumi. Smart-flooding: A novel scheme for fault-tolerant NoCs. In *Proc. SOCC*, 2009.

[123] Smruti R. Sarangi, Brian Greskamp, and Josep Torrellas. CADRE: Cycle-accurate deterministic replay for hardware debugging. In *Proc. DSN*, 2006.

[124] Jacob Savir. Syndrome-testable design of combinational circuits. *IEEE Trans. Computers*, C-29(6), 1980.

[125] Michael D. Schroeder, Andrew D. Birrell, Michael Burrows, Hal Murray, Roger M. Needham, Thomas L. Rodeheffer, Edwin H. Satterthwaite, and Charles P. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communication*, 9(8), 1991.

[126] Klaus-Dieter Schubert. POWER7 – verification challenge of a multi-core processor. In *Proc. ICCAD*, 2009.

[127] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Programming Languages and Systems*, 10(2), 1988.

[128] Jau-Der Shih. A fault-tolerant wormhole routing scheme for torus networks with nonconvex faults. *Information Processing Letters*, 88(6), 2003.

[129] Jeremy Siek and Lie-Quan Lee. BOOST graph library. `http://www.boost.org/doc/libs/release/libs/graph`.

[130] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. *ACM SIGARCH Computer Architecture News*, 20(1), 1992.

[131] Wei Song, Doug Edwards, José Luis Nunez-Yanez, and Sohini Dasgupta. Adaptive stochastic routing in fault-tolerant on-chip networks. In *Proc. NoCs*, 2009.

[132] Daniel Sorin, Milo Martin, Mark Hill, and David Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. ISCA*, 2002.

[133] Gregory S. Spirakis. Opportunities and challenges in building silicon products in 65nm and beyond. In *Proc. DATE*, 2004.

[134] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. The impact of technology scaling on lifetime reliability. In *Proc. DSN*, 2004.

[135] David Starobinski, Mark Karpovsky, and Lev A. Zakrevski. Application of network calculus to general topologies using turn-prohibition. *IEEE/ACM Trans. Networks*, 11(3), 2003.

[136] J. H. Stathis, B. P. Linder, R. Rodríguez, and S. Lombardo. Reliability of ultra-thin oxides in CMOS circuits. *Microelectronics Reliability*, 43(9-11), 2003.

[137] Chien-Chun Su and Kang G. Shin. Adaptive fault-tolerant deadlock-free routing in meshes and hypercubes. *IEEE Trans. Computers*, 45(6), 1996.

[138] Pao-Hwa Sui and Sheng-De Wang. An improved algorithm for fault-tolerant wormhole routing in meshes. *IEEE Trans. Computers*, 46(9), 1997.

[139] Pao-Hwa Sui and Sheng-De Wang. Fault-tolerant wormhole routing algorithm for mesh networks. *IEEE Computers and Digital Techniques*, 147(1), 2000.

[140] Sun microsystems OpenSPARC. http://opensparc.net/.

[141] Ming-Jer Tsai. Fault-tolerant routing in wormhole meshes. *Journal of Interconnection Networks*, 4(4), 2003.

[142] Babu Turumella and Mukesh Sharma. Assertion-based verification of a 32 thread SPARC CMT microprocessor. In *Proc. DAC*, 2008.

[143] Gert Jan van Rootselaar and Bart Vermeulen. Silicon debug: Scan chains alone are not enough. In *Proc. ITC*, 1999.

[144] Sriram R. Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Arvind Singh, Tiju Jacob, Shailendra Jain, Vasantha Erraguntla, Clark Roberts, Yatin Hoskote, Nitin Borkar, and Shekhar Borkar. An 80-tile sub-100-w teraFLOPS processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits*, 43(1), 2008.

[145] Ilya Wagner and Valeria Bertacco. Engineering trust with semantic guardians. In *Proc. DATE*, 2007.

[146] Ilya Wagner and Valeria Bertacco. Reversi: Post-silicon validation system for modern microprocessors. In *Proc. ICCD*, 2008.

[147] Ilya Wagner, Valeria Bertacco, and Todd Austin. Shielding against design flaws with field repairable control logic. In *Proc. DAC*, 2006.

[148] Chris Weaver and Todd Austin. A fault tolerant approach to microprocessor design. In *Proc. DSN*, 2001.

[149] Lee Whetsel. An IEEE 1149.1 based logic/signature analyzer in a chip. In *Proc. ITC*, 1991.

[150] Dennis J. Wilkins. Bathtub curve. `http://www.weibull.com/hotwire/issue21/hottopics21.htm`.

[151] Steven Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. ISCA*, 1995.

[152] David Wood, Garth Gibson, and Randy Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design & Test*, 7(4), 1990.

[153] Jie Wu. A fault-tolerant and deadlock-free routing protocol in 2D meshes based on odd-even turn model. *IEEE Trans. Computers*, 52(9), 2003.

[154] Jinlin Yang and David Evans. Automatically inferring temporal properties for program evolution. In *Proc. ISSRE*, 2004.

[155] Joon-Sung Yang and N.A. Touba. Expanding trace buffer observation window for in-system silicon debug through selective capture. In *Proc. VTS*, 2008.

[156] Yu-Shen Yang, N. Nicolici, and A. Veneris. Automated data analysis solutions to silicon debug. In *Proc. DATE*, 2009.

[157] Zhen Zhang, Alain Greiner, and Sami Taktak. A reconfigurable routing algorithm for fault-tolerant 2D-mesh network-on-chip. In *Proc. DAC*, 2008.

[158] Jipeng Zhou and F.C.M. Lau. Adaptive fault-tolerant wormhole routing in 2D meshes. In *Proc. IPDPS*, 2001.

[159] Jipeng Zhou and Francis C. M. Lau. Multi-phase minimal fault-tolerant wormhole routing in meshes. *Parallel Computing*, 30(3), 2004.

[160] Heiko Zimmer and Axel Jantsch. A fault model notation and error-control scheme for switch-to-switch buses in a network-on-chip. In *Proc. CODES+ISSS*, 2003.