Adaptive Distributed Architectures for Future Semiconductor Technologies

by

Andrea Pellegrini

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
2013

To my family and friends.

Acknowledgments

I would like to thank my advisor Professor Valeria Bertacco, who supported and guided me throughout most of my years at Michigan. I cherished the freedom she offered me to explore and develop new ideas. Moreover, my writing and presentation skills have grown significantly as a result of her mentoring and attention.

I am also grateful to my committee members. Professor Todd Austin has been a consistent source of inspiration throughout most of my research, and much credit goes to him for sowing the seeds of much of this dissertation. I would also like to acknowledge Professor Scott Mahlke for his technical insights and our friendly interactions. Last, but not least, I would like to thank Professor Silvio Savarese, who has been a dear friend for many years, and is leaving Ann Arbor at the same time as I am departing.

This dissertation would not have been possible without a number of people, who supported and helped me during my time in graduate school. I am thankful to have met some truly amazing people: Jason Clemons, Drew DeOrio, Joseph Greathouse, and Ilya Wagner. You have always been there, no matter the day or the time, whenever I was in need. Thank you so much for being such wonderful friends.

A special acknowledgement goes to all the students with whom I shared my office. I feel lucky to have been able to share all my frustrations and successes with you: William Arthur, Adam Bauserman, Kai-hui Chang, Debapriya Chatterjee, Kypros Constantinides, Shamik Ganguly, Jeff Hao, Chang-Hong Hsu, Andrew Jones, Doowon Lee, Rawan Khalek, Shashank Mahajan, Biruk Mammo, Andreas Moustakas, Ritesh Parikh, Robert Perricone, David Ramos, Steve Plaza, Shobana Sudhakar, and Dan Zhang.

A number of people contributed to making my experience at Michigan wonderful. It is people like you that make the CSE department one of the best places in the world for doing research: Professor Kevin Compton, Professor John Hayes, Professor Benjamin Kuipers, Professor Igor Markov, Professor Trevor Mudge, Professor Marios Papaefthymiou, Professor Karem Sakallah, Professor Tom Wenisch, Denise DuPrie, Laura Fink, Dawn Freysinger, Lauri Johnson-Rafalski, Stephen Reger, Timur Alperovich, Bashar Al-Rawi, Matt Burgess, Michael Cieslak, Hector Garcia, Valeria Garro, Anthony Gutierrez, Jin Hu, Hadi Katebi, Anoushe Jamshidi, Daya Khudia, Dave Meisner, Steven Pelley, Joseph Pusdesris, Jarrod Roy, Korey Sewell, Sara Vinco, Chien-Chih Yu, and Ken Zick.

Life in Ann Arbor would have been a lot less fun without Luca Cian, who never failed to make our evenings unique – I will surely miss you, my friend. I cannot count the fabulous memories I shared with Matt Burgess and Rick Hollander, you guys will be always in my heart. Thank you, Scott DeOrio, for being an amazing person. Thanks to Joe DeMatio, Mike Dushane, and Donny Nordlicht for letting me share a piece of your motoring Nirvana. Thanks also to a number of dear friends for their support: Filippo Bozzato, Guido Bozzetto, Sandro Colussi, Enrico Furlanetto, Flavio Griggio, Eric Hall, Alberto Lago, Paolo Leschiutta, German Martinez, Jessica Morton, Zamir Pomare', Ilario Scian, Scott Stuart, Fabio Tioni, Sever Tita, Silvia Tita, Mario Tomasini, Valeria Vavassori, and Massimiliano Zilli.

I would like to thank my family for all their love and encouragement: my mom Maria Assunta, my dad Adriano, my granddad Mario, my uncle Mario, my aunt Silva, my siblings Tommaso, Maria Diletta, Maria Grazia, and Giacomo. I would also like to thank three families that took pride in me and welcomed as their own: the DeOrios, the Gileses, and the Nashleys. I hope one day I will be able to give to others as much as you gave to me.

Finally, I would like to thank Natalia Amari for her endless support in the best and worst of times. Her unconditional love and support helped me to overcome the most daunting adversities and challenges.

Preface

As silicon technology continues to scale down transistor size, fundamental characteristics of this logic component are dramatically changing. Computer architects have grown accustomed to relying on a limited number of robust transistors; hence, they have focused on building machines that could achieve the best performance within a given transistor budget. Unfortunately, transistor features are shifting and future manufacturing processes are expected to integrate a massive number of extremely fragile components. This change imposes challenges that current computer architectures cannot overcome. Indeed, current microprocessor designs are not fit to work in these new technology scenarios and are restricted by their incapability to handle component failures, the inability to manage specialized hardware components and the lack of design modularity. This thesis develops a number of solutions for reliable and a adaptable computing, culminating in an original, distributed architecture that incorporates all these techniques to deliver better reliability and unlock increasing efficiency from the higher transistor density expected with future technologies.

The waning reliability of future transistors is the first concern. The consequences of this phenomenon are twofold: lower production yields due to higher rates of manufacturing defects and failures in the field. Neglecting runtime hardware faults can have dangerous consequences, as they could lead to service disruptions and corrupt program output.

Since core counts continue to increase, despite the stagnating energy efficiency of processors, researchers foresee that soon only a small fraction of a chip will be able to operate at full throttle. This leads to another issue stemming from future semiconductor technologies: computer architectures must accommodate a large number of hardware functional units and automatically adapt software execution to the subset of components that can be powered up based on the performance requirements of the application.

Finally, yet another challenge is the scarce design modularity of modern microprocessors. Current processors are large, monolithic systems. To provide design flexibility, future designs should be composable, meaning that their components can be organized in various combinations to satisfy specific user requests.

Solutions enabling the development of such architectures can allow future systems to

keep benefiting from the technological and economic advantages forecasted by Moore's law. However, in order to maintain these advantages, and thus continue to improve digital systems, we need to revolutionize how computers are designed.

This thesis first investigates the limitations of current processor designs, with particular focus on analyzing the consequences of unreliable components. Performing accurate fault analysis with traditional techniques is a tedious and time-consuming task. We directly address this issue with a complete framework to efficiently evaluate hardware malfunctions. The studies performed with this infrastructure expose the weaknesses in handling runtime failures, not only of current designs, but of state-of-the-art research solutions as well. These analyses also provide a deep understanding of how applications execute on hardware systems. Programs rarely stress all hardware units uniformly and microprocessor utilization varies greatly, even within the execution of a single application. Furthermore, long portions of a program often rely only on a few components. These observations guided the development of a low-cost adaptive reliability technique used to diagnose faulty components in a broad range of architectures, including those proposed in the latter part of this thesis.

We then propose to provide adaptability to hardware designs. An adaptable hardware system can alter itself to dynamically match software demands, environmental characteristics, and physical defects. With the target of providing this characteristic, we develop a distributed infrastructure that can be integrated in modern multi-processors, systems-on-chip and distributed systems alike. Chip components dynamically exchange information about their condition and utilization. This information is collected by a distributed software infrastructure, which reconfigures the design to match hardware functionalities and application needs without relying on a central manager.

Finally, to overcome the lack of design modularity, we propose a fully modular architecture. Our design organizes its hardware into a reconfigurable fabric of small, state-less modules. Each module can accomplish one or more services towards the execution of a portion of a program. Such a design greatly simplifies hardware organization, since each module is autonomous, and the number of available service providers does not affect the operations of the rest of the system. Thanks to its modularity and flexibility, it can achieve unprecedented reliability and adaptability. In our experiments, we found that these advantages are available at a moderate performance and power cost. Hence, it has the potential to reduce overall engineering costs of a system while also extending its lifetime.

The design methodologies and the novel execution paradigm developed in this research push the boundaries of modern digital microprocessor organization, enabling robust and efficient computing on even the most dense technologies and unreliable transistors. We believe that the adaptable and distributed designs developed in this dissertation can constitute a foundation for future computer architectures and empower further technological and economical growth for the semiconductor industry.

Table of Contents

List of Figures

List of Tables

Abstract

Year after year semiconductor manufacturing has been able to integrate more components in a single computer chip. These improvements have been possible through systematic shrinking in the size of its basic computational element, the transistor. This trend has allowed computers to progressively become faster, more efficient and less expensive. As this trend continues, experts foresee that current computer designs will face new challenges, in utilizing the minuscule devices made available by future semiconductor technologies. Today's microprocessor designs are not fit to overcome these challenges, since they are constrained by their inability to handle component failures by their lack of adaptability to a wide range of custom modules optimized for specific applications and by their limited design modularity.

The focus of this thesis is to develop original computer architectures, that can not only survive these new challenges, but also leverage the vast number of transistors available to unlock better performance and efficiency. The work explores and evaluates new software and hardware techniques to enable the development of novel adaptive and modular computer designs. The thesis first explores an infrastructure to quantitatively assess the fallacies of current systems and their inadequacy to operate on unreliable silicon. In light of these findings, specific solutions are then proposed to strengthen digital system architectures, both through hardware and software techniques. The thesis culminates with the proposal of a radically new architecture design that can fully adapt dynamically to operate on the hardware resources available on chip, however limited or abundant those may be.

Chapter 1
Introduction

Over the last five decades, improvements in semiconductor technology have allowed for a consistent increase in the number of devices that can be integrated in a computer chip. Such advances have been made possible through the systematic shrinking of the basic switching element used to build computers, the transistor. This trend has enabled designers to build progressively more powerful computers at a lower cost. In an effort to quantitatively demonstrate this growth, one can compare the change in cost over time of a machine that completes one billion operations per second – one GFLOP – a common metric for computer performance. The estimated cost for a machine of such capability decreased by 13 orders of magnitude over the span of only 50 years, from an estimated US $ 8.3 trillion in 1961 (inflation adjusted to 2013) to US $ 0.73 in 2013 [90, 194].

This incredible technological evolution was first described by Moore's law, which predicts that the number of transistors which can be integrated into a single chip roughly doubles every eighteen months [159]. In other words, Moore's law predicted that computers will be progressively faster and less expensive at the same time, and this has led to amazing social, technical, and economic consequences. Figure 1.1 quantitatively illustrates the dual benefits of such a technological advancement over a span of forty years for Intel's flagship processors. The blue curve in Figure 1.1 plots the number of transistors integrated in a single silicon chip, while the red curve reports the price in US dollars per single transistor built in an integrated circuit. The graph demonstrates the exponential enhancement of semiconductor technology in the past decades [91].

As a result of these advancements, computers have become ubiquitous in contemporary society. In fact, while only a few years ago, such systems were relegated to offices and server rooms, computers now have assumed central roles in virtually all aspects of our lives. For instance, digital systems are at the heart of mobile embedded devices, automobiles, airplanes, and even home appliances. The ubiquity of digital technology has enabled an unprecedented level of connectivity among people world-wide. This trend is well-captured by looking at the number of Internet users: almost 2.5 billion as of 2012 [92]. These figures are projected

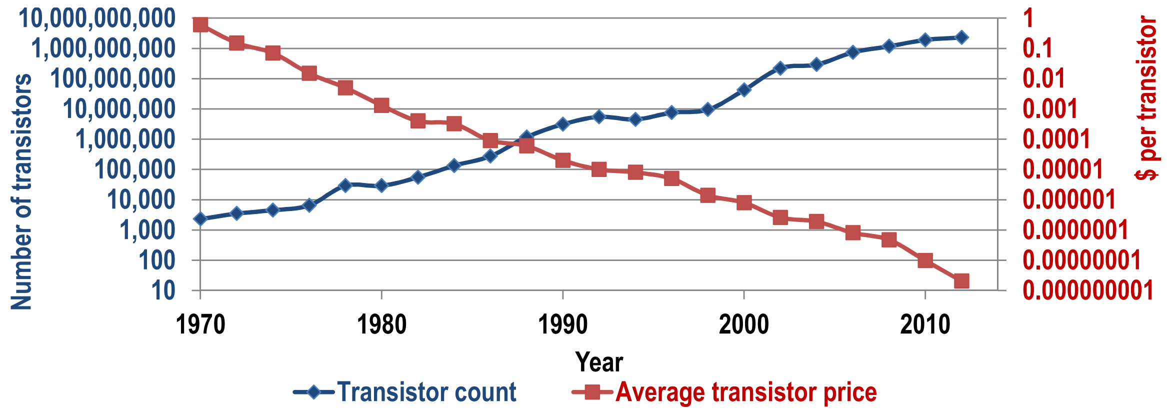


Figure 1.1 Number of transistor integrated into a singe chip and average cost per transistor over time. Moore's law is the observation that the number of transistors in an integrated circuit doubles approximately every two years. The plot also shows that, as more transistors are integrated into a chip, the individual cost of a transistor decreases exponentially over time. While a number of semiconductor manufacturers have been developing processors at the cutting edge of technology, for the sake of consistency here we only analyze the flagship Intel processors available year after year.

to increase even further as computer systems become smaller and more affordable. As these trends continue, technology will soon allow any physical device to be integrated into an information network and provide active input to improve human decisions [33].

Denser and more efficient transistors do not automatically translate into more powerful computer systems. The widespread adoption of these digital systems was only possible thanks to the effort of engineers to address a multitude of constraints such as performance, energy efficiency, and cost. Traditionally, computer architects have prioritized these characteristics over many others because they directly affected the commercial success of their designs. As a result, current processors feature sophisticated and complex mechanisms to achieve the best performance within limited power and cost budgets [15, 22, 140, 166].

While scientists continue on their quest to improve CMOS technology, they are also exploring and evaluating physical devices that go beyond classic silicon transistors, envisioning machines that could employ spintronics, optical communication, graphene, nanotubes, or even quantum computations [20, 59, 76, 84, 185, 215]. Even before considering the repercussions of adopting new exotic technologies, experts envision the next integrated circuits as complex machines composed of several billions of minuscule and fragile physical devices [21, 24, 185]. Beyond their intrinsic fragility, the increasing device density raises both electric current and power density, phenomena which accelerate the wear of already defect-prone nanoscale transistors even further [184]. As technology continues to evolve, the characteristics of the underlying logic components are dramatically changing: from limited and robust to plentiful and fragile. This leads to a scenario in which current architectures and design methodologies are no longer adequate.

The relative abundance of components has already significantly changed the design of shipping products. While engineers used to focus on single, monolithic central processing units, the last decade has seen semiconductor companies moving to designing entire systems-on-chip. Unfortunately, because component count is growing much faster than transistor power consumption is reducing, the percentage of a silicon chip that can operate at full frequency diminishes with each process generation. Therefore, in order to stay within a reasonable power budget, the vast majority of future chips will be effectively turned off, idle, or significantly under-clocked at any given time. With the goal of continuing to improve user experience and performance growth, researchers already predict that future computer chips will embed various hardware modules dedicated to specific applications. Hence, a system must dynamically ensure that only components directly contributing to the executing applications are active [52, 188, 190, 198].

Finally, the increasing number of transistors that enables the construction of microprocessors with more components also leads to constant growth in design size. Current microprocessors are monolithic systems difficult and expensive to design and customize. As the number of components in a chip is expected to increase with future technologies, this problem can only worsen [133].

Based on these trends, we identify three barriers that are limiting the advancement of future computer systems: i) the increasing fragility of the silicon substrate, ii) the challenges associated with the dynamic management of the numerous specialized components that comprise upcoming digital systems, and iii) the limited design modularity that impairs designer's ability to manage the system's development effectively. The goal of this thesis is to develop solutions to address these concerns enabling the design of high performance and highly efficient computer systems.

The first concern is the increasing fragility of future transistors. Current semiconductor technology can manufacture a transistor channel with less than 100 silicon atoms. Therefore, as transistor dimensions shrink further, even minute variations in a transistor's molecular structure may cause fatal breakdowns. The consequences of neglecting this phenomenon are twofold: production yields are plummeting, due to higher rates of manufacturing defects, and hardware's failures in the field due to faulty transistors are going to increase [61, 184]. Lower yields cause higher manufacturing costs, while runtime hardware faults can impact servicing costs and the overall existence of a manufacturer.

The second problem facing the computer industry lies in the dynamic management of the increasing diversity of specialized hardware components available in modern designs [13]. In the past decade, the design focus has shifted from the development of high-performance general-purpose processors to specialized devices targeting particular functions, so as to speedup workload execution while minimizing overall power consumption. In this new technological scenario, engineers trade silicon real-estate for higher performance and energy efficiency. This trend is affecting the way computer systems are designed, and researchers already envision that future processors may be able to dynamically trade among different implementation of specialized functional units to unlock the ability to improve power and energy. While design specialization provides significant benefits to digital systems, classic processors have limited flexibility and do not include mechanisms to automatically enable and deactivate specialized functional units based on application demands.

The third and last challenge is the increasing size and intricacy of modern microprocessors, which has reach the point that the correct design of a modern processor is an unattainable goal. Because of this complexity the number of new developments is decreasing every year. This monolithic design approach not only increases overall engineering costs, but has also already caused some high end development efforts to be canceled because the engineering team could not bring them to work correctly and consistently.

To continue to leverage the technological and economic advantages predicted by Moore's law, future computer architectures must tackle these important concerns. This work explores and evaluates novel solutions to target these challenges. To address the increasing fragility of silicon components, we propose a novel methodology to detect hardware failures at runtime. In this design, processors are periodically tested to assess the health of the hardware modules exercised by user applications. This allows a system to diagnose failures that may affect software state before they corrupt program output.

In order to deal more easily with highly diverse hardware designs, we propose a framework to enable hardware adaptability on any multiprocessor system. In such a design, components periodically broadcast diagnostic messages in order to report their availability and advertise their features. Such messages enable the entire system to know the state of all hardware elements in the chip and dynamically schedule hardware resources to match application demands. Besides simplifying the process of adding new functional units to a design, this mechanism can be used to keep power consumption under control, since this information can be used to activate, tune, and disable hardware components based on runtime software needs.

We then propose a fully modular microarchitecture: its components are connected through a common interface, and engineers can build a new system solely by relying on libraries of reusable task providers. Each component can then execute one or more tasks toward the completion of a program's instruction. Instructions executing on this machine are not pushed through hard-wired paths defined at design time, as classic processors would do. Instead, components dynamically connect to form a hardware configuration that can perform all the tasks required by an instruction. In order to avoid reliance on complex and pervasive centralized logic, hardware configurations are controlled through a number of independent scheduling units. The sole purpose of these units is to establish and manage system configurations that can perform all tasks needed by the instructions in a program. This novel execution paradigm reduces complexity because components interact exclusively through standard point-to-point communication media. Furthermore, since all hardware components in the proposed architecture only work on small groups of instructions, local state is kept separated and has a very short life-span. These traits greatly simplify the interaction among components in our architecture and allow engineers to design, optimize, and validate portions of the system in isolation.

Finally, we present a complete system that coheres all the solutions proposed in this thesis to build a reliable, adaptable, and modular design. Furthermore, thanks to its modularity and flexibility, this holistic solution can reach even higher levels of reliability and adaptability. Through experimental evaluation, we found that these advantages can be achieved at a very modest performance and power impacts.

In summary, the design methodology and architectures developed in this research push the boundaries of modern digital microprocessor organization, enabling robust, adaptive and high performance computing on unreliable devices.

1.1 Challenges

As manufacturing capabilities keep evolving and transistor density keeps growing, a significant transition in the semiconductor technology scenario is quickly approaching. Future computer systems are facing a series of new important challenges [23]. While prior research and designs addressed issues such as single-threaded performance and power consumption, other issues still require attention. As technology progresses further, classic techniques will not allow for the exponential improvements in performance and efficiency historically experienced by computers in the past decades. Technology experts agree that the most significant problems hindering the advancement of future microprocessors are: the increasing fragility of the manufactured components [21], the challenges connected to managing specialized hardware [198], and the lack of design modularity [25]. This section will analyze each one of these three challenges in detail.

1.1.1 Hardware Failures

The reliability of future designs is threatened both by increasing transistor counts as well as by the growing fragility of individual components. Looking ahead, upcoming semiconductor technologies will allow silicon chips to include tens of billions of transistors. Unfortunately, a significant fraction of these transistors may be faulty. The high device density enabled by contemporary technologies has already led to high current and power surges, which damage or may destroy the already fragile nanometric transistors [21, 23].

Some of the most worrisome sources of hardware failures are: variations in supply voltage or local temperature, energy particles hitting silicon components, transistors degradation over time [75, 96, 131]. Any of these events may cause either transient [49] or permanent hardware failures [184]. Transient faults are temporary upsets of a circuit and can alter a system's state, potentially corrupting its results [11]. These upsets do not damage the silicon device and therefore can typically be resolved by resetting the affected system or by re-executing the erroneous computations. Unlike transient upsets, permanent faults are irreversible alterations of physical hardware elements. These defects may either be introduced during manufacturing (and escape post-production tests) or may appear after a system is deployed. Permanent failures manifesting in the field are often caused by transistor wearout due to: electromigration [70], time dependent dielectric breakdown [51], or negative bias temperature instability [214]. Neglecting runtime hardware failures – both transient and permanent – can have dangerous consequences since they can lead to expensive service disruptions [107] or silently corrupt a program's outputs [145]. In order to tackle the increasing number and variety of hardware failures, future architectures must include mechanisms to dynamically diagnose faulty hardware and around it to bypass it, so that computation can continue to progress correctly.

1.1.2 Management of Specialized Components

Future microprocessors will likely comprise a diverse range of specialized components. In contrast to historical designs, which focused on maximizing the performance of general purpose cores [2], future designs must specialized hardware accelerators to achieve performance and efficiency. In many respects, the advancement of state-of-the-art microprocessors over the last decades has been steady and evolutionary. Around 2001, the switch to multicore designs allowed semiconductor manufacturers to enhance overall performance while keeping design complexity manageable and power consumption under control. Today, the most advanced multicores can, in aggregate, execute up to hundreds of programs at the same time [34, 166].

Since core count continues to increase despite stagnating energy efficiency, researchers already foresee that only a small fraction of a chip will soon be able to operate at full throttle [190]. Furthermore, recent studies have shown that, in the near future, technology scaling by itself can contribute at most 15% per year to the performance enhancement of highly concurrent applications. These figures have already been confirmed by recent trends in high performance commercial microprocessors. As of 2013, initial analyses of the newly announced Intel Haswell microarchitecture reported that this new chip improves single thread performance by 10 –15%, compared to previous generation's processors [194]. Such performance improvement is only a fraction of the generation-to-generation performance boost achieved in the past.

With the aim of continuing to increase overall performance, researchers have proposed designs which integrate specialized components to accelerate particular software applications [188, 198]. This new design paradigm disrupts the well-established microprocessor design cycle. Traditionally, computer architects focused their efforts on maximizing performance per unit of silicon area, and therefore strived to design general-purpose processors that could be economical and execute a broad range of applications efficiently. Thanks to the high integration density offered by current and future semiconductor technologies, designers can now embed a large number of diverse hardware features targeting particular functionalities, such as specialized adders, multipliers, FPUs, and SIMD units. This trend offers the opportunity to effectively improve both performance and power efficiency at the cost of a larger silicon device.

Future processors must harness the performance and efficiency advantage provided by these numerous and diverse functional units in order to continue improving user experience and computational capabilities. Hence, a computer architecture that can thrive in the scenario imposed by future semiconductor technologies must: i) accommodate a large number of specialized functional units, ii) allow applications to take advantage of them even for short functions, and iii) automatically match application needs with available features. These characteristics would enable future architectures to further improve their efficiency.

1.1.3 Lack of Design Modularity

In 1971, a handful of engineers developed the first microprocessor, the Intel 4004. Today, the design of a state-of-the-art processor requires hundreds of people working together for years. Computer architects exploit increasing transistor density to introduce more advanced features – such as complex pipelines and speculative execution – which can exploit instruction- or memory-level parallelism to accelerate software applications [82, 86, 178]. These features significantly increase the number of components embedded in a processor core, and, consequently, add pressure to the overall design effort. Although in the past decades computer-aided design tools have been rapidly improving, they still cannot match the vast sophistication of modern computer chips.

Modularity refers to the use of smaller, independent design entities to create customized versions of products. As companies strive to manage engineering effort to produce a large variety of products at lower cost and shorter time-to-market, modularity is becoming a major design focus [88]. Over the past years, as integrated circuits have become increasingly complex and expensive, designers have started to embrace design methodologies focused on modularity and component reuse. For instance, systems-on-chip (SoCs) are commonly designed using these techniques. Modular components are sufficiently general and configurable so to be deployed in a wide range of applications [158]. While a number of components in modern digital systems are modular, microprocessor designs are still monolithic large units, hence they are difficult to specialize to different users' requirements.

1.2 How to Address These Issues?

Current computer architectures, such as multicore processors, provide short-lived and partial solutions for mitigating the increasing fragility of the manufactured components, for managing large and diverse designs and for the lack of modularity. The adaptive distributed architecture presented in this dissertation strives to tackle these challenges through the development of novel solutions that, once combined, can form a reliable, adaptable, and modular computer design.

We have already introduced the need for reliable computing, which is due to the rapid degradation of transistor robustness. We start by quantifying the impact of transistor failures on current designs and develop solutions to efficiently diagnose faulty hardware at runtime. We then shift our effort to providing adaptability. Adaptability enables a hardware system to dynamically adjust its operations to match workload demands and available resources. To achieve this goal, we develop a low-cost mechanism to distribute and manage diagnostic information about hardware components and application characteristics. We then concentrate on developing a novel microarchitecture that primarily promotes modularity – the third research direction that we explore to unlock the design of future computer architectures. Modularity allows systems to be easily reconfigurable to match different users' requirements.

Once we develop solutions to address all three challenges, we finally aggregate them to

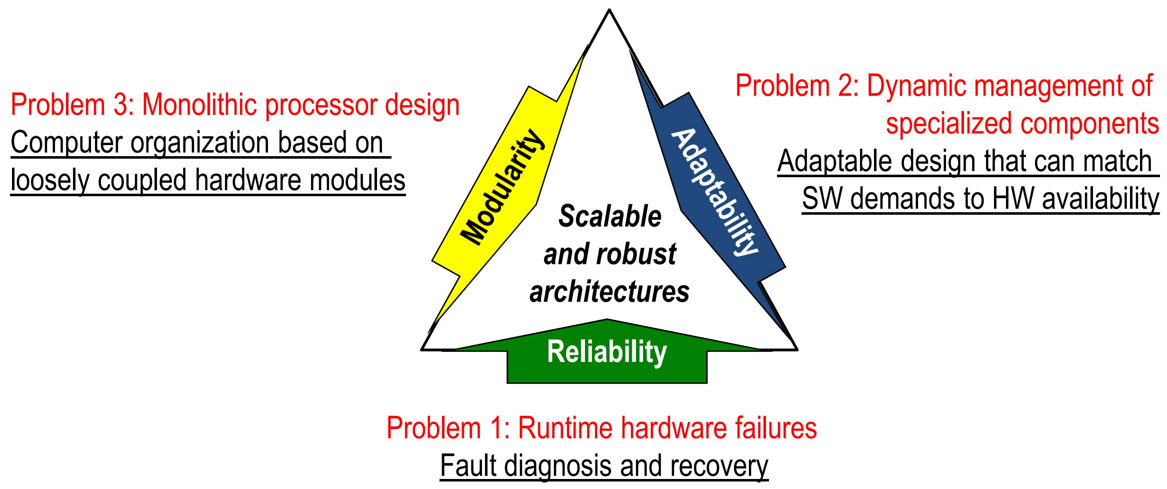

Problem 3: Monolithic processor design
Computer organization based on loosely coupled hardware modules
Modularity
Scalable and robust architectures
Adaptability
Problem 2: Dynamic management of specialized components
Adaptable design that can match SW demands to HW availability
Reliability
Problem 1: Runtime hardware failures
Fault diagnosis and recovery

Figure 1.2 Focus of this dissertation. The objective of this thesis is to develop novel computer architectures that address the challenges imposed by the transition to future semiconductor technologies: hardware failures, management of specialized components and lack of design modularity. In order to do so, this thesis explores, develops, and evaluates a number of reliable, adaptable, and modular solutions for distributed computer architectures.

form a hardware system which embodies them in one comprehensive solution. The resulting proposed architecture, the final goal of this dissertation, offers much more than its parts. Indeed, each solution often complements and enhances the others, therefore boosting the overall capabilities of the complete system. The goals driving the development of our adaptive distributed architecture are discussed in the remainder of this section and summarized in Figure 1.2.

1.2.1 Reliability

To address the increasing fragility of future transistor devices, previous research focused on developing flexible and robust computer architectures. These solutions provide both component redundancy and fault containment. Multicore designs enable simple mechanisms for redundant execution and to isolate faulty processors [127]. Other works focus on maximizing the lifetime of components that are naturally redundant, such as caches and other arrays of regular structures [26, 171]. BulletProof improves reliability on processors that contain multiple functional units, allowing for a design to gracefully degrade its performance as the number of faulty transistors increases [172]. StageNet is a more radical approach to reliability, which proposes a reconfigurable in-order control-flow architecture connecting multiple identical hardware components extracted from a traditional multi-core design [73]. Hence, faulty components can be individually disabled and the system can be reconfigured to run programs only on the available hardware.

Design for hardware reliability plays a pivotal role in our work, and the solutions for reliable computing presented in this thesis not only protect applications and users from dangerous hardware failures, but can also be leveraged to improve production yield. Therefore, our techniques can extend a system's lifetime while, at the same time, reducing overall costs.

We set off to investigate the limitations of current processor designs, with particular focus on analyzing the effects of hardware failures in modern computers. Performing accurate fault analysis with traditional techniques is a tedious and time-consuming task. Our CrashTest solution addresses this issue, providing a complete framework for effectively evaluating hardware malfunctions [146]. The studies performed with this infrastructure expose the weaknesses in handling runtime failures for both current designs and state-of-the-art solutions [145, 148]. These analyses also provide a deep understanding of how software applications execute on hardware systems. Based on them, we observe that programs rarely stress all hardware uniformly, and a microprocessor's utilization varies greatly even within the execution of a single application. Furthermore, our experiments show that long portions of a program rely on only a few components. These observations have guided the development of a low-cost adaptive technique for reliability that exploits an application's behavior to efficiently protect users against hardware failures [142]. This solution measures the usage of the different hardware modules to achieve high fault coverage and minimize the time spent in online testing. The deployment of this diagnostic solution is only the first step towards a completely reliable system. The next necessary step is the reconfiguration of the hardware system around broken components, so as to allow programs to work around hardware failures. This goal can be attained through our second research direction, adaptability.

1.2.2 Adaptability

To address the problems arising from the growing diversification of system components and features, hardware could adapt its operations to optimize applications' utilization of the resources available. Compared to simply adapting the software applications – for instance through just-in-time compilation – adaptive chips can use runtime information to activate and tune hardware modules and better respond to user's demands. Furthermore, such designs can adapt to a wide range of environmental conditions, such as external temperature variations or power sources ranging from a distribution network supply to a portable battery [183].

Adaptability is also fundamental to ensure that a system does not exceed its maximum power envelope. In fact, an adaptable system can actively reduce power consumption by adjusting hardware operations. For instance, one capability of an adaptive system could be to activate specialized components only if required by application's demands and immediately turn them off when they are not utilized. Furthermore, applications can take advantage of the ability to adjust hardware use at runtime to optimize for performance or power or other characteristics. Indeed, state-of-the-art commercial microprocessors recently started to include basic self-monitoring and self-controlling capabilities to manage their resources [44, 89, 119, 121]. Finally, the physical elements of a hardware system may be damaged by permanent defects. An adaptable system helps overcome such failures by isolating broken components and reconfiguring the system to work around them.

With the aim of providing complete system adaptability, this thesis presents a solution that can automatically manage computer resources. Chip components dynamically exchange information about their condition and utilization. These diagnostic messages are collected by a software distributed system, which reconfigures the design to match hardware functionalities and application needs without relying on a central manager (which would constitute a single point of failure) [143]. Our solution relies on local hardware detectors to run periodic tests on the silicon components, while delegating system reconfiguration, when necessary, to software routines. A distributed software resource manager collects these diagnostic notifications and dynamically reconfigures the hardware to maximize performance and utilization. Such partitioning of tasks provides an efficient response to hardware failures.

The techniques developed to achieve adaptability can be applied to any computer system. Thus, we first explore and evaluate our solution on a classic chip multiprocessor system. We then extend it to make it applicable on the distributed architecture that brings together all the solutions discussed and that we propose at the end of this dissertation.

1.2.3 Modularity

The struggle to enhance computer performance has driven engineers to create sophisticated, monolithic microprocessors. As a result of these design choices, the engineering effort necessary to design a modern state-of-the-art processor is enormous. Indeed, industry experiences report that designing and validating a modern microprocessor requires hundreds of man-years [17]. A second disadvantage of classic monolithic processors is that these designs are generally difficult, if not impossible, to customize to fulfill users' requirements.

An effective and practical solution to contain design costs while enabling customizability is to build modular systems by combining smaller, simpler components. Components in a modular system are independent and share few direct connections with each other. Therefore, they can be freely recombined and reorganized to match different design specifications. This property is particularly relevant for achieving scalability and reconfigurability, since modular systems can grow in size by simply adding more components. Indeed, a modular system is easily customizable and distinct designs can be created by including various combinations of components that satisfy different requirements.

This thesis introduces a novel architecture that provides modularity by design. This new architecture is based on a distributed execution engine that is dynamically configured to route instructions towards available hardware components [147]. This allows our solution to scale performance gracefully as the number of resources increase. Instead of pushing instructions through paths defined at design time, as classic processors do, this architecture relies on a flexible fabric composed of independent hardware components. These components are loosely coupled via a homogenous communication network to form a dynamic and reconfigurable execution engine. Each component can accomplish one or more services toward the completion of an instruction. They are fully autonomous and the number and type of service providers do not alter the underlying functioning of the other modules in the system. In such architectures, a program can always successfully execute, as long as the working hardware components can, in aggregate, execute all of its instructions. This flexibility is provided at a very modest cost, as our proposed system is only 20% slower than a traditional multi-core and it uses only 15% more power in the worst case.

The execution model and the modular design developed here constitute the backbone of our adaptive distributed architecture. As we discuss in the next section, the modularity provided by our design can also further enhance both reliability and adaptability.

1.3 A Comprehensive Reliable and Adaptive Distributed Architecture

To complete the research discussed in this dissertation, we present a comprehensive reliable and adaptive system that can thrive in the scenarios imposed by future semiconductor technologies [144]. This solution organizes hardware components into a reconfigurable fabric of small, state-less units. The design targets highly parallel workloads, and it promotes reliability, adaptability, and modularity as top-priority design foci.

Our dependable architecture tolerates a large number of transistor failures and gracefully degrades performance as faults accumulate in its hardware. It directly leverages the techniques developed in this research to diagnose failures in its components. Software requiring maximum reliability can either make use of fully redundant execution or of periodic hardware tests. Meanwhile, applications that wish to maintain high performance can protect only the most vulnerable portions of their programs. Alternatively, software that does not require correctness guarantees can disable all online reliability mechanisms for a performance benefit. Each application executing on the platform can employ the reliability feature that best suits it, independently from all others. Hardware modules detected as faulty are disabled without affecting the operation of the rest of the system. Furthermore, the dependability of our processor can be arbitrarily improved by adding extra copies of each hardware module to the sea of execution units. Even if one hardware component fails, the system can still make forward progress, as long as there are others available to accomplish the same task – or as long as the software does not require any task to be carried out by the damaged units.

The system proposed at the end of this dissertation deploys the solutions that we have outlined so far and that will be discussed in details in the following chapters. Specifically, it leverages our novel low-cost mechanism to enable adaptability and flexibility in any microprocessor system. This mechanism is fully distributed and relies on the periodic exchange of diagnostic messages among all components of the system. A subset of such components collects the information carried by these messages and uses it to acquire global knowledge about a system's dynamic state to improve resource scheduling. Thanks to this mechanism, our solution can be easily upgraded to include new hardware functionalities and task-specific accelerators, providing great opportunities to improve and facilitate design modularity. Furthermore, our architecture can dynamically adapt program execution to make the best use of hardware components to fit an application's needs, while also considering other programs demands. In doing so, our system has the potential to improve hardware utilization and overall efficiency.

Finally, the proposed architecture is modular, providing designers with the capability of customizing a system and scaling its throughput. It enables designers to arbitrarily split the design into independent modules which, in aggregate, can execute all the tasks required by an instruction set. Since these components only interact through explicit messages and their state is only influenced by their inputs, systems implementing the proposed design avoid reliance on physically centralized control logic. The number of microarchitectural structures in our modular design can vary in number as components are added or removed from its fabric. For all these reasons the architecture proposed has great potential to increase component reuse, hence reducing overall engineering costs and increasing customizability.

In summary, the solutions developed in this dissertation address the key three challenges that undermine the adoption of future semiconductor technologies: increasing transistor fragility, wide design diversification, and lack of modularity. These properties are achieved at a moderate performance, area and power costs. Overall, we believe that the design methodologies and the novel distributed execution paradigm developed in this research push the boundaries of modern digital microprocessor organization, enabling robust, adaptive and high performance computing on unreliable devices.

1.4 Dissertation Organization

The remainder of this dissertation is organized by system properties. Chapter 2 analyzes the motivations that drive this research. It describes and justifies the need for a novel, reliable, adaptable, and modular computer architecture. We address each of these three properties in Chapters 3, 4, and 5.

Chapter 3 addresses system reliability. It first analyzes the issues that arise due to the increasing fragility of integrated circuit components. Performing accurate fault analysis is traditionally a tedious and time-consuming task. The first contribution of this thesis, CrashTest, addresses this issue and provides a complete framework that enables designers to easily and accurately evaluate the effects of hardware malfunctions on live digital systems [146]. The studies performed with this infrastructure expose have weaknesses of current designs and even of state-of-the-art research solutions in dealing with runtime failures [145, 148]. They also led to the development of software techniques for reliability that adapt to the characteristics of both the software workloads and the available hardware components [142]. In the first part of Chapter 3, we rely on an experimental framework based on a system including one reconfigurable hardware device, which is configured to model an industrial-grade microprocessor, the Sun OpenSPARC T1. The gate-level description of this design is altered to inject a number of fault locations, and our experiments report the percentage of faults diagnosed by software symptom-based fault detection techniques. The second part of this chapter presents a novel solution to diagnose runtime hardware failures. In such a design, software execution is periodically interrupted to allow focalized hardware self-checks to test the hardware modules exercised by user applications. We evaluated our solution on the OpenSPARC T1, and used TetraMax, a commercial automatic test pattern generator, to measure the fault coverage achieved by our novel dynamic testing technique. Additionally, we utilized Virtutech Simics to measure the performance overhead of our latter solution.

Hardware adaptability is addressed in Chapter 4. Runtime hardware failures might change a system's functionalities. To address this issue, Chapter 4 develops a distributed and introspective mechanism that can oversee and reconfigure a hardware system. In this solution, the various components of a chip dynamically exchange information about their condition and utilization. Components then form a distributed system that can self-adapt to hardware availability and software needs [143]. To evaluate this solution, we developed a C++ model that allowed us to measure the time necessary for our solution to adapt to sudden hardware changes. We also evaluated the performance overhead of our design on a complete system modeled through the gem5 microarchitectural simulator.

Chapter 5 focuses on design modularity. It details the architecture of a system that overcomes the limitations of current processor designs, which are based on centralized control logic. This chapter develops a novel distributed computer architecture that breaks apart the classic concept of a microprocessor, dissolving its components into a reconfigurable fabric of smaller, interchangeable hardware units [147]. In such a machine, program execution does not rely on any centralized physical structure, and thus its size and performance can be easily changed by adding or removing more modules. The performance of this microarchitecture has been evaluated in the gem5 microarchitectural simulator. Furthermore, we used a variant of McPat to measure power consumption and area overhead of the design.

After addressing each of these three challenges separately, Chapter 6 brings together the key contributions of this dissertation in a system targeting highly concurrent applications. The culmination of this research is a comprehensive robust, adaptive, and modular system [144]. It leverages the reliability techniques explored in Chapter 3, adopts the solutions for hardware adaptability developed in Chapter 4, and builds on the new, distributed execution paradigm introduced in Chapter 5. Furthermore, this design is completely modular, providing architects with the capability of customizing a system. A gem5-based evaluation platform was used to evaluate its resiliency to faults and the graceful degradation of performance in presence of faults.

Finally, Chapter 7 summarizes the contributions of the dissertation and discusses future research direction.

Chapter 2
Motivation

Future semiconductor technologies will drive transistor miniaturization even further, enabling the next generation of computer chips to integrate tens of billions of transistors. In the past, Dennard scaling caused the shrinking of transistor features to directly translate into both energy and performance benefits. Because we have reached the scaling limits for device voltage, current and future semiconductor technologies can no longer take advantage of this physical phenomenon [52, 85]. Since solely scaling down transistor sizes does not automatically result in enhanced performance or energy, designers must focus on new architectural solutions to continue improving hardware systems and sustain the technological and economic growth of the semiconductor industry.

Until the early 2000s, computer architects invested the largest fraction of transistors to increase the performance of general-purpose CPUs. As technology allowed higher integration densities, it often made no sense to add further transistors to a single core, and therefore designers began to include different hardware features into a single integrated chip, effectively building entire systems on a single chip (SoC).

Since future semiconductor technologies are expected to revolutionize transistor properties, the characteristics of modern computer designs must also shift. Both academic and industry research trends are moving towards architectures that sport an astonishing number of heterogeneous hardware features [23]. On one hand, this technology trend allows computers to harness available silicon real estate and improve performance while decreasing overall power consumption [79]. On the other hand, current CPU architectures cannot manage the high number of specialized functional units necessary to improve efficiency [198], and have limited design modularity [25]. To make things worse, smaller transistors and higher integration levels are expected to increase device fragility [21]. This chapter details the problems that motivate our research. With the goal of addressing these challenges, this dissertation proposes solutions to allow future architectures to thrive in the semiconductor landscape to come.

2.1 Chapter Organization

This chapter details the three technological challenges that we address in this thesis and defines the targets we want to achieve with our proposed solutions. Sections 2.2, 2.3, and 2.4 discuss the three problems tackled in this thesis: the increasing fragility of the manufactured components [21], the dynamic challenges related to managing specialized hardware functional units [198], and the lack of design modularity [25]. Section 2.2 also details a study performed to assess the dangers of not planning for possible runtime hardware failures in a design. Finally, Section 2.5 presents the strengths and shortcomings of prior solutions attempting to address these issues.

2.2 Hardware Failures

Hardware failures are physical flaws in the hardware of a computer system, such as interrupted wires and defective transistors. Some of these failures may manifest in the system as errors, for instance switching the value of a bit. Such errors may appear immediately or remain dormant for long periods of time. Other failures may never affect a computer behavior and effectively be masked by the different layers (circuit, architecture, OS, software) of a modern computer system [175]. Technology experts have already warned that future digital systems will be threatened by a rising number of hardware failures, due to both the growing transistor count and the increasing fragility of individual devices [21]. Several physical events may trigger hardware failures, which are typically divided in two categories: transient and permanent.

Transient

Transient failures are temporary upsets of digital systems. They are typically associated with three environmental events: cosmic radiation, alpha particles, and electromagnetic interference. The Earth is constantly subjected to showers of cosmic rays, which are very high-energy particles originating in outer space. Cosmic rays interacting with the atmosphere convert atmospheric elements into a shower of secondary particles that irradiate the entire planet [217]. Alpha particles, instead, are generated by the natural decay of radioactive materials, often from materials constituting the package of the chip itself [117]. When either these particles hit a functioning computer system, they may alter the electrical charge used to store information in its circuitry, and hence potentially corrupt its computations. The third source of transient failures is electromagnetic interference from internal and external sources. This disturbance is introduced by the fact that any object subjected to rapidly changing electric currents emits electromagnetic radiation that can, in turn, be captured by the inductive components of an electronic device. It is worth noting that this interference might be generated by artificial objects such as electronic devices, or natural events such as lightning.

Future semiconductor technology nodes will be more susceptible to transient failures, since tiny transistors will store only small amounts of electric charge, and thus even particles with low energy levels may have an altering impact [14, 49]. While neglecting the effects of transient failures can yield to the severe consequences detailed in Section 2.2.1, these faults do not permanently alter the physical elements of a computer chip, hence they can be corrected simply by re-executing the corrupted operations or by resetting the system to a safe state.

Permanent

Permanent faults are irreversible modifications of the hardware components of a chip. Such defects may be caused by imprecisions in the manufacturing process or may be triggered by the wear of the physical components in the chip. Producing a chip requires a multitude of steps involving complicated optical, chemical, and mechanical processes. Regrettably, imprecisions in any step of this procedure, for instance due to temperature or chemical variations, might result in defects in the chips produced. While most defective chips are detected during post-production quality tests, some faulty chips might slip through these checks and exhibit erroneous behaviors in the field [175]. Physical wearout is the second source of permanent defects. Continuous usage, mechanical stress, and high operating temperatures are some of the phenomena that contribute the most to this source of permanent failures. Both wires and transistors can be subjected to wearout [75, 184]. Wires can be physically consumed by electromigration, which is caused by the gradual movement of the ions in a metal due to the kinetic energy of the electric charge moving within. The physical parts composing a CMOS transistor may also break over time. For instance, high energy charge carriers may stray out of the conductive channel between the source and drain and get trapped in the insulating dielectric between them [207, 208], effectively changing the characteristics of a transistor. Another aging factor for CMOS devices is due to charges that accumulate within the dielectric that separates the gate from the silicon substrate, thus short-circuiting the gate and the channel [161].

As feature size in future semiconductor technologies is expected to decrease even further, the practical significance of these effects increases significantly, and so does the probability that a digital system might be affected by a permanent failure in the field. Overcoming these failures is a three-stage process consisting of: i) fault detection, ii) fault isolation through hardware reconfiguration, and iii) system state restoration [175]. Fault detection and design reconfiguration granularity determines the number of failures a system can work around. For instance, most computer manufacturers provide the ability to disable faulty memory lines and cache cells through the addition of spare memory elements. This technique of isolating broken structures or supplementing a design with spares has also been extended to logic resources, up to the point of disabling entire cores. Large portions of this dissertation focus on innovative solutions to addressing these failures, which constitute a serious threat to the functioning of future computer systems.

2.2.1 Impact of Runtime Failures

The impact of transistor failures on microprocessors has been thoroughly studied in the literature [98, 180]. Current computers are very sensitive to hardware failures that affect their memory subsystems, and recent research has shown that information computed and stored in large data centers can be corrupted by hardware faults that escape error detection and correction mechanisms [134, 162]. Besides errors in the memory elements, reliability of future processors is also threatened by the growing fragility of semiconductor devices. As transistor continue to reduce in size, combinational elements also become vulnerable to hardware failures, further exacerbating the risk of computer errors [49]. For instance, large scale studies have already shown that existing processors are susceptible to error rates that are orders of magnitude higher than previously assumed [129].

Traditional solutions for high-availability and mission-critical computers address reliability through dual or triple-modular component redundancy [12]. However, these solutions are far too costly to be adopted in mainstream commercial applications. Because most computer systems do not have active mechanisms to overcome runtime faults, they can face critical failures when defects manifest during system operation. Two errant computer behaviors are particularly worrisome: **system unavailability** and **silent data corruptions**.

System Unavailability

In most cases, hardware runtime faults are known to cause software disruption [107]. Such behavior causes computer systems to become unavailable and therefore stall or even subvert infrastructures that rely upon them. Numerous examples of service interruption due to computer failures appear in the news every year, and costs can top millions of dollars per minute of downtime.

For instance, in 2009, a hardware glitch in a single computer board at Salt Lake City International Airport prevented communication among air traffic control computers in many parts of the United States. This forced air traffic controllers to manually type flight plans as they could not be automatically transferred between computers in different geographical regions. Hundreds of flights were canceled and thousands of passengers forced to reschedule their departures [205].

Supercomputers composed of thousands of computational nodes are already subjected to daily interruptions due to hardware failures [77]. Experts warn that future supercomputers containing millions of processing cores will experience hardware failures every minute or even every few seconds [182, 212]. Large super computers currently rely on programmers to handle hardware errors through application-level checkpoints. Unfortunately, such a solution cannot tolerate the fault rates expected at future semiconductor technologies, as future mean time between failures could easily be lower than the time necessary to checkpoint and restore the system, and operating costs may soon be dominated by diagnostic, testing, and replacing faulty components.

Hardware failures do not only concern high-end computers comprised of thousands of computational elements: as transistor size decreases with technology evolution, expected fault rates are projected to increase dramatically, potentially threatening any digital system [184]. For instance, physical aging is already a threat to current designs: in 2011 an Intel chipset model experienced a widespread transistor wearout problem in the field: premature transistor wear out in a SATA controller caused an increasing number of communication errors and eventually prevented the chipset from working. This problem triggered a worldwide recall in early 2011, with estimated costs to the company of up to one billion US dollars [170].

Silent Data Corruptions

While errors that reduce system availability may lead to severe consequences, even more worrying are faults that corrupt computations without providing any warning signs. Hardware failures causing silent alterations to program state or software output are particularly worrisome, as they might trigger unwanted actions that could damage valuable assets or even have safety impacts.

One of the most well-known examples of dangerous circumstances caused by silent data corruptions occurred during the Cold War in Omaha, Nebraska in 1980. The display of the command post of the Strategic Air Command showed that several Soviet ballistic missiles were headed towards the United States. In response to this alarm, B-52 bombers were ordered to start their engines and US warships were alerted to prepare to respond to an impending nuclear attack. The fact that no other military agency signaled any immediate threat provided American military authorities with evidence that no enemy attack was under way: it was a false alarm. In later analysis, technicians tracked down the root of the incident to a hardware failure in an electronic integrated component that was part of the communication system logic [63].

In modern commercial computers, users and applications trust the underlying hardware to correctly execute all instructions. Therefore, software developers rarely question hardware infallibility, even for widely adopted, sensitive applications such as security routines. However, the effects of unexpected events such as hardware faults on these applications can be dramatic. In the remainder of this section we demonstrate with a practical case study that silent data corruptions (SDCs) can have dramatic impacts on computer users [145].

2.2.2 The Dangers of SDCs: Faulty RSA Authentication

This section details a study we performed to highlight the dangers of not addressing runtime computer errors. Specifically, we demonstrate how undetected hardware failures can be used to perpetrate security attacks on a real microprocessor system executing an unmodified version of the widely used OpenSSL libraries. This attack model is not uncommon since many embedded systems, for cost reasons, are not protected against environmental manipulations that could trigger hardware failures.

The focus of our attack is the OpenSSL implementation of the commonly adopted RSA authentication algorithm [155]. Since it was introduced in 1977, RSA has been widely used for establishing secure communication channels and for authenticating the identity of service providers over insecure communication mediums. In this authentication scheme, the server implements *public key authentication* with clients by signing a unique message from the client with its private key, thus creating what is called a *digital signature*. The signature is then returned to the client, which verifies its authenticity using the server's known public key – as shown in Figure 2.1.a. In this scenario, an attacker does not need access to the internal components of the victim's chip, as it can simply collect the corrupted signature sent out by the server while subjecting it to transient faults. Once a sufficient number of corrupted messages has been collected, the attacker can perform offline analysis to extract the private key.

Hardware Fault Model

The fault-based attack developed here exploits hardware faults injected at the server side of a public key authentication (see Figure 2.1.b). Specifically, we assume that an attacker can occasionally inject faults that affect the result of a multiplication computed during the execution of the fixed-window exponentiation algorithm. Consequently, we assume that the system is subjected to a battery of infrequent short-duration transient faults, that is, faults whose duration lasts for a single clock cycle or less, so that they may impact at most one multiplication during the entire execution of the exponentiation algorithm. Moreover, our attack only considers hardware faults that produce a multiplication result differing from the correct one by a single bit. To make this attack possible, we collect several pairs of messages m and their corrupted signatures $\hat{s}$, where $\hat{s}$ has been subjected to only one transient fault with the characteristics described.

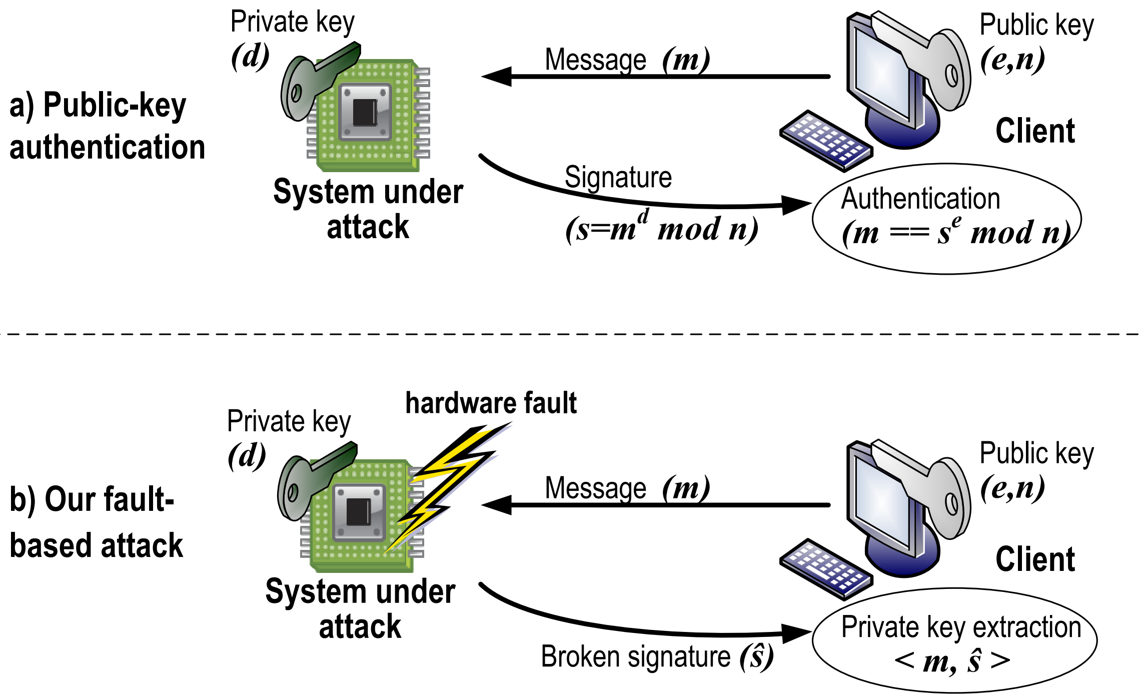


Figure 2.1 Overview of public key authentication and our fault-based attack. a) in public key authentication, a client sends a unique message m to a server, which signs it with its private key d. Upon receiving the digital signature s, the client can authenticate the identity of the server using the public key (n,e) to verify that s will produce the original message m. b) Our fault-based attack can extract a server's private key by injecting faults in the server's hardware, which produces intermittent computational errors during the authentication of a message. We then use our extraction algorithm to compute the private key d from several unique messages m and their corresponding erroneous signatures $\hat{s}$.

Evaluation

We devised our attack on a complete computer system mapped on a field programmable gate array (FPGA) device. The FPGA was configured with a complete, unmodified, Leon3 system-on-a-chip, representative of an off-the-shelf embedded device that could be the target of the attack presented above. The Leon SPARC CPU mapped to our FPGA was running at a frequency of 40 MHz and used a nominal supply voltage of 1.5 V.

The critical path on the integer pipeline of the Leon3 processor used in our tests is the carry signal of the multiplier. Thus, by manipulating the voltage of the power source of the FPGA device, we force the multiplier circuitry to miss timing requirements and sporadically corrupt its results. The higher the difference between the nominal voltage and the applied voltage, the higher the probability that the product of an input configuration will be wrongly computed. However, if the voltage is too high, the resulting fault rate will be too low, and it will take a long time to gather a sufficient number of faulty digital signatures. If the voltage is too low, the fault rate is excessive, causing system instability and multiple errors for each FWE algorithm invocation, thus yielding no private key information. Figure 2.2 shows the injected fault rate as a function of the supply voltage.

We analyzed the behavior of the hardware system computing the functions used in the OpenSSL library while being subjected to supply voltage manipulation. In particular, we studied the behavior of the routines computing the multiplication using 10,000 randomly generated operand pairs of 1,024 bits in length. As expected, the number of faults grows exponentially with decreasing voltage. The graph in Figure 2.2 also shows the fraction of FWE erroneous computations that fit in our fault model: those can leak portions of the private RSA key of the server. Note that, with decreasing voltage, eventually the fraction of single fault events begins to decrease as the FWE algorithm experiences multiple faults more frequently. We empirically established that the ideal voltage is the one at which the rate of single bit fault injections is maximized, for our setup it was 1.25V. The error rate introduced at that voltage is consistent with the computational characteristics of FWE, which requires 1,261 multiplications to compute the modular exponentiation of a 1,024-bit key. Thus, the attacker should target a multiplication fault rate of about 1 in 1,261 multiplications (0.079%). Using this particular voltage during the signature routine we found that 88% of all FWE invocations led to a corrupt signature of the type we were seeking.

The offline analysis and the private key extraction algorithms were executed on an 81-machine cluster of 2.4 GHz Intel Pentium4-based systems, and such a computer cluster was able to recover the private key of the attacked system in 104 hours.

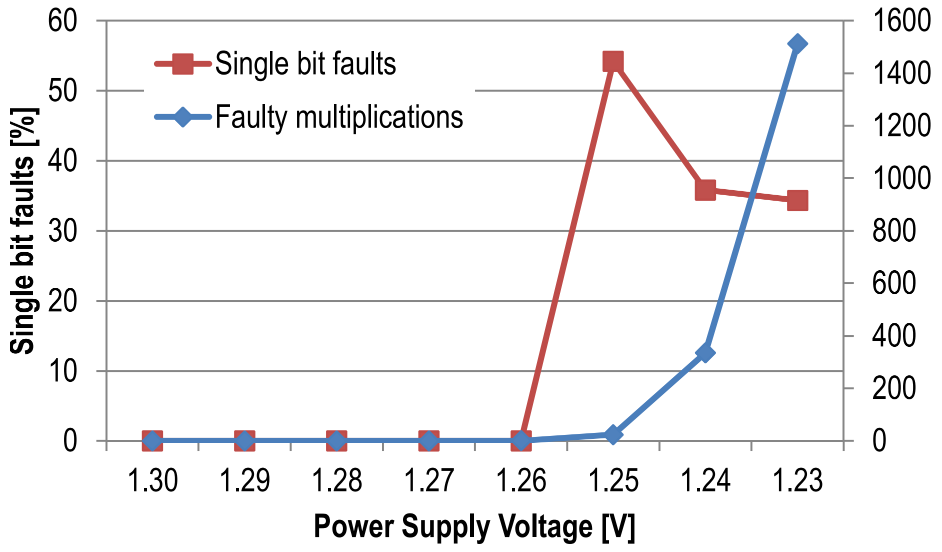


Figure 2.2 Sensitivity of multiplications executed in OpenSSL to voltage manipulations. The graph plots the behavior of the system under attack computing a set of 10,000 multiplications with randomly selected input operands at different supply voltages. The number of faults increases exponentially as the voltage drops. The graph also reports the percentage of erroneous products that leads to only a single-bit flip.

2.2.3 Reliability

In order to overcome the threats imposed by the decreasing transistor robustness, hardware systems need to consider reliability as a principal design requirement. Traditionally, high reliability is mandatory only for a limited number of applications, for which cost is not a major constraint. However, since we expect future semiconductor technologies to significantly increase failure rate for all digital systems, new architectural solutions must ensure correct computing on unreliable physical substrates [21, 62].

2.3 Management of Specialized Functional Units

Recently, computer architectures have undergone a number of radical changes. While in the past designers focused on improving the performance of general-purpose computing machines, modern computer chips tend to include an increasing number of diverse and specialized hardware components.

As the number of transistors available in a computer chip continues to increase, computer architects first shifted their focus from single microprocessors to multicore and many-core systems, prioritizing core count over single-threaded performance. As this trend continues, we can predict that SoC with hundreds or thousands of processing elements will be prominent in the market in the imminent future [22, 34]. Unfortunately, the demise of Dennard scaling caused a stagnation in the trends of both operating frequency and supply voltage. Nevertheless, future systems will soon be able to integrate tens of billions of transistors and enable single chips to include an astonishing number of hardware modules. This is particularly problematic, since it simply does not make sense to add more general-purpose processors to a chip if they cannot be powered up. The future time when most silicon available in a chip will have to be shut off due to power constraints is called "the dark silicon era" [42].

Researchers have recently started to investigate new designs that could improve processor performance while also reducing power consumption. For this purpose, several solutions propose to shift design emphasis from high-performance general-purpose processors to specialized hardware components that target particular tasks, thus improving energy efficiency and speeding up workload execution. In this computational paradigm, portions of the code that can be accelerated in hardware are offloaded and executed by dedicated modules. Modern SoCs include a number of specialized hardware modules, such as GPUs or FPGAs. Microprocessors can also be augmented to include specialized functional units such as fast multipliers, FPUs, and vector processing units. On one hand, external accelerators such as GPUs and FPGAs can be highly optimized and made accessible to all general-purpose processors in the system. On the other hand, the interface between these hardware accelerators and general purpose processors is burdened by the overhead required to establish a communication channel between them. As a result, programs that only sporadically need hardware accelerators cannot rely on these external modules because communication costs would quickly overshadow their benefits. Moreover, it has been shown that often software applications can be sped up by orders of magnitude on general-purpose processors simply by optimizing the execution of a limited number of operations [36, 209]. Therefore, in this thesis we focus on developing solutions that can enable general purpose microarchitectures to take advantage of these specialized functional units.

While general-purpose processors could deploy and leverage specialized functional units such as fast multipliers, FPUs, and vector processing units, both design and runtime limitations constrain the usage of these components in traditional CPU designs. Since current general-purpose cores have been conceived to operate in completely different technological environments, they can be extended to include only a few coprocessors. This characteristic severely limits a design capability to deploy and utilize the specialized functional units needed to overcome the dark silicon era. At runtime, the challenge lies in activating and managing the hardware components that can contribute to an application's execution, while disabling the ones that are no longer needed. Correct and prompt execution of these operations is extremely important. Indeed, while each of these specialized functional units can perform a subset of functions much more efficiently, it is very unlikely that a chip could power and cool down all of them simultaneously [52].

The intersection between all these requirements and constraints creates the need for a technology that can empower systems to include a large number of specialized functional units and efficiently manage them at runtime. In the remainder of this section we will briefly overview different solutions proposed to deploy specialized hardware accelerators, and introduce the desirable traits of a processor architecture that could succeed in the dark silicon era.

2.3.1 Specialized Hardware in the Dark Silicon Era

Hardware specialization can improve both energy and performance by orders of magnitude, opening a new horizon for future computer architectures. A plethora of hardware solutions based on FPGAs, SIMD, GPGPUs, and application-specific accelererators are already available to speed up software operations, routines, libraries, and even entire applications [198]. Some of these hardware accelerators, such as FPGAs and GPGPUs, are fully decoupled from the general-purpose processors. Others, such as SIMD units and specialized functional units, directly connect to the structure of a processor, and therefore require more attention. This thesis focuses exactly on developing processor architectures that could accommodate a large number of this latter type of accelerators. Beside the deployment of these hardware accelerators, these specialized functional units must also be managed at runtime, so as to let software applications benefit from their features.

Single Instruction Multiple Data Units

Single instruction, multiple data (SIMD) instructions allow software applications to directly exploit data level parallelism. SIMD instructions are provided to execute at high performance on SIMD units and are particularly beneficial for multimedia applications, and thus most modern CPU designs support them. An early example of a computer system that adopted these instructions is the PASM, a large-scale multimicroprocessor designed at Purdue University for image processing and pattern recognition [173]. It has been shown that the adoption of SIMD units in commercial processors for desktop and server computing improves performance by up to 60% at marginal area costs (10%) [152]. Processors in the mobile market also benefit from these instructions, as, for instance, MPEG-4 and H.264 video encoding and decoding accelerate by up to 60% [154].

Application-Specific Accelerators

Application-specific accelerators are modules designed to suit particular functions. Due to their high engineering costs, this type of specialized hardware is the most expensive but also the one that can provide the highest returns. Some applications require the development of complete computer systems for a single purpose. Anton, for instance, is a custom system designed to perform biomolecular simulations [168]. Designing such specialized computing systems from the ground up is very costly. Therefore, engineers often opt to design specialized functional units with the sole purpose of speeding up a subset of operations. Examples of such systems are: CryptoManiac, which focuses on accelerating cryptographic routines [209], and EFFEX, which is an architecture developed to speedup mobile computer vision algorithms [35]. A more general solution that targets popular commercial applications is GreenDroid, whose specialized logic can attain 10 to 1,000 times better energy efficiency compared to general-purpose processors [67]. Although these latter solutions propose complete, dedicated processors, their specialized functional units can be integrated into general-purpose CPUs. Indeed, most current commercial microprocessors already include some accelerators, which are accessible through dedicated instructions. Finally, beside the adoption of specialized functional units, recent research has proposed to augment a processor with a programmable memory subsystem to improve the performance of specific applications [36].

2.3.2 Hardware Adaptability

As mentioned above, systems solely based on general-purpose cores cannot fit the power constraints that characterize the dark silicon era [94]. Since the use of off-the-shelf processor cores is not a viable option, future designs must be easily extendable and adaptable to accommodate a large number of specialized functional units. Thankfully, the increasing component density available at future semiconductor technologies will enable computer chips to integrate these numerous specialized modules. While some of these accelerators, such as FPGAs and GPGPUs, can be easily decoupled from a general-purpose processors, others, such as SIMD and application-specific functional units, are tightly integrated. In order to enable future CPUs to deploy and leverage these specialized functional units within the short – and shrinking – time-to-market required, future designs should be adaptable.

An adaptable architecture should be able to dynamically match, activate, and assign these specialized resources to software workloads [89]. First, it should dynamically tune a number of hardware parameters (such as power consumption, operating frequency, and active features) to best match the available resources with application demands. Second, it should automatically respond to environmental and physical changes and self-configure to fit the various utilization scenarios [121]. Modern general-purpose processors already incorporate some degree of adaptability, and several of them include ancillary microcontrollers to monitor hardware utilization and power consumption, and even to ensure that the system is operating correctly. [44, 119].

Finally, adaptability will also greatly benefit fault-tolerant computing, since a hardware system could automatically isolate defective or imperfect components and utilize only the functional ones [183].

2.4 Lack of Design Modularity

Industrial state-of-the-art microprocessors are some of the most sophisticated artifacts known to modern engineering. In 2005, it was reported that the implementation of a new microarchitecture for the flagship Intel processor typically requires a maximum of more than 500 engineers across a wide range of tasks – architectural specifications, logic, circuit and physical design, validation and verification, design automation, *etc.*– for a timeframe of more than 2 years [18].

The challenges connected with correctly designing a processor are well-known and have been extensively analyzed and discussed by researchers both in academia and industry. Since components behaviors are affected by their local state, engineers cannot use a strict design-and-conquer approach and concentrate their efforts on testing the operations of parts of a design in isolation. Indeed, the combined state space of a modern processor is simply astronomical, hence, it is impossible to exhaustively analyze the behavior of a modern digital design [32].

Even design teams that adopt state-of-the-art techniques and CAD tools to improve their design process cannot possibly guarantee the correct functionality of their products, and thus machines with latent bugs are often shipped to customers [200, 202].

2.4.1 Modularity

One common trend in current design methodologies is that of integrating several hardware modules into a single chip to form a System-on-Chip (SoC). This trend is due to the increasing engineering costs and the shrinking time-to-market characterizing many modern digital systems. Design modularity has been the key to tackling both of these issues. For instance, the recent prospering of highly modular platforms, such as the ones provided by ARM, have shown that this design approach can enable small, agile design teams to release new products every few months. Indeed, modularity is needed to master complexity and to enable component reusability [8]. Moreover, designers consider modular systems more trustworthy than non-modular ones because it is easier to individually design and test their parts [88].

Still, while design modularity has been a major contributor to the development of SoCs [158], microprocessor designs are typically monolithic, hence they cannot benefit from this property. Modular microprocessor designs could increase their customizability while also reducing overall engineering costs. The adoption of a modular design methodology in the context of microprocessors would reduce scheduling and cost uncertainty.

2.5 Limits of Current Approaches

Current computer architectures fit technological scenarios that are completely different from the ones we foresee for the next 2 or 3 decades. Design choices that constituted key strengths in previous environments may quickly become weaknesses in the future. This section briefly compares three classic execution models: reconfigurable logic fabrics, control-flow machines, and data-flow architectures. The advantages and disadvantages of each of these execution models are discussed. Additionally, we analyze the reasons why none of them alone can provide the three proprieties we deem necessary for surviving the future technological challenges that we outlined in this chapter, that is, the increasing fragility of the manufactured components, managing diverse specialized functional units, and lack of design modularity.

2.5.1 Reconfigurable Logic

The struggle for an efficient reconfigurable hardware design has been long and arduous. Since the first introduction of programmable gate arrays in the late sixties [99], designers have proposed a plethora of configurable hardware solutions. Some of these systems target programmable accelerators [39, 112, 135, 136], while others envision full computer systems built of reconfigurable logic [38, 199]. Field Programmable Gate Arrays (FPGAs) are chips that can be programmed to perform any logic function and are the most successful reconfigurable architectures currently available. Such devices are composed of small configurable blocks, each of which includes both logic and memory elements. These chips are commonly deployed for rapid prototyping, application-specific accelerators, and even for small-scale productions [50, 139].

Even though FPGA devices are some of the most fault-tolerant, adaptive, and modular hardware devices available in the marketplace, they suffer from some major limitations. First, their efficiency is typically much lower than that of the other architectures, yielding to significantly lower performance/cost ratios [196, 204]. Second, FPGAs dramatically increase a project's complexity since mapping a design to fit within the given reconfigurable resources is typically a cumbersome and time-consuming process [113]. Third, area and energy footprints of generic designs mapped on FPGAs are significantly higher than those targeting custom silicon logic [95].

2.5.2 Control-flow Machines

The control-flow architecture, which is inspired by the work of early computer scientist John von Neumann, is the most commercially successful and widely adopted computational model. Instructions running on these machines are executed serially by the order established by a program counter. While the control-flow model does not suffer from any of the performance deficiencies of fully reconfigurable systems, monolithic central processing units are neither very adaptable nor modular.

With the target of augmenting these systems with both adaptability and reliability features, previous research has focused on developing flexible and robust control-flow architectures. These solutions can provide both component redundancy and fault containment. For instance, BulletProof targets reliability on processors implementing very long instruction words (VLIW), but this solution is hardly applicable to other designs [172]. Other works focus on maximizing the lifetime of components with natural redundancy, such as the reorder buffer, branch history table, caches, and other arrays of regular structures [26, 171]. StageNet is a more radical approach to reliability and adaptability. It proposes a reconfigurable control-flow architecture connecting multiple identical pipeline stages extracted from in-order cores [73]. Faulty pipeline stages can be individually disabled and the system can be reconfigured to force programs to operate only on the functioning hardware. Finally, multiprocessor designs enable naïve solutions to perform redundant executions and to isolate faulty processors. Unfortunately, all these approaches are only viable for systems affected by just a few hardware defects. Since each core relies on extensive centralized control logic, these solutions cannot sustain high failure rates [127].

All of these prior works build on the basic structure of typical control-flow machines and improve fault tolerance at a relatively low area and energy cost.

2.5.3 Data-flow Machines

Data-flow machines [7, 197] do not obey the serial execution model adopted by control-flow architectures. In fact, an instruction in this system executes as soon as all its inputs become available. Data-flow machines offer significant advantages over regular control-flow architectures. Since they are completely data-driven, their performance is only inhibited by true data dependencies. Furthermore, components in a data-flow system are stateless, and therefore free of side-effects. Such characteristics make this architecture very attractive for providing both adaptability and modularity. Although data-flow machines historically had limited commercial success, researchers have invested significant effort in developing and analyzing these architectures. For instance, Multiscalar [174], RAW [191], TRIPS [27], and Wavescalar [187] are all examples of recent works inspired by this execution model.

Unfortunately, the shortcomings of data-flow architectures typically overshadow their advantages. First, they often rely on dedicated compilers to extract operand and data dependencies and map them into the hardware. Such compilers are effective only on a very narrow set of applications, and legacy code cannot take advantage of the vast amount of logic available in data-flow machines. Second, there are several technical obstacles in the way of designing efficient data-flow machines due to the need to dispatch, distribute, and match tokens for both instructions and operands.

2.6 Summary

In summary, as manufacturing capabilities continue to improve, a transition in the properties of the semiconductor technology scenario is quickly approaching. Therefore, future computer systems will face a series of new challenges [23]. This chapter analyzed in detail the most significant problems jeopardizing the advancement of future digital systems: the increasing fragility of transistors [21], the challenges in managing specialized functional units [198], and the lack of design modularity [25]. Classic computer architectures such as fully reconfigurable fabrics, control flow machines, and dataflow architectures were designed to fit completely different technological scenarios, thus their performance will suffer greatly when faced with these new challenges. In this context, it is necessary to explore novel computer designs that could tackle all three of these challenges and operate successfully in the environment we expect at future semiconductor technologies.

Chapter 3
Reliability

The first hurdle that computer architectures must overcome is the increasing fragility of devices built with future semiconductor technologies. As transistor dimensions continue to shrink, the susceptibility of transistor devices to failures is expected to increase significantly [21, 49, 61, 184]. Emerging technologies promise to deliver massive integration of highly unreliable nanodevices [185], thus an environment where digital systems are affected by an extremely large number of hardware failures is plausible. Therefore, the solutions we develop in this thesis must provide low-cost mechanisms that can effectively prevent failure. In order to define the requirements for our design, it is imperative to study the effects of hardware failures on current designs. The insights from these studies drive the design choices of the reliability mechanisms developed for current and future architectures.

3.1 Chapter Organization

This chapter begins with a detailed analysis of the issues related to the increasing fragility of computer designs. In order to guide this research towards a solution that could tolerate a large number of failures, this work first analyzes the effects of errors on modern computer systems. Performing accurate fault injection campaigns is a tedious and time-consuming task. Three factors burden reliability studies of modern designs: the length of the simulations, the size and the complexity of current systems, and the high detail necessary to accurately model transistor faults. Due to the impracticality and imprecision of prior fault analysis tools, a new framework that is able to perform fast accurate reliability analyses, called CrashTest, is introduced in Section 3.2. CrashTest is publicly available on the Internet at the address:
`http://eecs.umich.edu/crashtest/`
The methodology developed for this work inspired and enabled a large body of research on effective fault analysis and several scientists have already adopted this framework [60, 68, 87, 130, 148].

Section 3.3 then presents the results obtained using CrashTest on a modern, industrial-grade microprocessor, the Sun OpenSPARC T1 [186]. This analysis also considers and evaluates a state-of-the-art hardware fault detection solution based on monitoring software malfunctions [53, 107, 203]. The results of this study demonstrate that modern industrial-strength microprocessors cannot cope with runtime permanent faults. Furthermore, these experiments show that even state-of-the-art research solutions can only partially mitigate this problem. In fact, a small but significant number of hardware failures can escape these mechanisms and silently corrupt a system's outputs.

Finally, Section 3.4 uses the insights from the experiments performed with CrashTest to develop a novel methodology to diagnosing hardware failures at runtime. A large body of previous research proposed a reliable computing paradigm that executes periodic hardware tests on an operating computer system. Microprocessors employing this model periodically suspend their operation to perform diagnostic tests on their hardware in order to expose any eventual physical defect [41, 110, 120, 172]. In the adaptable distributed architecture developed throughout this thesis, we use a similar execution paradigm to provide system reliability. In particular, we design a novel testing technique based on the observation that hardware utilization varies greatly among applications and even among different phases of one application. Our solution, called application-aware testing, proposes to adapt diagnostic routines to the actual hardware utilization. As shown in the experimental results, this design delivers high fault-coverage at very low performance overhead and near-zero area cost. These features enable autonomous detection and diagnosis of faults, enabling low-cost reliable computations on unreliable semiconductor substrates.

3.2 Analyzing Reliability Issues Through CrashTest

A detailed analysis of both transient and permanent faults can yield new insights about the behavior of faulty computers. While transient faults have been the focus of extensive previous research, scientists found major technical limitations in the study of runtime permanent faults. Transient faults have been a concern for a long time as they have been known to severely affect systems deployed in spacecrafts and avionics [48, 65, 124, 153, 157]. Hence, over the years, engineers developed a solid body of methodologies and tools to analyze the effects of these upsets – ranging from software simulators [43, 141] to the irradiation of physical devices with high-energy particles [11, 131].

On the contrary, there is very little research on the effects of permanent failures on active hardware. Nevertheless, as the probability of these faults occurring on live machines increases, engineers must also start considering them. Unfortunately, a fault injection campaign targeting this kind of failures is very tedious and time-consuming. Three factors burden such a study. First, the model of the hardware design under evaluation must be very detailed because faults manifesting at the transistor level can propagate across all layers of a computer system – circuitry, microarchitecture, architecture, and software. Second, the number of possible fault locations is extremely vast since hardware failures could manifest in any part of a design and they could taint large portions of it. Third, accurate fault injections require the observation of a design for millions, or even billions, of clock cycles as a fault's effects might be delayed or masked for a long period of time [105, 146].

3.2.1 Requirements for a Reliability Analysis Framework

The process of accurately assessing the robustness of a bare unprotected design, or evaluating the effectiveness of candidate fault-tolerant techniques, places the following requirements on a resiliency analysis infrastructure:

- **Low-level Fault Analysis –** High fidelity is a very important aspect of a resiliency analysis framework. Using high-level models of micro-architectural components with limited knowledge of the underlying circuit is inadequate to perform high-fidelity resiliency analysis. In order to correctly model the introduction, propagation, and possible masking of the faults, the resiliency analysis framework must accurately gauge circuit-level phenomena using a detailed low-level model of the design under analysis (*e.g.*, gate-level netlist).

- **Flexible Fault Modeling –** Due to the existence of multiple silicon reliability threats, a resiliency analysis framework needs to support an extensive collection of low-level fault models to cover silicon failure mechanisms that range from transient faults, to manufacturing faults, process variation induced faults, and silicon wearout related faults. Moreover, silicon fault modeling is an open area of research with continuous advancements [28, 69]. Often, new fault models are devised targeting emerging silicon failure modes or more accurately modeling existing failure mechanisms. Therefore, it is crucial for an analysis framework to be easily upgradeable with new fault models.

- **Fast Design Simulation –** The simulator must deliver sufficient performance to enable the analysis of complex systems, including booting an operating system and run applications. This will allow users to assess the impact of faults at the full-system level, and still deliver evaluations with a fast turnaround.

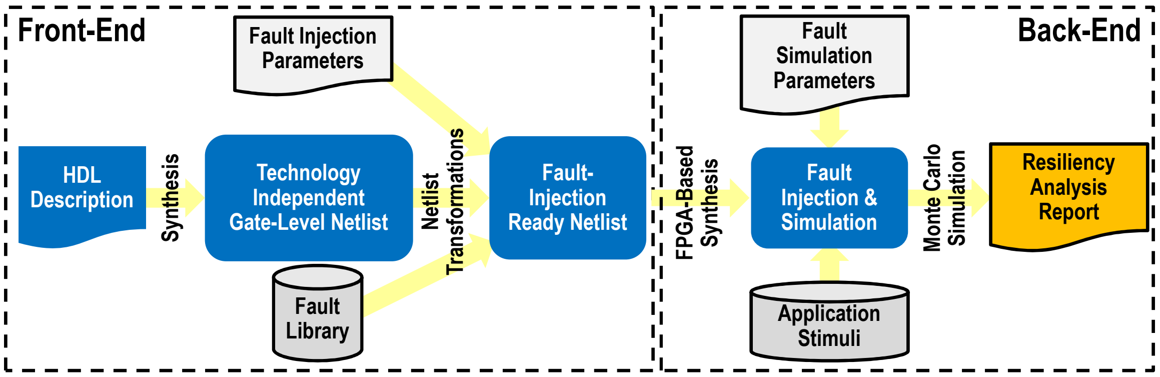

Front-End
Fault Injection Parameters
HDL Description
Synthesis
Technology Independent Gate-Level Netlist
Netlist Transformations
Fault-Injection Ready Netlist
Fault Library
FPGA-Based Synthesis
Fault Simulation Parameters
Back-End
Fault Injection & Simulation
Monte Carlo Simulation
Resiliency Analysis Report
Application Stimuli

Figure 3.1 Overview of the resiliency analysis framework: CrashTest is composed of (i) a front-end stage generating the fault injection-ready gate-level netlist and (ii) a back-end stage performing fault injection and analysis and generating the final resiliency analysis report.

- **Flexible Simulation Interface –** It is critical for the usability of a framework to provide an intuitive way to analyze a wide range of designs and fault-tolerant techniques. Thus, a resiliency analysis framework demands a flexible interface and proper stubs to accommodate the evaluation of different systems.

Given this challenging set of requirements for resiliency analysis, this thesis develops a FPGA-based infrastructure that can perform fault injection campaigns on gate-level models. This framework, called CrashTest, achieves both the accuracy and the performance needed to assess the reliability of modern computer systems.

3.2.2 CrashTest Overview

CrashTest's goal is to provide a fast, high-fidelity and comprehensive analysis of the effects of a broad range of fault classes on the applications running on a design under analysis (this could be either an unprotected design or a fault-tolerant design). Given the specification of the design under analysis in a hardware description language (HDL), CrashTest automatically orchestrates a fault injection/analysis campaign. This process is composed of two stages:

1. a front-end translation that generates the fault-injection ready gate-level netlist from the design;
2. a back-end fault simulation and analysis that performs the actual fault injection and fault monitoring, and evaluates the effects of the injected faults.

Figure 3.1 shows a graphical schematic of CrashTest.

Only a FPGA solution can provide sufficient performance to run software applications on large and complex designs. However, a major cost in adopting this solution is the overhead necessary to map a design on the FPGA fabric. For complex designs, the time required to generate a netlist mappable on the FPGA device can be prohibitively long. To reduce this overhead, CrashTest inserts multiple faults in a same mapped netlist. Each fault can be dynamically activated via software, thus amortizing the synthesis cost over several analyses.

Framework Front-End – First, the HDL model of the design under analysis is synthesized using a standard cell library to obtain a gate-level netlist. CrashTest does not require the use of any particular library, as long as the chosen library can be properly modified to support the fault models.

For each standard cell in the target library (*i.e.*, a combinational gate or a sequential element), CrashTest provides a gate-level logic transformation that modifies the cell and inserts extra logic to control the fault location. This extra logic can be activated at runtime to emulate the effects of a fault injected into the cell. CrashTest comes with a wide range of fault models and gate-level logic transformations to provide the capability of emulating different failure mechanisms. The collection of all logic transformations is stored in the framework's fault library. Based on the injection parameters selected by the user (*i.e.*, the fault models and the injection locations), the framework automatically generates the fault injection-ready netlist by selecting the transformed cells in the library. This netlist is then transferred to the fault analysis simulator.

Framework Back-End – The fault injection-ready netlist is then re-synthesized to target the FPGA device. After this step, the fault injection and analysis campaign is ready to begin. Based on the fault simulation parameters given by the user, the fault injection/analysis emulator injects faults at different sites in the netlist and monitors their propagation and impact on the design and the running applications. During fault emulation, the design under analysis is exercised with the application stimuli. To gain statistical confidence on the provided results, the experiments are repeated in a Monte Carlo simulation model by altering the fault sites and/or the application stimuli. After running a sufficient number of experiments to gain statistical confidence, the results are aggregated into the resiliency analysis report, which is the final deliverable of the CrashTest framework. The following subsections describe each step of the CrashTest framework in more detail.

3.2.3 Gate-Level Fault Injection Methodology

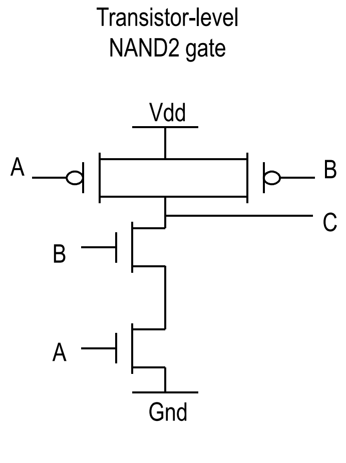
Technology Independent Logic Synthesis – The first step in the front-end stage of the CrashTest framework is to convert the user-provided high-level HDL model of the design

Stuck-at: The stuck-at fault model is the industry standard model for circuit testing. It assumes that the defect behaves as a node stuck at logical 0 or 1. The stuck-at fault model is most commonly used to mimic permanent manufacturing or wearout-related silicon defects.
Stuck-open: The stuck-open fault model assumes that a single physical line in the circuit is broken. The unconnected node is not tied to either Vcc or Gnd and its behavior is rather unpredictable (logical 0 or 1 or high impedance). This model is commonly used to mimic permanent defects that are not covered by the stuck-at fault model.
Bridge: The bridge fault model assumes that two nodes of a circuit are shorted together. The behavior of the two shorted nodes depends on the values and strength of their driving nodes. This model covers a large fraction of permanent manufacturing or wearout-related defects.
Path-delay: The path-delay fault model assumes that the logic function of the circuit is correct, but that the total delay in a path from inputs to outputs exceeds the allocated threshold and causes incorrect behavior. This model is used to mimic the effects of process variation or device degradation due to age-related wearout.
Single Event Upset: The single event upset (SEU) fault model assumes that the value of a node in the circuit if flipped for one cycle. After this one-cycle upset, the node returns to its normal behavior. The SEU fault model is used to mimic transient faults commonly caused by cosmic radiation or alpha particles.

Table 3.1 Fault models. CrashTest is enhanced with an extensive collection of fault models. These fault models cover transient faults as single event upsets and also a variety of permanent hard faults related to manufacturing, wearout, and process variation silicon defects.

under analysis into a common format that the framework can analyze to obtain an accurate list of candidate circuit locations for fault injection. This is achieved through logic synthesis with Synopsys Design Compiler targeting a technology-independent standard cell library (GTECH). The resulting gate-level netlist is composed of simple logic gates (*e.g.*, AND, OR, NOT, Flip-Flops, *etc.*,) and it is free from any fabrication technology-related characteristics. This netlist is subsequently parsed to generate a list of all possible injection locations, that is, a list of all logic gates and flip-flops in the design. This list can be used to specify the injection locations; alternatively, if randomized fault injection is desired, a random selection of fault sites can be performed by the framework.

Netlist Fault Injection Instrumentation – After fault locations selection, the gate-level netlist is instrumented with extra fault injection logic that, when enabled, emulates the effects of the injected faults. Each fault model supported by the framework is associated with a specific gate-level logic transformation to achieve this goal. The collection of gate-

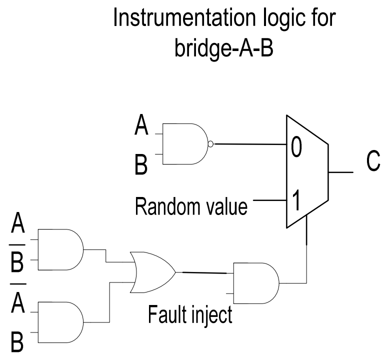


a)

b)

Fault symptom table

A	0	1	0	1
B	0	0	1	1
Fault-free	1	1	1	0
Bridge-A-B	X	X	X	0
Bridge-A-C	1	X	X	X
Bridge-A-n1	1	1	1	X
Bridge-B-n1	X	1	1	X
Bridge-B-C	X	1	1	X
Bridge-C-n1	1	X	X	0



c)

Figure 3.2 Logic transformations - Bridge fault: The CMOS transistor-level design of a gate in (a) is used to generate the gate's fault symptom table for the fault model shown in (b). Part (c) shows the instrumentation logic for emulating the effects of the Bridge-A-B fault.

level logic transformations comprises the framework's fault library. This modular design makes upgrading the framework with new fault models a straightforward task, by simply implementing new logic transformations into the fault library. This resiliency analysis framework is already equipped with a large collection of fault models and their corresponding netlist logic transformations. This collection spans an extensive spectrum of silicon failure mechanisms, ranging from transient faults due to cosmic rays, to permanent faults due to silicon wearout. Table 3.1 shows a list of supported fault models along with a brief description.

Gate-Level Logic Transformations – Some fault models require trivial gate-level logic transformations. For instance, the instrumentation needed to emulate a stuck-at fault is just a multiplexer that controls the output of the faulty gate and has one of its inputs connected to logic zero/one. However, there are more complex fault models that affect the design at the transistor level. For example, the bridge fault model assumes that two circuit nodes in the design are shorted together. To emulate the effect of a bridge model with high fidelity, we simulated the faulty CMOS gates at the transistor level and generated a corresponding fault symptom table. To illustrate this process with an example, Figure 3.2(a) shows the CMOS transistor level representation of a NAND2 logic gate, while Figure 3.2(b) shows the corresponding fault symptom table.

The table shows that the fault's effects are masked for some input combinations, thus the faulty gate behaves as a fault-free gate. However, for other input combinations the fault's effects propagate to the gate's output and result is an unstable output signal (Random Value in Figure 3.2(c)). The framework's fault library is populated with a fault symptom table for each combination of standard cell library gate and supported fault model. Given a gate type and a fault model, the instrumentation engine accesses the fault library and applies the corresponding logic transformation to instrument the fault. Figure 3.2(c) shows the instrumentation logic for a bridge fault between circuit nodes A and B of the NAND2 gate. A fault-tolerant design should be capable of handling these faults and either mask the errors introduced or reconfigure itself to exclude the faulty part of the design.

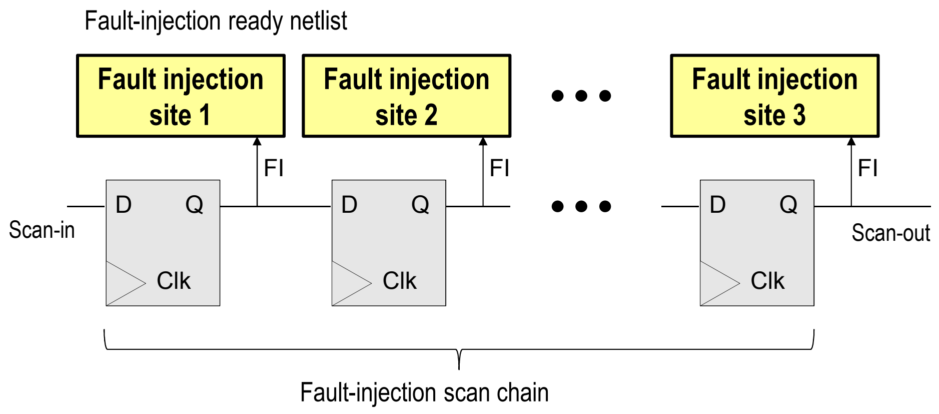


Figure 3.3 Fault injection scan chain. The netlist is instrumented with fault injection logic for multiple faults. The scan chain controls the fault injection during emulation.

Path-Delay Fault Model – The logic transformations required by most fault models are similar to the one presented in Figure 3.2(c) for the bridge fault. One exception is the path-delay fault model, which has slightly different characteristics. Path-delay faults are characterized by slower combinational logic gates causing longer delays than foreseen at design time. Whenever these slower gates are exercised, they may cause timing violations (*i.e.*, flip-flops at the end of the path miss to latch the newly computed value). CrashTest emulates the effects of the path-delay fault model through a gate-level logic transformation such as that in Figure 3.4. To determinate the set of flip-flops affected by the slower faulty gate, we trace forward through the combinational logic and find all those flip-flops that have a path including the faulty gate. From that set of flip-flops we choose only those that have a path delay with a timing slack smaller than a predefined threshold specified by the user (*i.e.*, the expected delay due to the faulty gate).

Fault Injection Scan Chain – To avoid re-instrumenting the netlist each time a new fault must be injected, the netlist can be instrumented for several faults at multiple locations. This accelerates the emulation at the back-end of the framework, but also increases the instrumented circuit size. Moreover, the insertion of each fault into the netlist adds an extra control signal required for enabling and disabling it at runtime (for instance, signal Fault Inject and Random Value in 3.2(c)). As shown in Figure 3.3, these control signals are latched and connected to a scan chain, which is accessible at runtime by the Fault Injection Manager (see Figure 3.5).

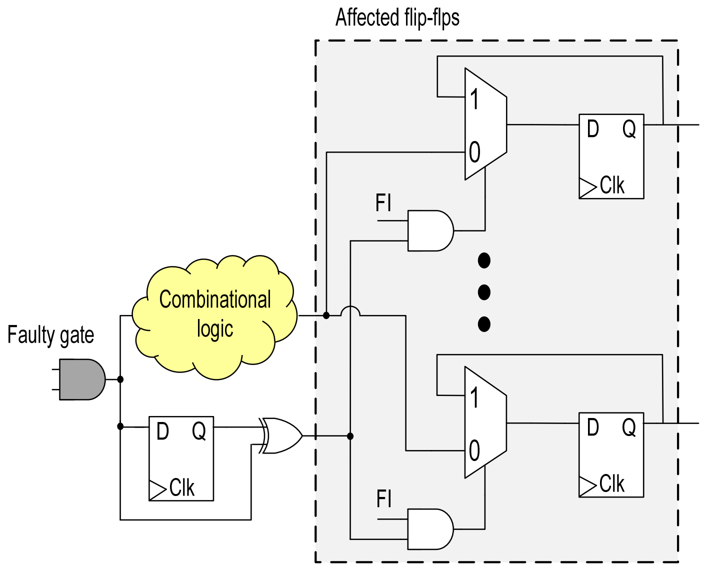


Figure 3.4 Logic transformation for the path-delay fault model. If the output of the faulty gate changes in a given cycle, all affected flip-flops miss latching the newly computed value and hold the previous cycle's value.

3.2.4 FPGA-Based Fault Emulation

CrashTest employs an FPGA platform to emulate the fault injected hardware and accelerate the fault simulation and analysis process. The first step in this process is to synthesize and map the fault injection-ready netlist to the target FPGA. To provide a standard simulation interface that is independent of the design under analysis, CrashTest automatically generates an interface wrapper to the fault injected-ready netlist. This interface wrapper provides a seamless connection with the fault injection manager, an automatically-generated software program responsible for orchestrating the fault injection and analysis campaign. The interface wrapper and the fault injection manager are connected through an on-chip interconnect bus. Figure 3.5 shows the major components and the data-flow of the fault injection, simulation and analysis process.

A small general purposes processor embedded in the FPGA runs the fault injection manager. Some FPGA devices already include a fully-featured cpu. Alternatively, a general-purpose processor can be mapped on the FPGA itself (*e.g.*, Microblaze [211]). The fault injection manager is responsible for several tasks:

1. Feed the instrumented injection scan-chain with all the control signals required in the fault injection campaign. This is done through a FIFO queue updated each time a new

fault is activated into the design. The fault injection parameters (*i.e.*, fault location and time) are stored on an off-chip memory accessible by the injection manager.

2. Stimulate the design through the input registers. The applications stimulus is either provided by the user or automatically generated, and it is stored in the off-chip memory.

3. Monitor the output of the FPGA-mapped design for errors through the output registers. The output is compared to a golden output that is collected with a fault-free version of the same design and it is stored in the off-chip memory.

4. Maintain fault analysis statistics and store the results to the off-chip memory for later processing.

5. Synchronize the FPGA-mapped design with the fault injection process through the interrupt counter.

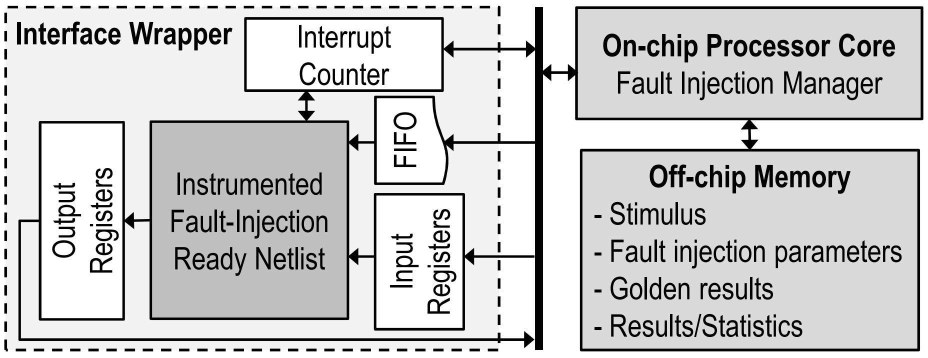


Figure 3.5 FPGA-Based fault injection and simulation. The FPGA-mapped netlist is wrapped by a standard interface providing a seamless connection to the fault injection manager that is running on an on-chip processor core.

3.3 CrashTest'ing the OpenSPARC T1

This section presents the results of an analysis performed through CrashTest on an industrial microprocessor: the OpenSPARC T1 [186]. This study reports the effects of two permanent fault models (stuck-at and path-delay) on the logic of the microprocessor, indicating whether the fault was either masked, caused software disruptions, or silently corrupted program output. Note that several researchers proposed to leverage software anomalies to detect permanent hardware failures [106], and therefore this study directly evaluates the capability of such mechanisms to be adopted as near-zero cost fault detectors.

3.3.1 Fault Injection Methodology

The fault injection procedure is illustrated in Figure 3.6. For these experiments, fault locations are activated in two application execution points: the first one is immediately after benchmark initialization, while the second is approximately halfway through its execution. A restore operation is performed after each fault injection. After each restore, the fault injection manager allows five seconds for the system to rewarm the caches and populate its TLB entries. Experiments which do not trigger any fault detector must be executed to completion to determine if the fault corrupted their outputs.

```
 1. Configure FPGA with n fault sites enabled
 2. Download OS and modified firmware
 3. Boot system
 4. Start benchmark
 5. Run benchmark until test point
 5. Checkpoint system
 6. For each j of the n faults:
 7.   Warm up Caches and TLBs
 8.   Enable fault location j
 9.   Wait terminating condition or timeout
10.    If software anomalies detects the fault
11.      Report detection latency
12.    If application successfully terminates
13.      Analyze output files to detect SDC
14.   Disable fault location j
15.   Restore to checkpoint
```

Figure 3.6 Fault injection procedure on the FPGA. This routine activates and analyzes the effects of each fault location individually. In order to speed up the fault injection campaign, system state is checkpointed after the benchmark is initialized and restored after the effects of each fault are analyzed.

Table 3.2 Modules of the OpenSPARC injected with faults.

OpenSPARC T1 unit	Gate count	FF count	Stuck-at faults	Path-delay faults
Arithmetic Logic - ALU	1,968	65	19	9
Divide - DIV	3,277	486	31	65
Error Corr. and Ctl. - ECC	998	237	10	32
Execution Control - ECL	1,727	335	17	45
Float. Point FE - FFU	5,776	836	55	112
Instruction Fetch - IFU	13,980	3,775	225	511
Load Store - LSU	24,127	4,397	635	594
Multiplier - MUL	14,665	647	138	87
Reg. Management - RML	1,206	231	11	31
Reg. Bypass Logic - BYP	5,938	708	56	95
Trap Logic - TLU	18,693	3,737	334	502

For these experiments, stuck-at and path-delay faults were injected in various nets in the design. The core was partitioned into multiple modules and a number of faults proportional to their area were injected in random locations. Table 3.2 lists the units into which faults were injected, the total number of gates in each unit and the number of different fault locations that were introduced within each unit. The targeted number of faults injected in each module is a function of its area (approximated by the number of gates in the module's gate-level netlist) and was computed for a confidence level of 95% and a confidence interval of 4% (Table 3.2). For three hardware units– instruction fetch, load-store, and trap logic– the ratio between the number of faults and number of gates is higher than for the other modules. These three units, instrumented with the checkpoint mechanism and faults, could not meet the timing requirements on the FPGA. Thus, we partitioned them into even smaller submodules that were instrumented separately. For each of these three modules, the total gate count for the submodules is higher than for the original unit since the synthesizer has a narrower optimization scope. Note, however, that the increased ratio only increases the confidence of our results for the experiments performed on the IFU, LSU and TLU. No faults were injected in the memory array structures of the design (such as register file, caches, and TLBs) since faults in these units can be easily modeled through microarchitectural simulations. Faults, however, were injected in the control logic of these large memory elements.

The timing achievable for the fault-enabled OpenSPARC core on our FPGA device was 100ns, enough to run the design at a frequency of 10MHz. Even though this frequency is four times slower than the one reachable by the original design on the same FPGA device, it still yields a six orders of magnitude speed-up compared to a software simulations with equivalent fault accuracy.

The effects of stuck-at and path-delay fault models were studied on five applications from the SPECInt 2000 benchmark suite with a combination of the test and reduced input sets [101] shown in Table 3.3. This table reports execution times for the FPGA system running at 10MHz. The SPECInt 2000 benchmarks were chosen to better compare these results with previous evaluations on architectural simulators already present in the literature. For each fault that is not detected, the application must run until completion to determine if the fault was masked or caused a silent data corruption (SDC). Since these experiments consisted in testing the effects of more of 50,000 faults, running the reference input set for such benchmarks was not a practical option due to the extremely lengthy runtime (about 5 years of FPGA-time for all considered applications). Therefore benchmarks were executed with either "test" or "medium" input sets reduced through the techniques presented in [101].

CrashTest allowed us to study the behavior of 30,800 stuck-at and 20,830 path-delay faults across all modules of the OpenSPARC T1 processor core design. Figure 3.7 shows the outcome of the stuck-at-0, stuck-at-1 and path-delay experiments.

Table 3.3 Benchmarks evaluated in our experiments on the OpenSPARC T1. benchmarks were executed with either "test" or "medium" input sets reduced through the techniques presented in [101].

Benchmarks	Input set	Instructions	FPGA time
175.vpr (place)	medium reduced	458M	9m 9s
181.mcf	test	419M	5m 27s
197.parser	medium reduced	913M	6m 16s
255.vortex	medium reduced	547M	11m 55s
300.twolf	test	415M	5m 15s

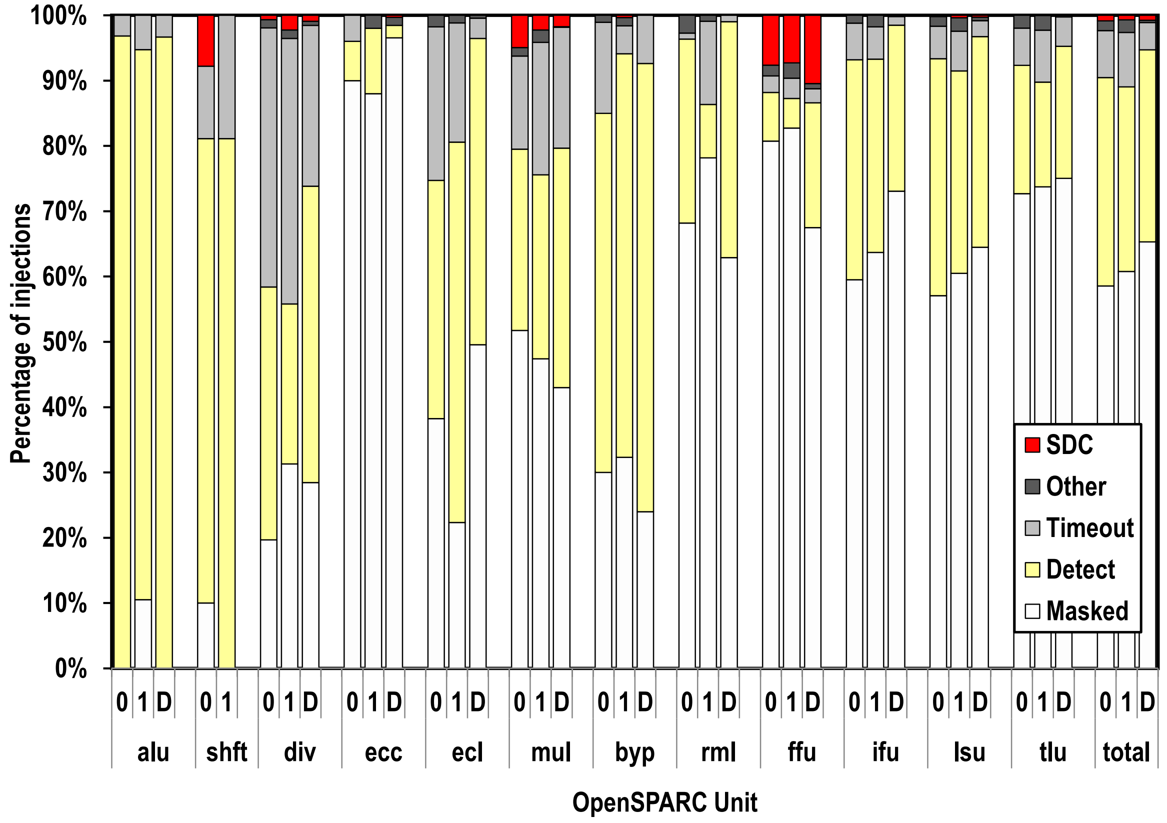


Figure 3.7 Breakdown of experiments for stuck-at and path-delay faults. The X-axis indicates the OpenSPARC modules studied, while the Y-axis reports the percentage of experimental outcomes. For each module, we report fault injections outcomes for stuck-at-0 (0), stuck-at-1 (1), and path-delay (D).

3.3.2 Fault Injection Results

For each fault injection, we monitored the behavior of the system, and categorized any possible experiment outcome in five mutually exclusive categories, that will later be discussed in detail later.

- **Detection:** 29.83% of the experiments triggered at least one of the software anomalies detectors considered: 1) Fatal Traps 2) Kernel Panics 3) Hypervisor Crashes 4) Firmware Checks 5) Hardware Stalls 6) Application Abnormal Exits.

- **Masked:** overall, 61.93% of the faults were masked, meaning that the applications finished without a detection and their output matched the correct outputs.

- **Silent Data Corruption (SDC):** only 0.76% of the experiments were not detected by the symptom-based detectors and silently corrupted application's output. Our definition of SDC is somehow conservative as many of the outcomes differ in ways that are not important to the user. Additionally, some of the fault outputs in this category clearly showed erroneous behavior (i.e., the fault is detectable by the user but may not be recoverable). Future work will focus on further analysis of these instances.

- **Timeout:** to limit the experiment time, we declared timeout if the benchmark ran for longer than 150% of the time than it does on a fault-free system (path-delay experiments had this time period extended to 200%). A total of 6.40% of fault injections caused the benchmarks to execute above these time thresholds. We expect many of these cases to be detectable by a hang detector.

- **Other:** these cases constituted only about 1.10% of the fault injections. Given the large number of experiments and system complexity we were pleasantly surprised to have such a low fraction of uncategorized system behaviors. Due to their rarity, these experiments do not significantly alter the overall results.

Masking

A high masking rate of 59.7% for stuck-at and 65.3% for path-delay was observed. These results are higher than previously observed in microarchitecture-level permanent fault injections [107] (16%) and the gate-level results for the three modules simulated with previous techniques (30% to 40%). Several factors can contribute to this discrepancy.

First, the OpenSPARC core was originally designed to support four hardware thread contexts. However, limited FPGA resources constrained us to only test the one threaded version of the core. Unfortunately it is difficult (if not impossible) to prune out all the logic relating to the other threads at the RTL level. This causes unused hardware logic to be left in the synthesized design and such hardware may be chosen as a fault injection site, thus increasing the overall masking rate. Second, some modules, such as the multiplier, the floating point frontend, and the trap logic unit, contain circuitry that is not exercised frequently, if at all, by the applications used. For example, the benchmarks do not contain streaming, MAC, or floating point instructions that rely on the advanced features provided by such hardware units. Similarly, the trap logic unit performs very little exception handling, since the SPEC benchmarks evaluated require very little I/O and OS interaction. Furthermore, CrashTest models design aspects not previously considered (such as gate-level characteristics of the design), thus adding extra layers of masking that could prevent injected faults from affecting application outputs. The last difference is that fault injection campaigns performed by previous works only focused on a portion of the design, not testing three large units – instruction fetch, load-store unit, and trap handling logic – which all have a masking rate above 55% for both stuck-at and path-delay experiments.

The ALU is a particularly interesting case, since it reports a much lower masking rate than the one observed for stuck-at and path-delay faults in previous fault analysis campaigns - up to 40% in the result reported with by Li, *et al.* [105]. This difference can be explained by the fact that OpenSPARC uses the ALU for both address generation and normal arithmetic integer operations, whereas that previous work modeled an out-of-order core with separate address generation and arithmetic/logic units.

The overall detection rate is 30.1% and 29.4% for stuck-at and path-delay fault models, respectively. In Table 3.4, we show the percentage of detections that each detector is responsible for.

We found that a large portion of hardware stalls, roughly 54%, were due to hardware faults injected in control logic in the load-store unit. Also, 38.7% of detections are Kernel Panics due to faults in the data path submodules of the load-store unit. Overall, the types of detections occurring in the OpenSPARC platform are of a larger variety than the ones reported in previous evaluations of software anomalies detectors, due to unexpected application or OpenSolaris services failure.

Table 3.4 Symptom-based fault detector breakdown.

Fault type	Kernel panics	Fatal traps	Firmware checks	Hypervisor crashes	Abnormal exits	Hardware stalls
Stuck-at	31.5%	25.7%	10.8%	9.9%	5.8%	16.2%
Path-delay	44.7%	20.4%	4.7%	3.6%	9.0%	17.6%

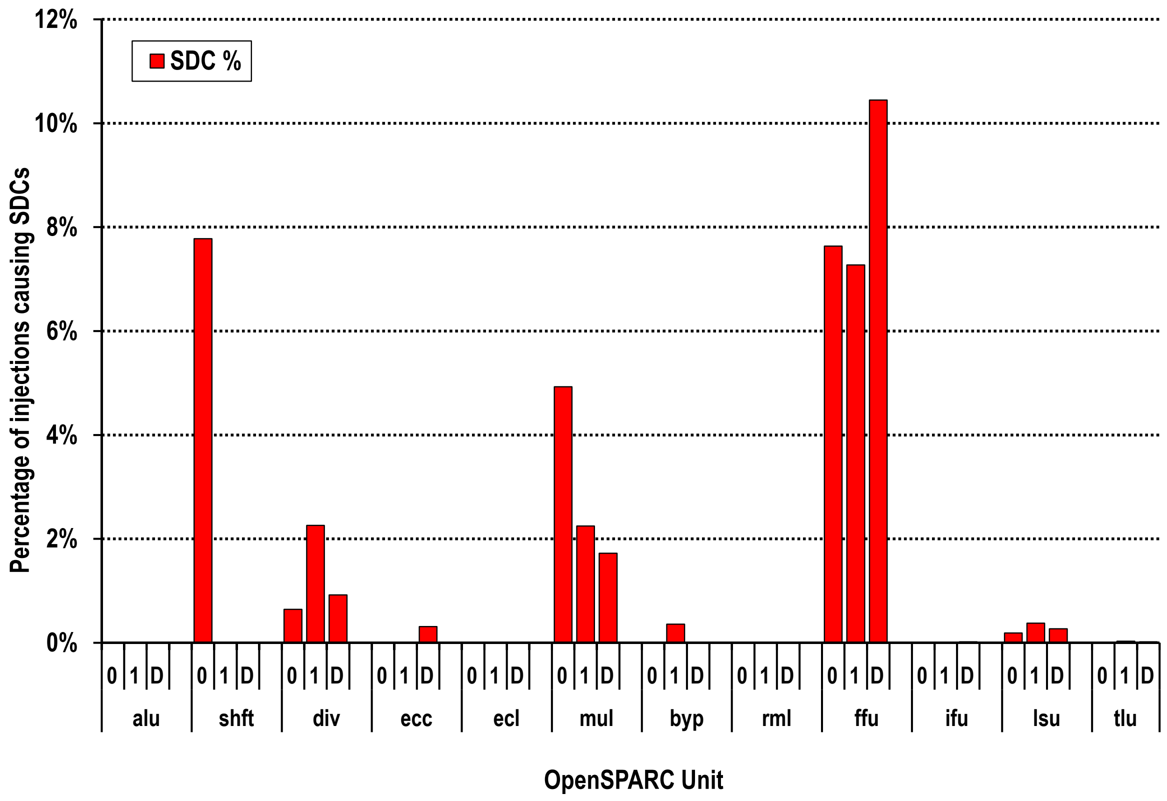


Figure 3.8 Breakdown of SDCs per unit for both stuck-at and path-delay faults. The X-axis shows the OpenSPARC unit under study, and the Y-axis shows the percentage of fault injections which resulted in SDCs. For each module, we report fault injections outcomes for stuck-at-0 (0), stuck-at-1 (1), and path-delay (D).

Timeouts

About 6.4% of the fault injections experiments hit an established simulation time limit (7.8% and 4.2% of the experiments for stuck-at and path-delay, respectively).

Other Anomalous Outcomes

About 1.1% of the fault injection experiments were placed in the "other" category (1.6% of stuck-at experiments and 0.4% of path-delay experiments). A small fraction of these cases were due to erroneous software behavior that caused the file system to become unusable or full. In a real system, the file system free space will typically be much larger than on our experimental FPGA platform, so it is unclear how these situations would translate in a real system. Additionally, there are a number of cases where the framework could not interact with the standard output of the application for more than three minutes.

Silent Data Corruptions

These experiments yielded an overall silent data corruption (SDC) rate of 0.76% for stuck-at faults, and a rate of 0.75% for path-delay faults. Figure 3.8 show the number SDCs for each application, broken down by the units generating the SDC.

Interestingly, all but four units produced none to very few SDCs (0 SDCs for ALU, ECL, and RML; under 0.4% SDC rate for BYP, ECC, IFU, LSU, and TLU). SHFT, DIV, and MUL had higher SDC rates, but under 5%. The FFU had the highest SDC rate at 7.45% for stuck-at faults and 10.47% for path-delay faults. Thus, the vast majority of the SDCs are concentrated in units that are used in computing data values for the program (as opposed to addresses or control related operations) – improving reliability in these units is the focus of the next chapters.

Overall, the SDCs from these experiments resulted in corruptions that ranged from subtle to drastic and obviously wrong. Subtle SDCs had the effect of adding or subtracting a line of text or changing the value of a field by a small amount. Other more drastic corruptions would cause non-ASCII characters to be output, or integer fields in the output to be changed to very large values. With a proper metric for application-level fault tolerance, as described in [109], it is possible that some of these corrupted outputs might be acceptable for the target application.

3.3.3 Detection latency

Figure 3.9 shows the detection latencies for the stuck-at and path-delay experiments. These results report the number of instructions committed between when a fault occurs and when it is detected. Any detections with unknown latency (e.g., application-level abnormal exit) were included in the $>$100M category for completeness.

Our measurements show that functional units like DIV, MUL, and FFU generally have long detection latencies, which might simply indicate that a fault activation depends on the execution of specific and relatively infrequent instructions. For instance, we observed the OpenSolaris service manager daemon fail due to a violation of the legal value range of a floating point variable. Other units such as the TLU may have longer detection latencies, since fault activation could be related to rare events, such as interrupt or exception handling. Overall, these results prove that software symptoms can indeed be effective in detecting hardware faults. The large fraction of detections occurring with latencies below 100K instructions also shows that low-overhead hardware recovery solutions are practical to recover from hardware faults [150, 176]. Nevertheless, further studies are required to understand

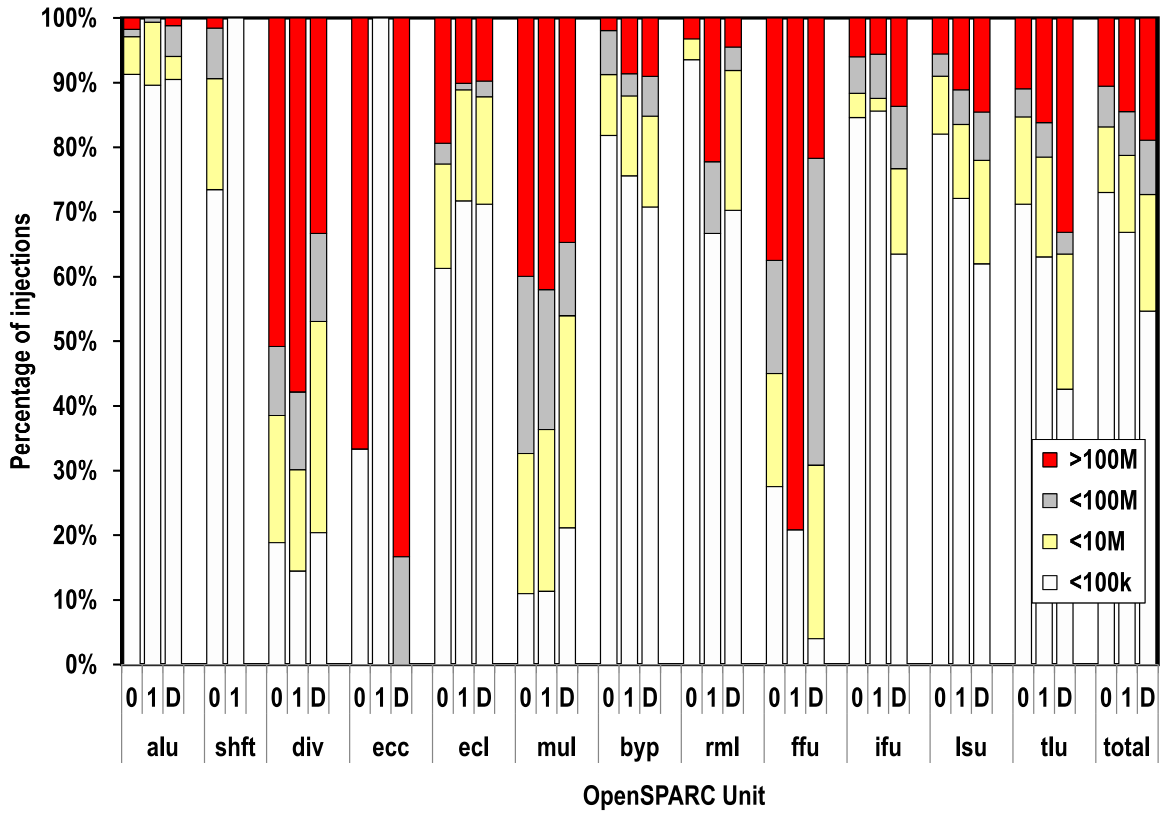


Figure 3.9 Breakdown of the detection latencies for stuck-at and path-delay fault experiments. The X-axis reports the OpenSPARC module under study, and the Y-axis shows the percentage of experiments with that detection latency measured in the number of retired instructions after a fault was enabled. For each module, we report fault injections outcomes for stuck-at-0 (0), stuck-at-1 (1), and path-delay (D).

how the fault injections that manifest with much higher latencies impact the checkpointing and recovery overheads.

3.4 Adaptive Hardware Reliability

As discussed in Chapter 2, the capability to handle failures and hardware alterations at runtime is a major requirement for future computer architectures. The dense device integration responsible for increasing fault rates can also enable architectural solutions to improve system reliability. Indeed, faulty cores in modern multiprocessors can be simply disabled without compromising the availability of the other components [167]. Furthermore, current mission-critical systems often adopt expensive fault-tolerant techniques, such as triple modular redundancy, to detect and correct hardware errors [12]. Unfortunately, these solutions are only viable on systems subjected to a limited number of faults, since each fault may significantly impact the throughput of a system. The results presented in the previous section empirically demonstrate that modern industrial-strength microprocessors alone cannot cope with runtime permanent faults. Such experiments also show that low-cost fault detectors based on software malfunctions can detect a significant portion of non-masked hardware failures.

Unfortunately, a small but significant number of hardware failures can escape these protection mechanisms and silently corrupt a system's outputs (about 1% of our fault injections, on average). Such faults are particularly dangerous, as undetected hardware malfunctions can damage valuable assets [5] or even endanger human beings [63]. While a plethora of near-zero cost solutions have been proposed to detect and correct hardware errors that cause patent erroneous behaviors [53, 81, 107, 203], none of them can handle the more subtle faults that result in silent data corruptions (SDC). Interestingly, the results presented in Section 3.3 report that failures causing this dangerous behavior are concentrated in a handful of hardware components. In fact, the vast majority of SDCs appear in components used only for computing data values for a program. The data we collected show that fault injections performed on the shifter, the multiplier or the divider of the OpenSPARC T1 report higher SDC rates than all the other units combined (between 2 and 8%). Finally, fault injections in the hardware that executes the floating point instructions report an amount of SDCs above 10%.

Several solutions have been proposed to detect and overcome these insidious hardware faults. A large body of previous research proposed a reliable computing paradigm that partitions a program execution into short computational intervals, called "epochs" [41, 110, 120, 172, 177, 210]. Microprocessors employing this model periodically suspend their operations to perform diagnostic tests on their hardware in order to expose any eventual physical defect. These machines do not trust their own results until the integrity of their underlying hardware is confirmed. Once all of the periodic hardware tests terminate successfully, the operations executed in the previous epoch are allowed to commit to memory or to update the state of I/O peripherals.

Periodic testing techniques are known to achieve high fault coverage – up to 100% – at a reasonable performance cost. These online tests can be accomplished through the addition of ad-hoc hardware testing components [120, 172] and/or through the execution of high-quality software test sequences [41, 110]. These periodic tests are effective at diagnosing hardware failures, however the approach is time-consuming, resulting in sensible system slowdowns of up to 30% [41]. Independently from their implementation, all previously proposed techniques focused on maximizing the fault coverage of the entire silicon system.

However, the results obtained in our analysis provide interesting insights into how software utilizes a processor's hardware components. These experiments report that a very high number of faults are masked, especially when they are injected in hardware modules that applications do not exercise frequently. Examples of these hardware modules include the multiply-accumulate unit, the floating point unit, and the circuitry that handles traps and exceptions. Indeed, the SPEC CPU2000 benchmarks do not execute any of the specialized instructions that rely on the advanced features provided by these hardware units. Similarly, the trap logic unit is rarely triggered by these benchmarks since they require very little I/O and OS interaction.

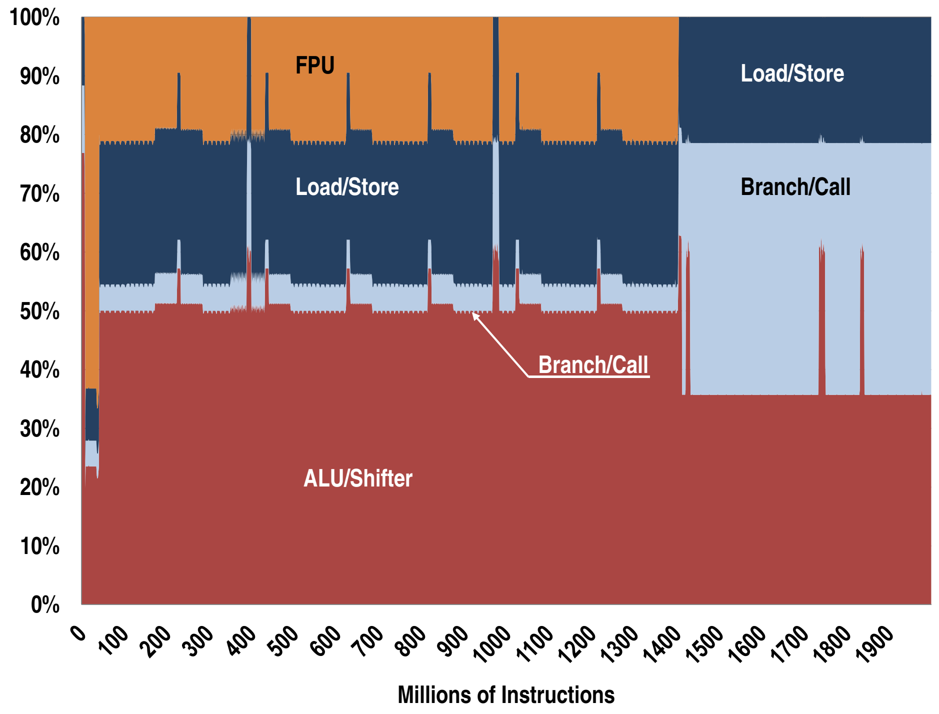


Figure 3.10 Dynamic instructions in the Nas FT benchmark. This application solves differential equations using Fourier transforms. The figure shows the type of dynamic instructions executed over a window of 2 billion instructions and their distribution by type.

The research performed in this thesis revealed that hardware usage of different functional units by a software application varies greatly during execution. Consequently, the fault locations that might corrupt the state or the outputs of an application tend to vary over time. For example, Figure 3.10 shows the type of dynamic instructions executed over a window of 2 billion instructions by a scientific benchmark application, Fourier Transform from the Nas suite [54]. Note that the execution of this workload is characterized by long phases executing instructions which only require few processor units. These patterns of hardware utilization are not unique to this benchmark, but are common to most applications.

Accounting for this important characteristic of a hardware/software system can lead to a more efficient and accurate solution to online testing of processors, which can emphasize application sensitivity to runtime hardware faults. The remainder of this chapter introduces a new metric which measures the quality of runtime hardware tests and a novel methodology to performing periodic sanity checks on microprocessors. The new application-aware technique proposed here delivers high coverage at very low performance overhead and near-zero area cost. These characteristics make our online testing solution extremely valuable to providing reliability to the adaptive distributed architecture we target in this research.

3.4.1 Application-Aware Coverage

Traditionally, the quality of online tests has been measured by the fraction of transistors in the entire system for which a defect would be detected by a given test. Therefore a new metric, called *Application-Aware Fault Coverage* - A^2FC, is necessary to evaluate fault coverage in the context of an application's dynamic behavior. A^2FC measures the quality of a test with respect to its ability to detect a fault in hardware units that the application has exercised. For instance, if an application only uses the integer pipeline of a processor, a test that detects faults occurring exclusively in the floating point unit (FPU) would provide an A^2FC of 0%. If an application were to use the FPU during half of its execution cycles, a test that exclusively provides 80% coverage over the FPU unit's transistors would have an A^2FC of 40%. In other words, instead of measuring the fraction of transistors that a test covers, A^2FC measures the likelihood of detecting a hardware fault that might have corrupted some computations.

Below we first analyze the relevance of a failure manifesting at the transistor level, the hardware unit level and the chip level. We then consider the area fault coverage provided by a given test to define our A^2FC metric. Let P_f represent the probability of a single transistor permanently failing when it switches. Then the probability of that transistor not failing is $1-P_f$. If the number of switching events between two testing intervals is s, the probability of a transistor not failing after s switching events is $(1-P_f)^s$, and the probability that the same transistor fails within s switching events is $1-(1-P_f)^s$. Consider a hardware unit comprising n transistors, all subject to the same switching activity. Assuming that transistor failures are independent events, then the probability of at least one error occurring in a unit after s toggles is $1-(1-P_f)^{sn}$. Using Taylor binomial expansion, this expression becomes:

$$snP_f + \binom{sn}{2} P_f^2 - \cdots$$

If P_f is negligible when compared to s and n, the expression above can be approximated with snP_f. This is the case in all practical situations because the probability of a transistor failure due to a single switching event is extremely low. For example, if a 5GHz processor composed of 10 billion transistors had a Mean Time Between Failures of one day, and a hardware test is triggered every second, the value $1/P_f$ is 5 orders of magnitude greater than the product sn. As a conservative assumption, we consider that all transistors switch as often as the one that switches the most.

At the chip level, a processor is composed of i units. Assuming a negligible probability of having two faults manifesting in the processor within the time frame delimited by two subsequent tests, the probability of a chip incurring a fault is given by the sum of the individual probabilities that any of its modules incurred a fault: $P(chip\_fails) \approx \sum_i s_i n_i P_f$

Finally, one must take into account the ability of a test to detect these faults. Assuming that a test covers a fraction c_i of the transistors in each module i, the probability that it can detect a fault is: $\sum_i (s_i n_i c_i P_f)$. Therefore *Application-Aware Fault Coverage* is defined to be the ratio between faults that can be detected by the test and all possible occurring faults. Since P_f is common to all terms, it can be removed from the expression, yielding:

$$A^2FC = \frac{\sum_i (s_i n_i c_i)}{\sum_i (s_i n_i)}$$

Note that A^2FC takes into account usage of hardware modules due to a particular workload. For example, we can assume that a simple processor consists of only two modules that occupy the same area on the chip: an integer pipeline and a floating point unit. Then, we can consider two tests: one achieves 90% fault coverage over the entire processor area, while the other provides 95% coverage for the integer pipeline and 65% for the floating point unit. When we compare these two tests in terms of area fault coverage, the latter provides significantly lower coverage (80%) than the former. However, if an application does not utilize the floating point unit, an user attains much better protection from the second test. Taking dynamic behavior into account, our A^2FC metric would report a 90% coverage for the first test, against a 95% for the second. The remainder of this section uses A^2FC as a metric to evaluate how effective a test is in protecting a system against failures that can affect the correctness of software application.

3.4.2 Application-Aware Diagnosis

The diagnosis framework here proposed takes advantage of the dynamic program behavior observed to reduce the overhead required for periodic testing without affecting A^2FC. Classic online testing technologies invest significant effort to thoroughly test all components of a processor. In contrast, we propose to constantly monitor the activity of all functional units of the CPU and test only those contributing to the outcome of the user's application. With reference to Figure 3.10, note how the FPU is used steadily in the first part of the benchmark's execution, while the last portion only exercises the integer pipeline. Similarly, a test that optimizes performance without affecting A^2FC would invest time to check the FPU unit during the first part of the benchmark's execution, but would only focus on the integer pipeline in the last portion.

Application utilization of the underlying hardware is assessed through hardware counters, called *activity monitors*. An activity monitor is associated with each functional unit in the processor. Every time an instruction exercises a particular functional unit, the corresponding counter is incremented. To characterize the dynamic behavior of an application, the activity monitors are reset at the beginning of each epoch. At the end of the epoch, this framework evaluates the monitor's counters to determine which hardware units should be tested. This detection mechanism adapts to the applications executed on the processor, so that unit-focused tests are triggered on demand. Specifically, during each testing phase, it executes test routines only on the functional units that were utilized by the software application during the last execution interval. Units which did not experience utilization, based on the information from the activity monitors, are not tested since they would not improve overall A^2FC. This approach is beneficial for two reasons. First, units that have been exercised by the application, and might have corrupted it if faulty, are closely monitored. Second, test length is reduced by skipping tests of unused components, thus improving user's experience.

To further boost fault coverage in hard-to-test units we increase design observability by adding dedicated *observation points*. Enhancing the system with observation points does not impact performance and requires very limited hardware additions. Since data from the observation points is collected only during the testing phase, they are transparent to software applications. In addition, the same hardware counters used as activity monitors, can be used during testing to collect data from the observation points and their final value is used to verify test success. Our experiments show that this integrated approach can expose the vast majority of microprocessor's faults and, in particular, those to which applications are most sensitive, without incurring the high cost of traditional testing mechanisms such as BIST and scan-chains.

As a case study, we analyzed the behavior of the application Nas FT using an epoch of 20 million cycles. During normal computation, every time an instruction exercised a functional unit, the corresponding activity monitor was incremented. At the end of the epoch, the activity monitor associated with the floating point unit and the divider reported an utilization of 5.9 million and 1 million instructions, respectively. We set a 10% utilization threshold for this case study and, consequently, our A^2Test triggered hardware tests for the integer pipeline and for the floating point unit, but not for the divider.

Software Tests

In the hardware testing community, it is recognized that software-based fault testing can be very effective to expose the majority of faults in a processor [151]. Several techniques have been proposed in the literature to build software test routines to this end [16, 31, 210]. This framework uses a similar approach, and relies on the software regression suite developed for the functional verification of the processor under study. This set of software tests was selected because they strive to check all hardware components and system's behaviors. From this suite, we want to select several test subsets, one for each hardware unit. Each subset should comprise those tests most effective in detecting faults for a given unit. We accomplish this goal by formulating an integer linear programming (ILP) problem, such that its solution provides the set of tests we are seeking. To start this process, we partition the processor into several functional units and create an ILP problem for each of them. For instance, for the processor considered in our experimental evaluation, we partitioned the design in five separate units: integer pipeline, divider, multiplier, floating point front end, and stream processing unit. Solutions for the ILP problems generated are computed only once at design time and used to select the routines that should test each module at runtime.

A fault coverage matrix is built from the outcome of the software tests. Coefficients in the matrix specify which fault locations are exposed by each test (Figure 3.11.a). From the fault coverage matrix, the constraints for the ILP problem are generated (Figure 3.11.b): a binary variable is associated with every test (t_i) and fault location (f_j). One inequality is also added for each possible fault location. The binary variable that represents a test i, t_i, is set to 1 if and only if the associated test is selected for execution. A variable associated with a fault location j, f_j, is greater than 0 only if the fault is exposed by at least one of the tests a)

Tests	Fault locations F_0	F_1	F_2	F_3	F_4	F_5	Cost
T_0	0	1	1	0	1	0	c_0
T_1	1	1	0	0	0	1	c_1
T_2	1	0	1	1	0	0	c_2
T_3	0	0	0	0	1	0	c_3
T_4	0	0	0	1	1	1	c_4

b)

$$t_i = \begin{cases} 1, \text{ if } T_i \ (0 \le i \le 4) \text{ is selected} \\ 0, \text{ otherwise} \end{cases}$$

$$f_j = \begin{cases} 1, \text{ if one of more tests exposes } F_j \ (0 \le j \le 5) \\ 0, \text{ otherwise} \end{cases}$$

Constraint inequalities:

$$t_1 + t_2 \ge f_0$$
$$t_0 + t_1 \ge f_1$$
$$t_0 + t_2 \ge f_2$$
$$t_2 + t_4 \ge f_3$$
$$t_0 + t_3 + t_4 \ge f_4$$
$$t_1 + t_4 \ge f_5$$

c)

	Additional constraint	**Goal**
Integer Pipeline	$\sum_i t_i c_i \le$ test time budget	$\text{Max}(\sum_j f_j)$
Module Directed	$\sum_j f_j \ge$ target fault coverage	$\text{Min}(\sum_i t_i c_i)$

Figure 3.11 Formulation of the ILP problems for test routines selection. a. Example of a fault coverage matrix: a non-zero coefficient at location (i, j) indicates that the i-th test exposes the j-th fault. The last column reports cost in execution cycles. **b.** Constraints derived from the fault coverage matrix. **c.** Additional constraints and goals for the two types of ILP problems.

that will be executed. Since the integer pipeline is active all the time while the system is operational, the corresponding test is selected in each testing session. As a result, this test has the most impact on test execution and, consequently, we set a hard constraint on its time budget. In contrast, the test time for all other units is less critical, since they are triggered only occasionally. For those, high coverage becomes the most relevant parameter. Below we present the specific aspects of both problem setups.

Integer pipeline test. For this test we add a hard constraint to the ILP problem instance so that the total execution time of the tests selected is below a preset threshold. This choice is driven by the frequent use of this test and its consequent high impact on overall performance. In addition, the objective function of this ILP instance is to maximize the number of distinct faults covered by the execution of the tests (Figure 3.11.c).

Module-directed tests. For the other functional units, the primary goal is to achieve high coverage. A specific modular test is developed for each complex module in the microprocessor not already covered by the integer pipeline test. The ILP problem setup is similar; however, we do not set a hard constraint on the test execution time. Instead, we add a constraint requiring that overall coverage is above a user-specified threshold (Figure 3.11.c).

The ILP problem for the integer pipeline in a complex processor such as the OpenSPARC T1 consists of more than 300,000 fault locations, over 850 tests, and occupies more than 2GB of memory when stored in a file system. A commercial ILP solver spends between 2 and 40 hours to find a solution to the problem and peaks at 30GB of memory usage. Note that the solution to each ILP problem must only be computed once, when the microprocessor is designed; thus, although these problems are resource-consuming, their complexity is well manageable, even for an advanced processor, such as the T1.

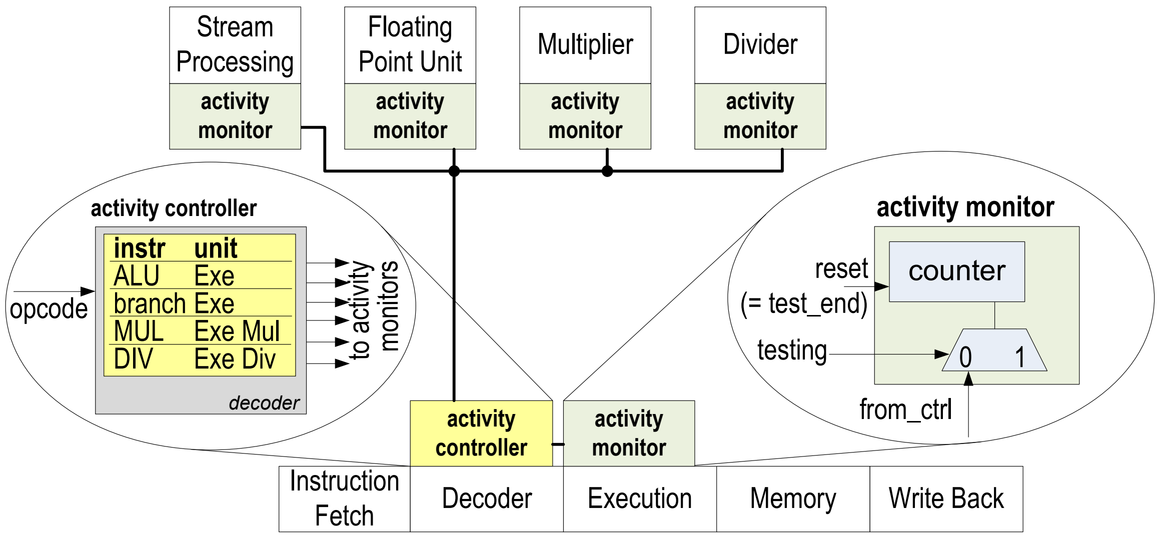


Figure 3.12 Activity monitors to track the use of each processor unit so that tests can be adapted to target those activated during the last execution interval. Monitors' counters are incremented by a controller based on the instruction flow and reset at the beginning of each execution interval.

Hardware Activity Monitors

We utilize activity monitors to track switching activity in the various units. These consist of counters associated with each complex unit in the microprocessor's architecture. Each activity monitor oversees a processor's functional module, and its counter is incremented every time the corresponding module is subject to switching activity. The counters are reset after each hardware integrity check (testing phase). In practice, module utilization can be approximated by analyzing the instruction flow: in this solution, we use a dedicated controller, which observes each instruction entering the processor's decoder stage and increments appropriate counters based on which units a given instruction exercises. The activity monitors are embedded in the processor's hardware, as shown in Figure 3.12. We envision that software routines evaluating the need of triggering a unit test can access counters' values. Functional unit testing can be triggered when a functional unit's utilization

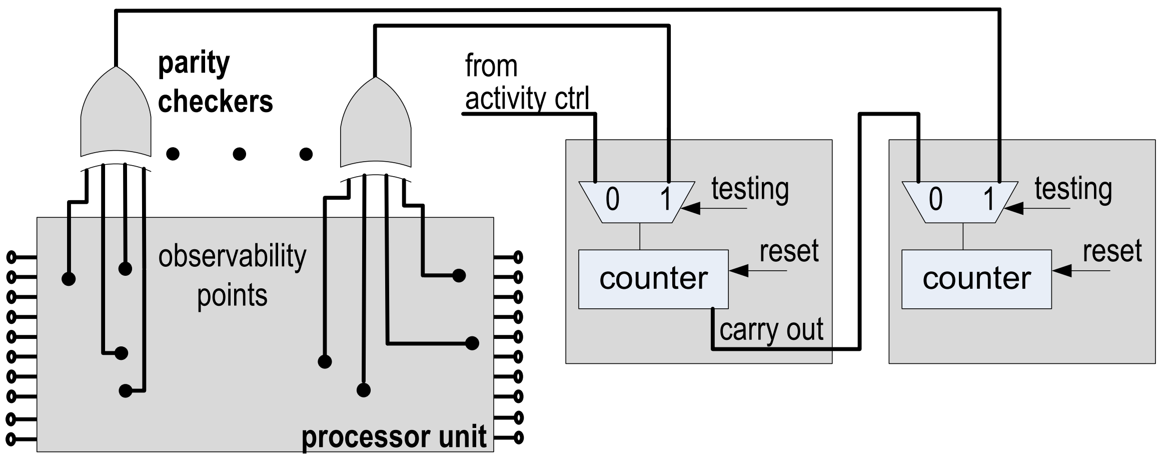


Figure 3.13 Observability extensions. Each processor's unit is augmented by a set of observability points, compressed through a parity checker and fed to local counters. Counters' values are then evaluated determine test correctness.

rises above a preset threshold. In our framework, we assume that users could configure the desired trigger thresholds dynamically, so to trade-off performance overhead with A^2FC.

Microprocessor Observability Extensions

To boost the coverage provided by the test routines we augment the processor's logic with observability points. Indeed, faults not detected during a test can be classified as either non-controllable or non-observable. Non-controllable faults lay in logic paths that are not exercised by testing. Usually they correspond to nodes that are stimulated only by rare events not controllable through deterministic software programs, such as external interrupts and error conditions. Non-observable faults correspond, instead, to internal nodes that toggle during the test, but whose eventual failure does not manifest in the test's outcome.

The tests selected can control the vast majority of fault locations in the design. By analyzing the non-observable nodes in the gate-level netlist of the processor, we found that these are often grouped in cones of logic. From the processor netlist, we built a graph connecting all non-observable locations in the design. We then identified the cones of logic rooted at each of these locations through a breadth-search-first algorithm. The non-observable nodes corresponding to cones containing at least four other non-observable locations were selected to be instrumented with an observation point. Through this selection, system's observability is extended for a very low hardware cost.

To reduce the amount of signals to monitor, we developed a simple compression circuit consisting of a parity detector. Several observation points are fed to the parity detector and its output is connected to a counter, so that each time the parity signal is asserted, the value of the counter is incremented. The value stored in the counter is reset at the beginning of the testing sequence. After the test completes, the counter is compared against a reference value and it is considered successful only if the difference between these two values is within an acceptable range (set at 10%). Counter value variations below the threshold are considered within the normal range for a complex processor executing a same test multiple times. Indeed our experiments show that the occurrence of a fault causes a significant difference in the counters values (greater than 10%). Figure 3.13 represents the schematic of our compression circuit. Note that the same counters can be used for both our compression circuitry and for the activity monitors.

3.4.3 Evaluation

We evaluated the quality of our solution on a Sun's OpenSPARC T1 processor [186] and compared against traditional non-adaptive testing solutions in terms of performance overhead, fault coverage, A^2FC and area impact. Table 3.6 provides a direct comparison of our evaluation with a number of previously published solutions. The processor implements the SPARC V9 ISA and supports 4-way fine grain multi-threading. The pipeline logic of the T1 was synthesized with Synposys Design Compiler targeting the Artisan IBM 130nm library. Fault coverage was obtained through fault simulation of functional vectors with Synposys TetraMAX. The Nas parallel benchmark suite was used to estimate the performance overhead on CPU-intensive programs [54]. In addition, we evaluated our solution on I/O intensive benchmark suites such as Bonnie [193] and Stream [118]. To estimate performance on a benchmark that relies on both CPU and I/O, SPECWeb was also considered [181]. Statistics on functional unit utilization were collected through Simics simulations. Performance was measured in number of committed instructions and impact of our design was evaluated against three epoch lengths: 20, 50, and 100 million cycles. Our experiments focus on stuck-at faults and do not account for faults either marked as undetectable by an automatic test pattern generator or within the design-for-test structures. Because all memory structures are protected with either parity bits or error-correcting codes, single permanent faults in memory are detected by mechanisms already present in the design [186]. Finally, the hardware additions necessary for our diagnosis system were developed in Verilog RTL and synthesized with the IBM Artisan 130nm library with Synopsys Design Compiler.

Table 3.5 Fault coverage achieved by integer pipeline tests. For each module in the OpenSPARC T1, the table reports the area occupied and fault coverage attained for test groups targeting the integer pipeline. The last two rows indicate total area coverage attainable and A^2FC for an application relying exclusively on the integer pipeline.

OpenSPARC T1 unit	**Area (%)**	**Test coverage (%)**					
		No limit	**5.0M cycles**	**2.5M cycles**	**1.25M cycles**	**0.5M cycles**	**1.25M w/ obs**
Instruction Fetch	7	94.4	93.8	93.2	88.9	82.8	
Execution	10	97.1	96.4	95.9	95.2	94.0	
Load Store	6	89.7	88.1	87.6	86.2	82.8	89.1
Trap Logic	10	88.7	86.0	85.5	84.3	78.7	87.1
Error Detection	1	33.6	33.5	29.6	27.7	26.5	
Multiplier	4	99.2	96.6	96.5	91.0	80.1	
Divider	4	98.7	98.7	95.5	95.5	91.3	
Stream Processor	3	93.7	89.1	84.8	79.9	60.5	
FP Front End	4	91.5	90.0	85.3	77.9	67.7	
Memory	51	100.0	100.0	100.0	100.0	100.0	
Total *(w/ Memory)*	100	96.3	95.5	94.9	93.6	91.0	94.1
A^2FC		96.6	95.9	95.7	94.8	93.2	95.5

Fault Coverage

We first determined the maximum fault coverage achievable when using Sun Microsystems' functional verification software routines. Because the overhead introduced by running all these programs sequentially is very high, we partitioned the processor into several functional units and grouped the test routines based on the functional units for which they provide high coverage. The functional units are listed in Table 1. We first focused on the processor's integer pipeline since its correctness is vital to nearly every instruction. The integer pipeline consists of four modules: instruction fetch, execution, load-store, and trap logic. As detailed in Section 3.4.2, an ILP solver was used to select offline the group of tests that yields the highest fault coverage within a given time budget. Table 3.5 shows the coverage attained for each module when running our integer pipeline test battery, over a range of execution budgets. The last two rows in the table report the total fault coverage attainable for the system, and the Application-Aware Fault Coverage for an application workload that relies exclusively on the integer pipeline.

The area-based fault coverage attainable by executing all tests in the integer pipeline group is extremely high, 96.3%. However, this comes at a very high cost: the test sequence requires nearly 26 million cycles. Thus we deemed necessary to select a subset of tests that would still lead to high fault coverage but within a limited time budget. As shown in Table 3.5, when the time budget is reduced, fault coverage for the integer pipeline modules is not effected as significantly as for other functional units. Among the T1, the load-store unit and the trap logic unit suffer of limited testability and, in an effort to increase their fault coverage, we enhanced them with 869 and 738 observation points, respectively. This led to a 3% improvement in the area-based fault coverage of these units, as indicated in the last column of Table 3.5.

Note that the coverage for the other functional units plummets as the time budget decreases, since the test group is focused only on the integer pipeline. For the other functional units, distinct test groups were selected by solving dedicated ILP problems: target fault coverage for the multiplier and the divider was set to 98%, and the cycles necessary to complete the corresponding test group are 27,383 and 290,715, respectively. For the floating point front end and for the stream processing unit, the target fault coverage was set to 96%, requiring a runtime of the 230,033 cycles for the first test group and 1,572,807 cycles for the second. These test groups target a very high fault coverage but are very time consuming. For instance, performing a thorough integrity check on the stream processing unit is extremely expensive, accounting for more than 1.5 million cycles. However, the unit is utilized infrequently, indeed none of the benchmarks made use of it. This observation further supports the hypothesis that, in order to maintain low performance overhead, tests should only be triggered for the hardware units that could have impacted computation results.

We then compared the degradation in fault coverage observed when running an application-oblivious test vs. an Application-Aware test, over a range of application programs. In this experiment the epoch considered was 100M instructions long and we used a threshold of 1 instruction to trigger unit tests. From our experiments, we report that the area-based fault coverage is reduced by 1.7% on average when going from an oblivious test to an adaptive one; correspondingly the A^2FC metric is only reduced by less than 0.1% on average.

Performance Overhead

To evaluate the performance overhead of our solution, we set a bound of 1.25 million cycles for the integer pipeline, since it provides a good compromise between area-based fault coverage and test runtime. For all the considered epoch lengths, the average of the performance overhead of our Application-Aware solution was more than 50% lower when compared against an online testing solution that is oblivious to application behavior.

The runtime overhead of our adaptive test system was also measured for several individual benchmarks against different test trigger thresholds. Figure 3.14, plots the runtime overhead of our proposed technique against an oblivious testing solution for some representative applications and for the average among all the considered applications. In particular, we compare the variation of A^2FC in our system against the A^2FC obtained by an oblivious solution. The epoch length for this experiment is set at 100 million instructions. Again, several test trigger thresholds to activate the module tests are considered (from 1 instruction to 20% of the instructions executed in the epoch). Note that for the benchmark Nas IS, the A^2FC achievable by our adaptive system saturates when the test trigger reaches 1% of the committed instructions. This behavior is common among several applications that only rely on few CPU functional units at a time. In addition, the runtime required to perform the proposed A^2Test on the OpenSPARC T1 is extremely small compared to the results obtained by techniques such as those in [41] and [110]. As expected, by using test adaptation, the performance overhead of online testing decreases when the threshold increases. Moreover, Application-Aware testing reduces the performance overhead by a factor of two over oblivious solutions.

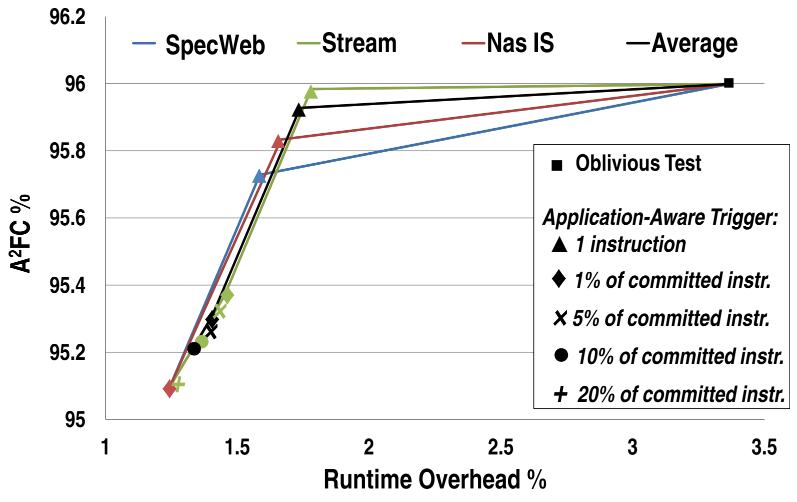


Figure 3.14 Trade-off between runtime overhead and A^2FC for an epoch length of 100M cycles. This figure shows the impact on performance and A^2FC of our Application-Aware adaptive mechanism for some significant benchmarks. The markers in the graph report different instructions thresholds triggering the functional test, from 0 (oblivious solution), to 20% of the committed instructions.

Area Overhead

Our diagnosis mechanism requires additional hardware for counters, which are used at different times as activity monitors and as fault detectors for the observation points. In the design considered, only five 64-bit hardware counters were required, yielding a total area overhead of 0.4%. We assume that the five hardware counters can be split in eight 8-bit

Table 3.6 Comparison of online testing techniques. For each technique we report the fault coverage achieved for stuck-at faults, the number of cycles required for testing and the area impact. Note that A^2Test, ACE, and CASP provide results for the OpenSPARC T1, while the other results are based on different architectures.

Solution	**Test technique**	**Test coverage (%)**	**Core downtime (cycles)**	**Area overhead (%)**
A^2Test	Hybrid	95 - 96	1.3 M - 3.4 M	0.8
ACE [41]	Structural	100	5.4 M	5.8
Bulletproof [172]	BIST	88.6	1.5 K	6
Bulletproof2 [120]	BIST	95.2	600 - 3.3 K	14
CASP [110]	Structural	99.5	240 M	6
Health Adapt. [71]	Functional	0 - 97	0 - 690 K	2.6
SW-Based [151]	Functional	90	44 K	0

counters. If these 8-bit counters are time-multiplexed three times during the test, no further area additions are needed for our framework. Moreover, our framework can take advantage of counters already present in silicon for other purposes, such as post-silicon validation or design for testability (DFT), in which case our solution will not have any hardware impact. We estimated that the addition of the parity compressors for the extra observation points requires an additional area overhead of 0.4%.

To put our results into context, Table 3.6 compares all three aspects of the evaluation for a range of previously published solutions: note how our solution achieves the best coverage for the smallest area overhead within a reasonable performance impact.

3.5 Summary

In order to define the reliability requirements for our new architecture, we first analyzed the effects of hardware failures on current processors and on state-of-the-art reliability solutions. Unfortunately, no previous technique can deliver at the same time the accuracy and the performance needed to evaluate the effect of transistor faults on modern computers. Therefore, we first developed a novel FGPA-based framework for resiliency analysis called CrashTest. Our framework is capable of automatically orchestrating a fault injection campaign of a gate-level netlist. With the objective of accelerating the reliability analysis, multiple fault locations are simultaneously inserted in the design through logic transformations. CrashTest supports an extensive and expandable collection of fault models, ranging from transient upsets to permanent defects. When compared to equivalently accurate software simulations, CrashTest speeds up fault injection campaigns up to six orders of magnitude, hence enabling accurate and detailed reliability analyses.

We used this infrastructure to test the reliability of a modern, industrial-grade microprocessor, the OpenSPARC T1, and evaluated the effectiveness of low-cost fault detectors based on monitoring software anomalies. We injected a total of 30,620 stuck-at and 20,830 path-delay faults throughout the logic of the OpenSPARC core. Our results report that software anomalies could detect more than 80% of the unmasked faults. Overall, a very small fraction (less than 1% on average) of our experiments led to silent data corruptions, even though in some hardware modules, such as the floating-point front-end and the shifter, the rate of SDCs reaches 10%.

These studies not only exposed the limitations of prior architectures, but, and more importantly, provided interesting insights on how conventional processors behave when subjected to hardware failures. We observed that software programs rarely stress all hardware uniformly, and long portions of a program often rely only on a few components. These observations inspired two design choices for our work: i) the application-aware testing technique presented in this chapter, and ii) the modular hardware organization introduced in Chapter 5.

Here we introduced a novel low-cost adaptive technique to diagnose hardware defects in the architecture developed in this thesis. Such a technique relies on both hardware and software mechanisms to adapt periodic tests to the dynamic use of a processor's structures. In our design, components that are exercised more often are tested with higher frequency and accuracy. This allows a significant reduction in performance overhead while improving software protection, since our tests focus on detecting hardware faults that are more likely to corrupt computations. The experiments presented in this chapter demonstrate that tuning online tests to hardware utilization allows a computer system to maintain elevated fault coverage (up to 95.5%) while limiting performance overhead (1.3%).

In conclusion, this chapter analyzed the reliability limitations of current microprocessors, and proposed techniques to detect hardware failures in our architecture. Hence, these solutions directly address the first issue affecting future semiconductor technologies: the fragility of transistor components. The following chapter will focus on addressing the second concern, the challenges related to managing specialized hardware units, and will present the hardware adaptability mechanisms developed in our architecture. As detailed in Chapter 4, hardware adaptability provides two advantages. First, it empowers our design to match hardware resource to application demands. Second, it enables our system to work around the defective components diagnosed through the techniques illustrated in this chapter.

Chapter 4
Hardware Adaptability

In the previous chapter we discussed how the problem of increasing transistor fragility can be addressed to extend the lifespan of future silicon chips. In this chapter we discuss another critical issue for upcoming systems: peak power consumption becomes a larger concern for all modern digital systems, computer architects are starting to deploy an increasing number of specialized hardware features that can boost both performance and energy efficiency.

We begin by first outlining a family of applications that would benefit from the ability to leverage specialized hardware components. Adaptability, as this growing characteristic of modern designs is called, empowers hardware designs to deploy and leverage specialized functional units, enabling improved computational efficiency.

With the goal of providing hardware adaptability, this chapter presents a low-cost software-based solution that enables a chip to adjust its operations to both user needs and hardware constraints. The mechanism developed here, called Cardio, empowers components to exchange runtime information concerning their condition and utilization. This information is handled by a software distributed system, which reconfigures and optimizes hardware features to match application demands, without relying on a central manager to coordinate this activity [143]. In summary, the solution presented in this chapter enables a design to dynamically adapt its operation to match user demands and system requirements, while also allowing it to reconfigure around hardware failures by using the techniques introduced in Chapter 3.

4.1 Chapter Organization

Section 4.2 introduces an example of a family of applications that benefit from specialized hardware. Section 4.3 then analyzes the requirements that drive our adaptable hardware solution and the three principles that guide our design: i) flexibility, ii) low area and performance impact, and iii) quick responsiveness to hardware changes.

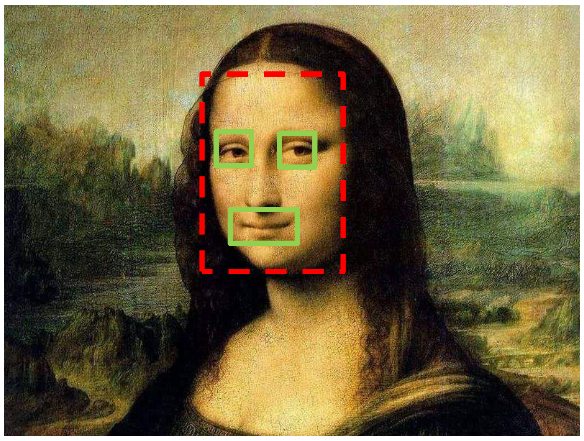


Figure 4.1 Example of computer vision application. A computer vision application might be designed to recognize features in this image such as a face.

Then, Section 4.4 details Cardio's protocols to manage system resources at runtime. Briefly, each hardware component periodically broadcasts local information about its condition and functionalities. These diagnostic messages are then collected and aggregated by a distributed resource manager, which relies on software routines to trigger a hardware reconfiguration when needed. Section 4.5 presents the experimental evaluation of the adaptive technique developed in this dissertation. Despite Cardio's extremely low silicon area profile, our results demonstrate that it is highly effective and responsive in reconfiguring a computer system to hardware alterations. Finally, we show that our solution has very little performance and energy impact on systems deploying specialized functional units.

4.2 Better Computer Vision With Specialized Hardware

Computer vision is concerned with the theory and the mechanisms that allow artificial systems to comprehend the physical world around them. Reaching this target often requires interaction among image processing, artificial intelligence, and machine learning. Typically, the final goal of computer vision algorithms is to analyze, process, and discriminate among found in images and videos. Figure 4.1 shows an example where a computer vision application utilizes a feature detection algorithm to locate a person's face. Beyond this simple example, there is a wide variety of applications, such as defense, surveillance, and autonomous vehicles, which already successfully employ computer vision algorithms.

The growth and the commercial success of computer vision is a remarkable achievement for modern technology. Continually increasing computing performance has enabled computer vision algorithms to be executable in most modern computer systems. As more applications make use of computer vision algorithms, there is a growing demand for processors that are optimized to accomplish such computationally intensive tasks [1, 137].

It has been recently shown that hardware specialized to accelerate computer vision algorithms can improve performance and reduce energy demands by orders of magnitude [35, 36]. Empirical studies of such designs demonstrated that the deployment of optimized data paths and memory subsystems can greatly reduce execution time.

A hardware system that could dynamically activate/deactivate and tune its components to the application would maximize resource utilization and optimize performance for a given power budget. This opportunity is not unique to the family of applications presented above, as it is available in most computer chips deploying specialized hardware modules [29, 198]. Hence, designs that could monitor and manage their available resources at runtime can provide a variety of possible tradeoffs and match the available hardware resources with application demands. The remainder of this chapter will introduce Cardio, the solution developed in this thesis to manage hardware components, enable system-level reconfiguration and harness the potentials of hardware diversification.

4.3 Hardware Adaptability through Cardio

Adaptable hardware can optimize the utilization of available resources, thus maximizing computational efficiency. In order to understand how adaptability can help, we may analyze the studies performed on the Sun OpenSPARC T1 in Chapter 3. This core includes a significant number of specialized functional units, such as cryptographic accelerators and support for single instruction multiple data (SIMD). Our results indicate that programs relying on one specialized functional unit do not typically utilize the others. Such behavior is also observed among the different execution phases of one application. As mentioned previously, adaptable systems can tune resource assignments based on dynamic application needs, enabling a system to achieve different performance and power goals.

Lastly, adaptability can help overcome permanent hardware failures. In modern systems-on-chip, faulty processing elements can be disabled without compromising the rest of the system [127]. At the time of writing, commercial designs leverage this aspect only to boost manufacturing yield [167]. However, as suggested by the empirically evidence in Chapter 3, a large portion of permanent failures in modern processors (about 60%) results in neither service disruption nor silent data corruptions. This surprising outcome is due to the fact that a partially faulty processor can still correctly execute a large number of instructions, as also observed also in prior research [6, 149]. In this context, hardware adaptability also offers the possibility to salvage a partially faulty processor. This goal can be achieved, for instance, if a program only composed of integer instructions is scheduled to execute on a processor with a non-functional floating-point unit.

Cardio allows for hardware adaptability at a very low cost. It enables architectures to deploy and leverage specialized functional units to improve computational efficiency. The remainder of this chapter focuses on a general technique applicable to any multiprocessor system, while Chapter 6 will adopt the principles and the protocols presented here to provide hardware adaptability to the novel distributed architecture developed throughout this dissertation. For the sake of simplicity, and without losing generality, the following sections will consider a CMP design composed of homogenous processing elements.

4.3.1 Design Philosophy

An adaptable hardware system must resolve two challenges:

1. hardware components and functionalities in the system must be diagnosed as either available or unavailable;

2. the connectivity and possible communication paths among available components must be known and maintained up-to-date.

Current solutions for runtime on-chip adaptability typically rely solely on expensive hardware mechanisms to resolve both challenges. Furthermore, no previous research provides a complete solution for distributing and managing components' diagnostic information

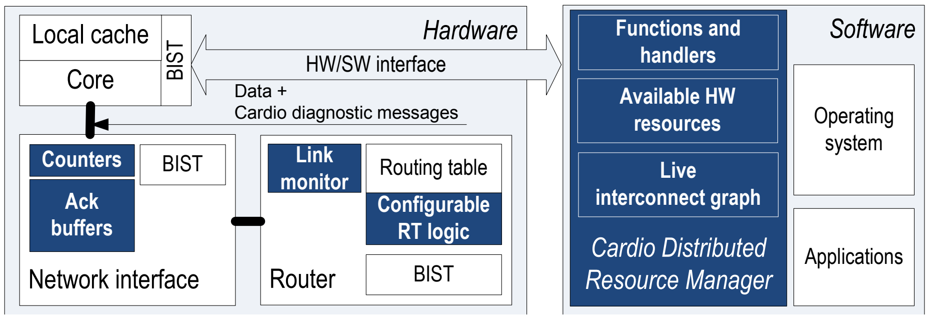


Figure 4.2 Cardio architecture overview. Cardio hardware and software additions are highlighted in the figure. Communication endpoints are augmented with acknowledgment buffers and counters to handle transmission issues; routers are enhanced with logic to diagnose link-connectivity and to reconfigure routing tables. Each general purpose core in the system executes an instance of the distributed manager.

in CMPs. We equip each Cardio unit with the capability of periodically broadcasting diagnostic information about its state to the entire system. In our proposed design, a distributed software resource manager executes on each of the general-purpose cores. This resource manager dynamically maintains and organizes information about the CMP's hardware features. Figure 4.2 shows a high level schematic of the hardware and software additions necessary to equip a baseline CMP system with Cardio. With the goal of handling hardware failures, hardware units can also feature diagnostic mechanisms such as the ones detailed in Chapter 3.

To maintain a low area cost, hardware modifications are limited to the intra-chip communication subsystem and consist of enhancements to both network interfaces and routers. Network interfaces are augmented with: i) a buffer containing packets injected into the network and waiting for acknowledgment and ii) a set of counters to trigger automatic retransmission in case of time-out. Routers are enhanced with: i) a link monitor to collect information about the components directly connected to the router and ii) a configurable routing table to redirect NoC traffic around faulty components. Cardio software additions are more significant, and consist of:

1. a data structure to contain information about the CMP's state (list of cores and functions available);
2. a graph of the connectivity among the functional on-chip components;
3. the software routines necessary to update the system and handle diagnostic messages.

4.4 Cardio Runtime Operation

Consistent with the execution model detailed in the previous chapter, Cardio also partitions workload execution time into epochs. Because resource manager instances execute independently on the CMP's cores, they must synchronize so that they all have an identical image of the system's resources and, if needed, can enforce a sound system reconfiguration. The problem of reaching a common decision among several components is a simpler instance of the "Byzantine Generals' problem" [103], called the "consensus problem" [58]. Solving this problem in our case consists of providing common knowledge of the available resources to all functional cores in the CMP – also known as "consensus vector" in the literature. Cardio relies on diagnostic message broadcasts to provide an efficient solution to this problem: the remainder of this section details the mechanisms leveraged to achieve this goal.

While applications execute on the system, processors and NoC routers periodically and independently suspend their tasks to assess runtime characteristics of the underlying hardware. These local tests are not globally synchronized, and the only constraint imposed by Cardio is that all hardware units must complete their self-tests by the end of an epoch.

After each self-assessment completes, its outcome is broadcasted to the rest of the system. Since Cardio targets a low performance impact, each unit shares only the necessary diagnostic information. For instance, the interconnect is tested every few thousands cycles: if each router were to broadcast test outcomes so frequently, these messages would severely burden the communication infrastructure; therefore routers broadcast system-wide updates only upon discovery of a new fault in a unit. More frequent diagnostic tests lead to more prompt reactions to failures, but also entail higher performance impacts and diagnostic message proliferation. Indeed, diagnostic frequency is a design trade-off analyzed in Section 4.5. Since one of Cardio's objectives is to promptly report dynamic hardware changes, we rely on techniques similar to the one presented in Chapter 3 to diagnose faults in hardware components [142].

Diagnostic messages are collected by the various instances of the distributed software manager, which, in turn, updates two internal data structures: the list of available hardware resources and the graph of the functional interconnect links (shown in the right side of Figure 1). The first structure lists all hardware resources available in the CMP. The second structure is used to compute the routes to be followed by packets in the NoC. When a resource manager instance detects a change in hardware functionality, the operating system is notified and the chip is reconfigured to address the new state of the hardware. If critical hardware changes are detected, for instance due to newly discovered failures, all application results produced since the beginning of the last completed epoch are discarded. Otherwise, if no critical event is identified, the resource managers commit the results produced in the previous epoch. Note that, once a critical event is discovered, the state of each active component in the CMP must be recovered in order to restart software execution. For this purpose, Cardio can rely on either software or hardware checkpoint techniques [150, 176].

The mechanisms developed in Cardio can also be deployed to improve a system's adaptivity to network traffic, components usage, and temperature, by simply adding more detailed data to the diagnostic messages. For instance, each router could provide information about link usage. Routers experiencing high utilization can then broadcast this information to the entire system, so as to trigger an eventual hardware reconfiguration. As we demonstrate in this work, a hardware system can rely on diagnostic message broadcasts to promptly react to hardware alterations without hindering its performance. The design principles developed in Cardio are therefore extendable to a variety of digital systems, as long as enough hardware resources are available to execute one instance of the resource manager.

Since our hardware availability assessment and reconfiguration procedures differ for processor cores and chip interconnect, we discuss them separately in the following two sections.

4.4.1 Core Monitoring

Handling hardware changes and online failures on processors requires several steps. As soon as the diagnostic tests detect a significant change in the hardware fabric, the system suspends its execution and reconfigures to optimize software execution. In order to gather and distribute system-level knowledge about the functionalities of a CMP's cores, all processors in a Cardio-enabled CMP follow the sequence of operations illustrated in Figure 4.3.

Cores in modern CMPs rely on independent clock signals, which cannot be easily synchronized to provide a global signal to all components in the system. Therefore, each core in Cardio asynchronously and periodically suspends its normal execution to perform a self-assessment of its functionalities (step 2 in Figure 4.3). These intervals are typically several tens of million cycles long - computational epoch lengths adopted in previous works are 10M, 20M, 100M and 1,000M cycles. Partitioning regular execution into epochs slows down performance by 2% to 30%, depending on workload, testing technique, and epoch length [41, 110, 142]. It is worth noting that even briefly suspending cores might cause jitters in the execution of an application. However, this effect can be significantly mitigated through modifications to the operating system's scheduler [111].

Several techniques have been proposed to perform online checks of digital designs, ranging from structural to functional tests [41, 142]. If the self-tests are successful, the processor's architectural state and its memory state are checkpointed (step 3 in Figure 4.3). Test results are wrapped in a diagnostic message marked with the unique identifier of the tested core and broadcasted to the entire system (step 4 in Figure 4.3). For instance, it may be possible to report a processor with a missing or non-functional floating point unit as available, but only capable of executing integer programs [142].

Since all resource managers must agree on the available hardware resources, Cardio imposes a barrier to allow all cores to synchronize their information about the state of the system. While waiting to receive all diagnostic messages from the other computational elements, cores may start executing the subsequent epoch. For instance, in step 5 of Figure 4.3, core 0 begins computation in the new epoch even if it is still missing a diagnostic message from core 3. Since system synchronization is also used to ensure the health of the system, a local checkpoint can be safely committed only when all diagnostic messages from the functional cores are received (step 6 of Figure 4.3). If n is the number of cores in the CMP, a core may receive at most $n - 1$ unique diagnostic messages from other cores for each epoch.

A disabled or faulty core is not required to advertise its status to the rest of the system: the other resource manager instances will detect a missing diagnostic message at the end of their speculative epoch by mean of a local timeout. Once a fault is detected, a special message is sent to all cores in the system to rollback to the previously synchronized check-

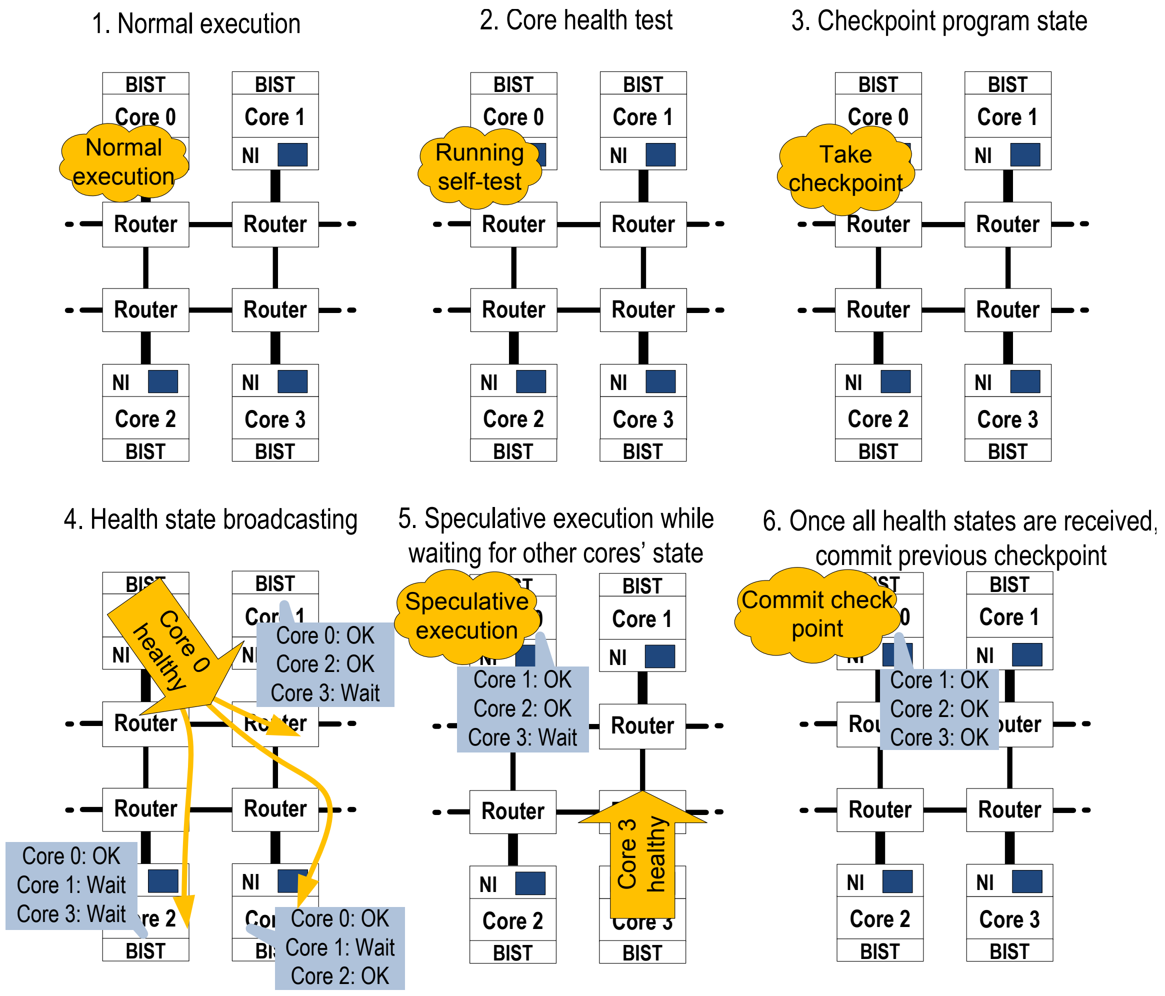

1. Normal execution
2. Core health test
3. Checkpoint program state
4. Health state broadcasting
5. Speculative execution while waiting for other cores' state
6. Once all health states are received, commit previous checkpoint
BIST
Core 0
Core 1
Core 2
Core 3
NI
Router
Normal execution
Running self-test
Take checkpoint
Core 0 healthy
Core 0: OK Core 2: OK Core 3: Wait
Core 0: OK Core 1: Wait Core 3: Wait
Core 0: OK Core 1: Wait Core 2: OK
Speculative execution
Core 1: OK Core 2: OK Core 3: Wait
Core 3 healthy
Commit check point
Core 1: OK Core 2: OK Core 3: OK

Figure 4.3 Core monitoring and recovery in Cardio. To maintain an up-to-date state of the available cores in the system, Cardio relies on a five-step sequence. 1) The cores perform their normal functions. 2) Core 0, independently from the other Cores, executes a self-assessment procedure to report its state and number of functionalities to the system. 3) If the test completes successfully, a local checkpoint of the current core state is taken. 4) A diagnostic message is broadcasted to all other cores to signal that core 0 is functional. 5) Before core 0 can commit its computation, it must receive successful fault-free acknowledgments from all cores that were functional in the epoch that was just completed. This step is accomplished to ensure that no hardware failures affected the system. In the meantime, it can speculatively continue its execution. 6) Finally core 0 receives the last positive fault-free acknowledgment from core 3 and commits its results up to the last checkpoint.

point, so as to prevent them from committing potentially incomplete or corrupt speculative results. Cardio provides a synchronization mechanism that eases the deployment of a global checkpoint system. The checkpoint solution used in Cardio is very similar to ReVive [150], where each node logs the content of the cache lines that are written during a speculative epoch. After a next checkpoint is established, the logs from the previous checkpoint are committed to memory. If the self-tests are successful, both the processor's architectural state and its memory state are checkpointed. In such event, all computations performed since the previous checkpoint cannot be trusted to be correct, since faulty hardware might have been exercised during the current epoch. This mechanism is deployed to prevent critical hardware events (such as failures) from silently corrupting program output. Silent hardware corruptions are extremely dangerous and current unprotected microprocessors cannot cope with them, as we previously reported in Chapter 3. Therefore, all speculative results are discarded and the faulty hardware is identified and isolated. Once a core's functionalities change, for instance its operating frequency is reduced due to high local temperature, the OS can migrate active applications to optimize their execution of the available resources.

In order to prevent potential livelocks and guarantee that all cores in a same connected region have a consistent view of the hardware, resource managers exchange checksums of the available chip resources. In case of checksum mismatch, Cardio forces all cores to suspend their activity to drain all in-flight messages from the system. All cores then restart their diagnostic protocol and exchange a second set of checksums about the state of the system. This second time both the diagnostic messages and the checksums are guaranteed to arrive in time, as the system is not burdened with other traffic.

Our diagnostic protocol is inspired by the one reported in [58], which has proven to be deadlock and livelock free if all the following conditions are met:

1. the communication system is reliable and only cores can be subjected to changes – note that communication problems that disconnect one or more cores are equivalent to cores disappearing;
2. each core can unequivocally determine the sender of any received message;
3. any core's failure to send messages is detectable;
4. any non-disabled core can broadcast information to all non-disabled cores.

Our system meets condition 1 since: 1) temporary communication glitches are tackled through our end-to-end network-level retransmission protocol, and 2) permanent communication errors cause the affected cores to be detected as faulty. Condition 2 is fulfilled as each core marks all generated messages with its unique ID. Timeout counters at the network level meet condition 3. Finally, condition 4 holds true because Cardio can only be applied to interconnects that support message broadcasting. Thus, at the beginning of a computational epoch:

1. each j of the r available cores is tested, and its diagnostic v_j message is broadcasted to the system (for instance, 1 if available and 0 if the core is not present).
2. Each instance of the resource manager:
   (a) if it receives a value v_j from all cores, $1..r$, then it takes v as its system's availability image; broadcasts v (or its checksum) to the system; waits until the end of the computational epoch;
   (b) otherwise, if it does not receive a message from at least one of the cores expected to be available (a timeout occurs), then it broadcasts such information to the system.

At the end of the epoch:

1. If each resource manager received a matching value v from all other cores during the epoch just completed, then it takes v as its consensus value and commits results generated in the previous epoch.
2. Otherwise, a hardware change has been detected and the system is rolled back to the previous checkpoint and is reconfigured to handle the alteration.

Since message payload is typically protected through error correcting codes, we do not consider faults that may silently corrupt the data carried by a packet [126]. Cores advertise their status and functionalities to the rest of the system when self-assesment succeeds. However, it is possible that a fault in the self-test logic causes a processor to incorrectly advertise its status as available. The probability of such events can be arbitrarily reduced based on the quality of the periodic self-tests: for instance, runtime test solutions that achieve fault coverage very close to 100% have been proposed [41, 110]. In addition, the self-test logic can be protected with traditional reliability mechanisms such as triple-modular redundancy.

In order to maintain knowledge about the functional hardware, each resource manager builds a list of available cores from the diagnostic messages received. Diagnostic messages must synchronize, so all cores can receive them before the end of the next epoch. Since each core handles diagnostic message generation independently, Cardio relies on real-time counters to trigger diagnostic message broadcast. The introspective operations performed by

Cardio rely heavily on such timers, and their hardware should be replicated to ensure their functionality. We send at least two copies of these messages within one epoch to guarantee their timely arrival: in fact, even if transmission glitches cause the loss of one message, the second is likely to arrive on time. Note that it is still possible, although unlikely, that both messages get lost due to transient failures. This would cause the system to flush all the in-flight operations and initiate a new system-wide test procedure.

4.4.2 Interconnect Monitoring

Interconnect correctness and performance are fundamental for any CMP. On-chip routers deliver messages between cores, and a single router can connect multiple processors. Cardio can be successfully adopted in any NoC topology, as interconnect state and routing information are handled in software by the distributed resource manager. Previous work characterized NoC malfunctions as either: 1) corruptions in the payload/data or 2) errors in the delivery system [3]. The first kind of errors can be easily addressed with error-correcting codes and retransmission. The latter can cause packet loss or network deadlock, and both behaviors can be directly mapped to malfunctions in a router's links. Here, we introduce a routing algorithm that dynamically discovers link failures in an arbitrary network and updates communication routes accordingly. In general, two families of routing algorithms are available for this purpose: link-state and distance-vector [102]. On one hand, self-configuring NoCs typically adopt some flavor of the distance-vector protocol, because its limited complexity is well-suited to hardware implementations. On the other hand, algorithms based on the link-state protocol introduce lower communication overhead, and therefore converge faster and scale better than those based on distance vector.

In typical link-state protocols, for instance those developed for computer networks, every node constructs a graph of its local network connections and broadcasts it to the others. Then, each node independently computes the best path from itself to every possible destination in the network. Hence, network nodes only exchange information about their local connectivity, but must perform complex computations to generate network routes. Cardio overcomes this drawback by delegating route generations to the software resource manager.

Figure 4.4 shows the steps performed by each router when the online testing procedure is activated. First, each router independently suspends its activity to perform a self-check on its own hardware structures, for instance testing its input and output buffers and its crossbar (step 2 in Figure 4.5). Any of the several techniques proposed in the literature can be adopted for this purpose [40, 55]. The outcome of this test determines whether the router

```
 1: Drain output links
 2: Test Router logic
 3: For each output link:
 4:   Send discovery request
 5: For each output link until timeout
 6:   If discovery response received
 7:     Update link table
 8: If link table changed
 9:   Broadcast updated link table
10: Resume operations
```

Figure 4.4 Router periodic self-assessment procedure. First, the online testing algorithm on the router tests the router's hardware. Then the state of the direct links between the router and its neighbors is checked. Directly connected neighbors that do not respond within a certain time threshold are considered unavailable. Note that only changes to the local link table are broadcasted to the system.

is operational. Once this first check phase is completed, the router being tested probes its links to discover all directly connected neighbors. Each node in the NoC independently performs local link discovery. To maintain an accurate state of the local connections, each router periodically generates a discovery "heart beat" that is sent to all adjacent nodes, as illustrated in step 3 of Figure 4.5. A router receiving a "heart beat" discovery responds including its node ID (step 4 in the Figure). Each link monitor then populates a table where every functional local link is associated with the ID of the node connected to it (step 5 in the Figure). Routers also store the unique identifiers of all directly connected cores.

Routers may trigger the detailed hardware tests performed in the first step rather infrequently, since accurately testing NoC hardware components often requires a considerable amount of time. Indeed, other directly connected routers can discover critical failures that jeopardize a router functionality, and failures causing packet corruption can be detected by error-checking codes.

After these two phases, each router is able to detect failures that prevent communication with its directly connected nodes, perhaps because a failure interrupts a communication path. Once a router discovers a new link malfunction, it discards all packets directed towards the broken link. The updated local link table is then broadcasted to the system to notify the resource managers about the change in network topology (step 6 in Figure 4.5). Due to storage and performance constraints, the period between two subsequent link tests is limited to a few thousand cycles, as discussed in Section 5.3. All links, even those previously considered faulty, are periodically tested. This allows routers to recover parts of the network that are only temporarily unavailable, for example due to intermittent faults or high traffic congestion. When a failure is detected, the system starts the following hardware reconfiguration routine:

1. the router that discovered the failure broadcasts this event to the entire system;
2. resource managers receive this notification, suspend the current execution, and discard

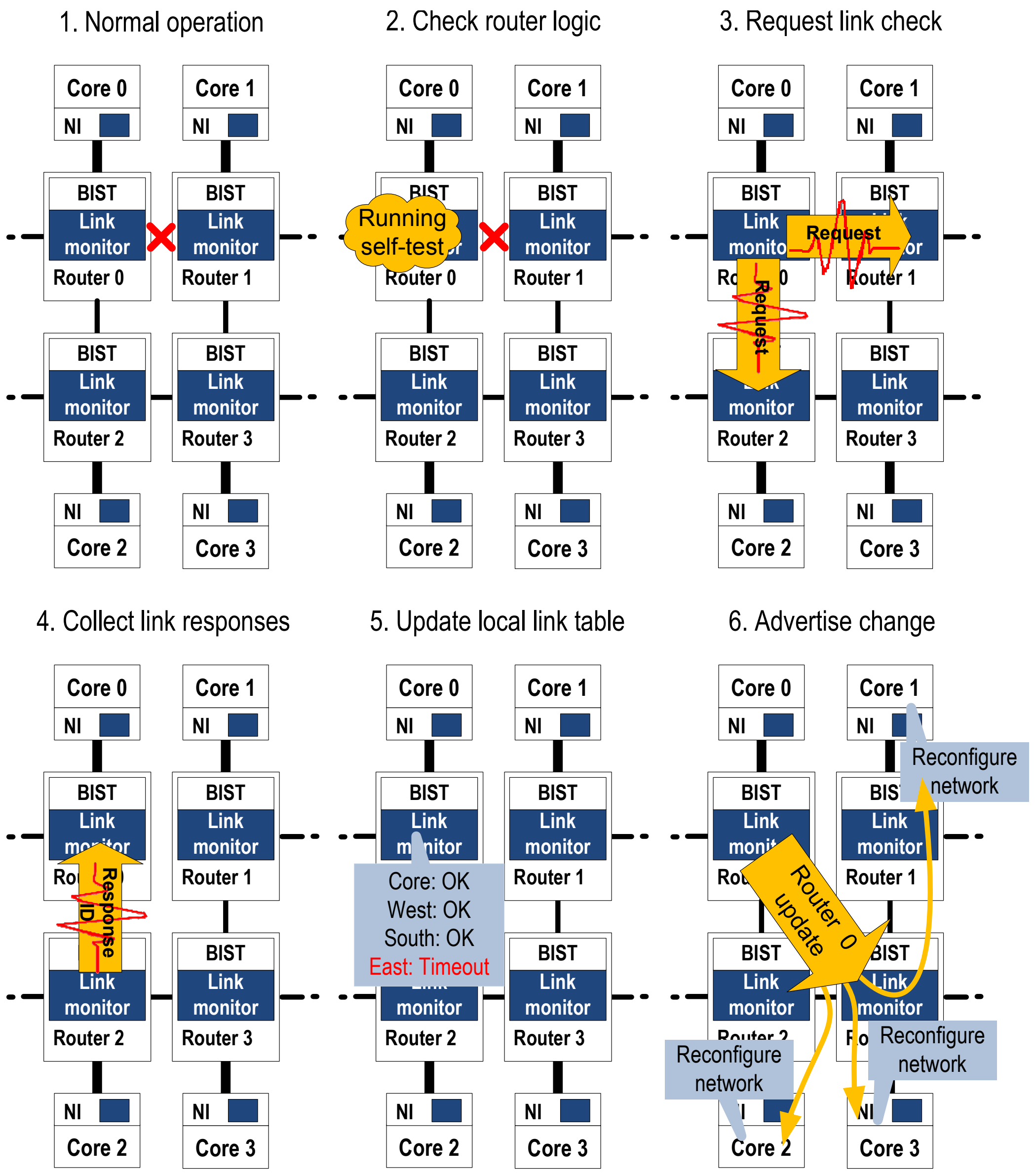


Figure 4.5 Dynamic interconnect management in Cardio. Self-discovery and reconfiguration in the interconnect are organized in five steps. To reduce the amount of extra traffic in the CMP, only topology *changes* are advertised. In the figure: 1) The NoC performs its normal functions. 2) Router 0 suspends its execution to perform a self-test routine. 3) Since its hardware is found to be functional, discovery messages are sent to all output links. 4) Router 2 replies to the request with its Router ID. 5) No response is received from Router 1 within the deadline imposed by the timeout, thus a failed link is detected. 6) Because of the new fault, Router 0 broadcasts a diagnostic update requesting to reconfigure the network.

all speculative computations;

3. software routines are triggered to compute new routes and the hardware is reconfigured accordingly;
4. the system rolls back to a previous checkpoint to restore software state and restart execution.

Since our dynamic network testing routine might discard in-flight packets, we deploy a retransmission mechanism to avoid communication loss. Cardio addresses sporadic transmission glitches through an end-to-end acknowledgment protocol: every time a message successfully reaches its destination, the receiver notifies the sender. All interconnect endpoints therefore are enhanced to maintain hardware counters and store pending messages waiting for acknowledgment. These counters are incremented every cycle, and any message that does not receive an acknowledgment within a certain time threshold triggers a timeout. In case of timeout, the network interface retransmits the timed out message; if the second attempt is also unsuccessful, the network interface affected by this problems notifies its directly connected cores of a potentially more severe reliability threat by raising a hardware exception. Acknowledgments may be sent through specialized packets or may be piggybacked to regular data packets. Acknowledgment buffer size is a storage and performance trade-off, which is evaluated in Section 4.5.

4.4.3 Cardio Distributed Resource Manager

Cardio's distributed resource manager is responsible for monitoring and managing the system's reconfigurable hardware. This light-weight software layer leverages the information collected from the local tests to assess hardware availability and connectivity. When necessary, Cardio suspends user applications running on a core to execute resource manager maintenance routines. User applications are also interrupted every time a hardware component raises an exception or when a diagnostic message is received.

Resource managers use the information about local connections broadcasted by the routers to generate a connectivity map of the on-chip network. All cores in a connected region reconstruct the same topology. If the interconnect is partitioned into multiple disconnected regions, each core running an instance of Cardio's resource manager only reconstructs the region to which it belongs. Once the interconnect graph is built, one local resource manager (for instance the one running on the core with the lowest identifier number) computes and distributes all routing tables, thus configuring the system to permit communication among all available components. From the diagnostic messages that are broadcasted to the system each resource manager also populates the list of the available hardware components and the interconnect graph. As previously discussed, a checksum of these two data structures is transmitted to all cores in the CMP to verify that all resource manager instances agree on the current state of the system. If a checksum mismatch is detected, resource managers initiate a re-negotiation among themselves, eventually pruning routes not accessible by one or more cores.

4.5 Evaluation

Cardio is a distributed mechanism to manage and organize on-chip resources at runtime and therefore it adds extra on-chip traffic due to the diagnostic messages exchanged by the self-checking hardware components. Thus, our experiments focus on measuring Cardio's impact on the system's interconnect by considering a variety of topologies and workloads. We first focus on finding the optimal size of the acknowledgment buffers at the NoC endpoints and measure packet latency sensitivity to interconnect discovery.

We then evaluate how Cardio handles dynamic hardware alterations. In order to estimate our solution's reactivity to system-level changes, we measure Cardio's dynamic response to hardware alterations due to permanent failures. We first analyze the effects of hardware failures on the static performance of a system, where a number of changes to the system's connectivity are made at design time (before starting our simulations). We then studied the behavior of Cardio in the presence of dynamic failures, therefore evaluating the reactivity of our design to hardware changes. We also report the impact of our solution on the interconnect performance and energy. In order to assess Cardio's energy footprint we measure the extra traffic introduced by our solution.

4.5.1 Experimental Setup

We divide our experimental setup in two different phases. In the first phase we use a dedicated simulator to explore the tradeoffs of our solution, measure the responsiveness of our design to sudden hardware changes, and evaluate its cost. In the second part we concentrate on measuring Cardio's impact at the system-level, and for this purpose we adopt an open-source architectural simulator, gem5.

Cardio's Interconnect Overhead and Responsiveness

We perform an initial set of experiments using a system-level C++-based simulator. In our model, communication details are separated from the implementation details of functional units. We used this infrastructure in order to focus our evaluation on the performance effects of the extra communication due to Cardio and to measure its response time to dynamic hardware changes. In order to evaluate Cardio's adaptability, we focus on sudden hardware alterations due to runtime failures, as the response time to such events can be easily measured by comparing fault injection time against a complete system reconfiguration. Since our target is to explore the effectiveness of hybrid HW/SW solutions and protocols in relatively large CMP systems, we developed an infrastructure that would allow quick turn-around for a range of architectures and HW/SW co-designs. To this end, we developed a simulation framework that includes models at different levels of abstraction: a more accurate model for the interconnect and a simpler one for the processing elements in the system. Core functionalities are implemented at the transaction-level through clock counters, while the interconnect model is cycle-accurate at the packet granularity (we do not consider flit-level structures or virtual channels).

Two fault models have been deployed for the interconnect links: in the first all packets attempting to traverse a link are dropped *(drop-packets)*, and in the second a communication path is blocked at a selected link *(hold-packets)*. These two fault models are based on faulty behaviors observed in RTL simulations with detailed fault models [55].

The CMP simulated in this first set of experiments is composed of 16 cores, each connected to a dedicated network interface. We considered four different interconnect topologies: ring, mesh, torus and crossbar. The system frequency is set at 2.4GHz, with five-stage routers transferring packets of up to 32 bytes in size. Packets are buffered at every router; routers can store up to two packets at the time. In our experimental evaluation we adopted source routing, embedding routing information in the packet itself. Routing tables are stored in the network interfaces and communication paths are computed by the resource manager using the up*/down* routing algorithm [163]. Cardio does not impose limitations on the routing algorithm we have adopted: these design choices were driven by the goal of simplifying simulation troubleshooting.

In our work we considered uniform random traffic as well as traces from the SPECMPI benchmark suite [125]. On one hand, random traffic ensures uniform link utilization so packet latency and fault impacts are not biased by traffic patterns imposed by a benchmark's characteristics. On the other hand, traffic patterns from the SPECMPI benchmarks offer a more realistic model to evaluate Cardio's performance and traffic overhead. For uniform random traffic injections we report packet injection rates as the probability that a core injects a new packet in the network (in %). For the SPECMPI applications we collected communication traces through the Tuning and Analysis Utilities [169]. To contain simulation time, we reduced the number of cycles between MPI transactions, thus the performance overhead we report for these benchmarks is worse than that of their original traffic pattern. Finally, we also evaluate the quality of our Cardio infrastructure on a system-on-chip design by evaluating it with the MEVBENCH benchmark suite [37].

4.5.2 Acknowledgment Buffer Sizing

In order to handle glitches in the communication system, Cardio considers any point-to-point data transmission incomplete until the sender is acknowledged by the receiver. Thus, every non-broadcasted packet is temporarily maintained in an acknowledgment buffer at the source until a confirmation message is received. The goal of our first experiment is to study the trade-off between storage and average traffic latency on network interfaces augmented by

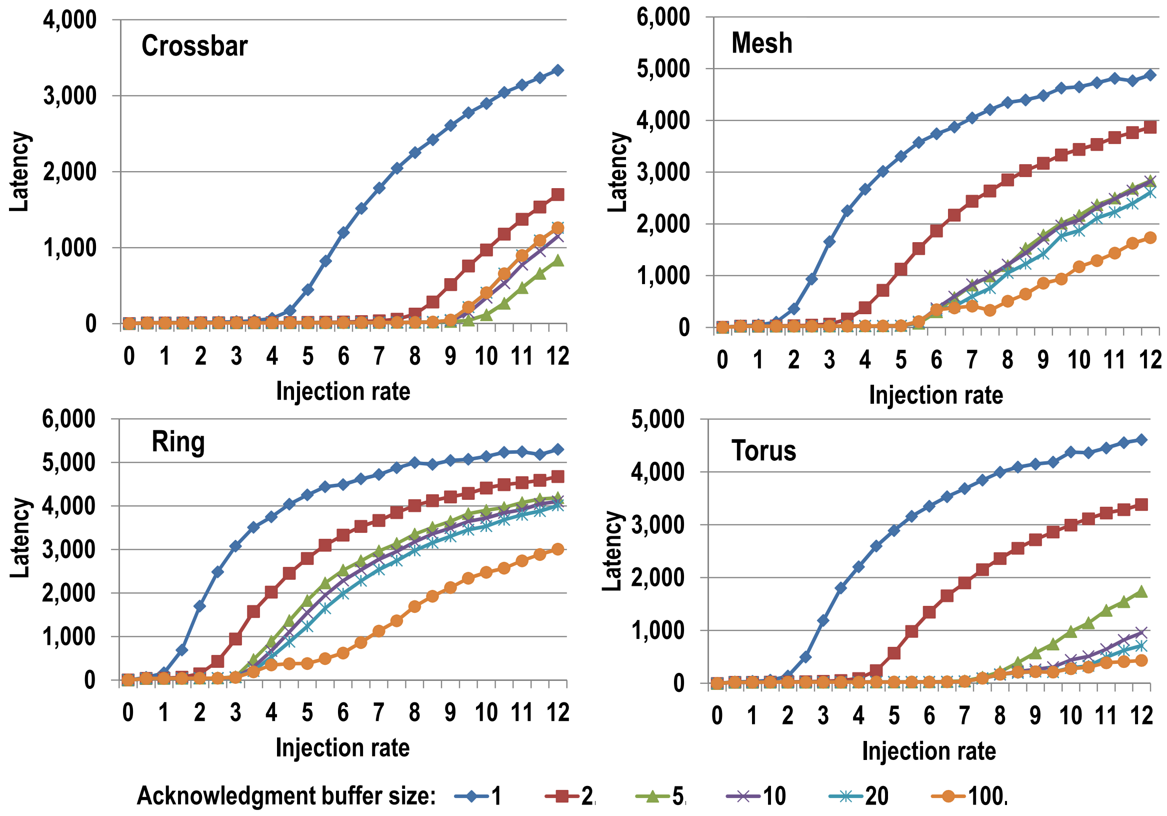


Figure 4.6 Packet latency vs. injection rate for different acknowledgment buffer sizes. Each curve represents a different acknowledgment buffer size as indicated in the legend. The X-axis represents the probability that each node (in %) injects a new packet in the network. Buffers of size 10 provide the best trade-off between storage requirements and packet latency.

packet acknowledgment buffers. We evaluated several buffer sizes, ranging from 1 data packet (that is, the network interface must receive the acknowledgment for a previous packet before transmitting the following one), up to 100 outstanding packets. No faults are injected for these experiments. Figure 5 shows the relation between the number of outstanding messages and the average packet latency. Traffic injection rate is measured as the probability of each network interface to input a new message in the interconnect at any given clock cycle, while packet latency is measured as the number of cycles between when a data packet is generated to when it is received by its destination. For the considered topologies we found that an acknowledgment buffer of 10 packets is a reasonable compromise between storage requirements and packet latency. Indeed, acknowledgment buffers containing less than 10 data packets lead to significantly worse average packet latency, while even doubling their size provides minimal benefits. In order to evaluate the area tradeoffs introduced by these retransmission buffers, we used CACTI to estimate the size of these additional memory elements [189]. The total overhead of including 10 extra buffers adds $0.027mm^2$ to the area of a core developed in 90nm technology, less than 0.2% of the area of a processor from the OpenSPARC T1 [186]. As a reference, allowing network interfaces to store 100 32-byte messages would increase the area by $0.146mm^2$. Thus, network interfaces in all subsequent experiments include acknowledgment buffers capable of containing up to 10 outstanding packets. It is worth noting that the latency curves we observed level off as the traffic injection rate increases. This behavior is due to self-throttling limitations in the injected traffic imposed by the acknowledgment buffers. Indeed, when the acknowledged buffers fill up, the cores are forced to suspend execution, creating de facto a self-throttling effect – since the acknowledgment buffers are full, the cores attempt but fail to inject packets in the network.

4.5.3 Dynamic Discovery Period

We then analyzed the effects on interconnect latency due to the extra traffic caused by the discovery packets. With this goal, we studied the sensitivity of average packet latency to the interconnect discovery period. This experiment was run with a traffic injection rate of 5% and no faults in the interconnect. From the data gathered in our analyses we observed that resource contention in the network starts to impact packet latency at injection rates higher than 5%. As the period between interconnect discoveries increases, average packet latency decreases due to bandwidth limitations. As shown in Figure 4.7, this trend is steeper for topologies such as mesh and ring, for which links are subjected to a higher contention.

Given the results obtained in this experiment, the network discovery frequency for our

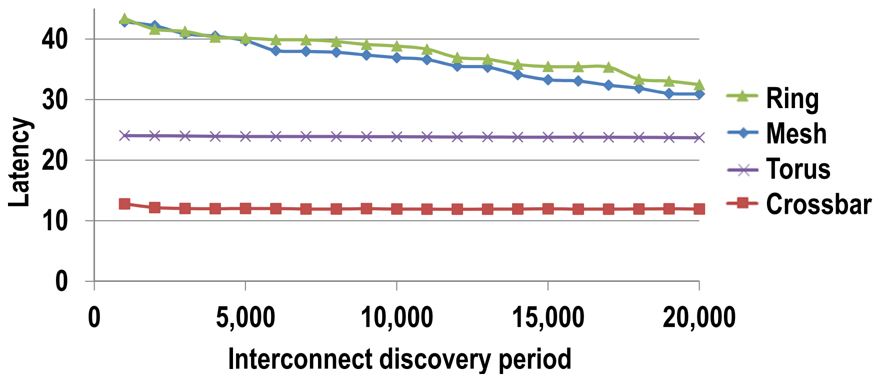


Figure 4.7 Packet latency vs. discovery period. The impact of varying the discovery period differs for different topologies: mesh and ring are more sensitive to variations due to a smaller network bisection.

subsequent analyses is based on three different discovery periods, from a very frequent periodic test of 5,000 cycles to a much slower discovery period of 20,000 cycles.

4.5.4 Static Hardware Adaptation

Our third set of experiments evaluates how a Cardio-enabled CMP can adapt its behavior in the presence of hardware alterations. For this experiments we rely on random traffic patterns. Network links are disabled (set as faulty) before starting the simulation, so as to model a number of similar systems in which connectivity is modified at design time due to arbitrary design choices. We perform these studies to show that Cardio can be adopted to ease design diversification and that various versions of a SoC can be deployed with no effort. These results also show that our solution can be used to improve manufacturing yield and to demonstrate Cardio's operation in various defective topologies.

In Figure 4.8 we first report the packet average latency as a function of the number of faulty links in the system (X-axis) and the traffic injection rate (Z-axis). As with the previous experiments, traffic injection rate is measured in probability (%) of packet injection per core. Interestingly, packet latency for the crossbar, ring, and mesh reduces as the number of faulty links increases. For the crossbar, this phenomenon is caused by the fact that a single disabled link is sufficient to disconnect a processor from the system. Therefore, as the number of faults increase, the number of active cores connected to the crossbar – and thus the traffic injected into the system – decrease. In the ring, hardware failures partition the topology in smaller, partially connected sub-networks. As shown in the graph, just 12 disabled nodes suffice to disconnect most cores from the system, therefore causing the average packet latency to plummet to zero. Figure 4.8 also shows that both the mesh and the torus can tolerate a high number of disconnected links, maintaining performance up to and beyond 20 faulty links. Nevertheless, an increase in the number of faults in the mesh leads to a lower average packet latency. We found that disabling a large number of links partitions the system into smaller sub-networks, which experience less average traffic congestion and thus lower packet latency. This phenomenon has also been reported by other researchers [55]. In contrast, the torus does not manifest this behavior. In fact, the higher number of links available in this topology allows it to maintain connectivity among most nodes in the system, even when affected by more than 10 faults. Nevertheless, disabled links do affect the possible routes of packets, thus increasing the average communication latency.

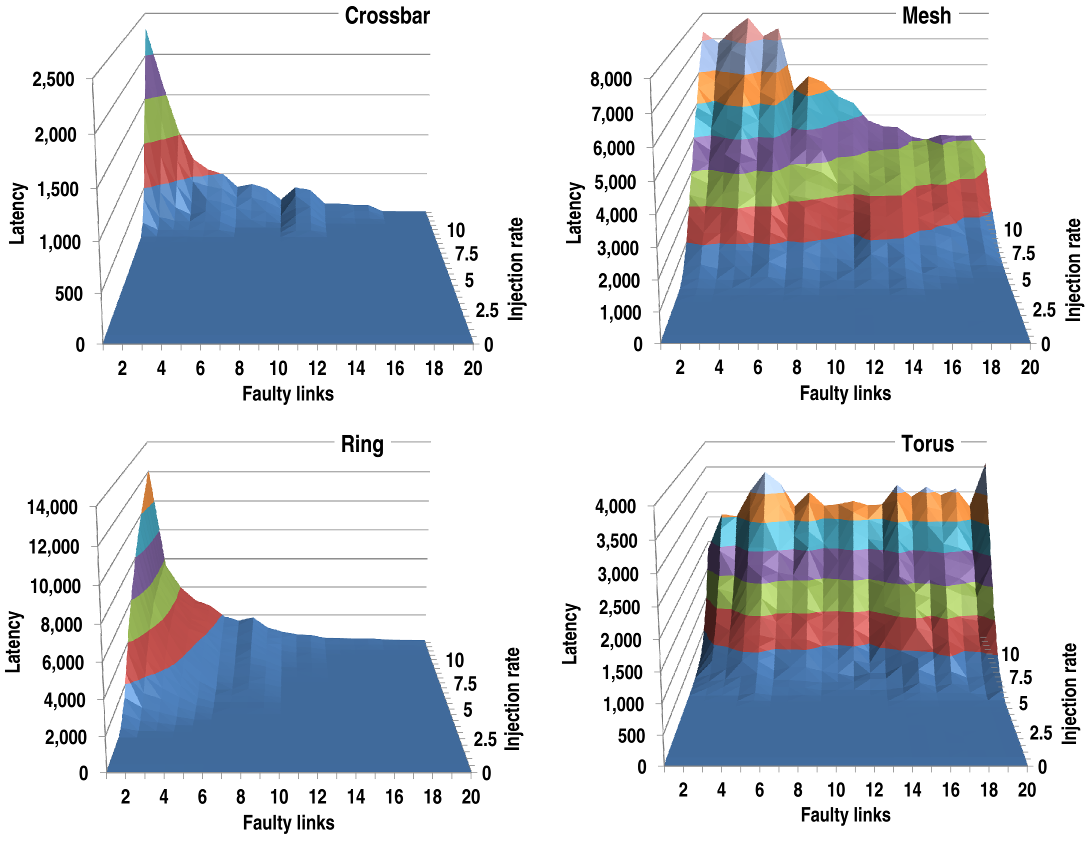


Figure 4.8 Average packet latency for faulty topologies. In these graphs we report the average packet latency (Y-axis) as a function of the number of disabled (faulty) links in the system (X-axis) and the traffic injection rate in % (Z-axis). When subject to faulty links, topologies give different responses. Topologies with higher connectivity can maintain low latency even with a significant number of faults.

4.5.5 Dynamic Hardware Adaptation

In this section we studied the short-term dynamic behavior of Cardio on a mesh at the time of occurrence of a permanent fault, evaluating its reactivity in detecting and overcoming runtime hardware alterations. For this experiment, we simulated the system with no faults for 150,000 cycles to reach a steady state, and then a randomly selected link is modeled as faulty. Two fault models were considered for this study: the first - *drop-packets* - causes the loss of all packets directed towards a faulty link; the second - *hold-packets* - forces a faulty link to stop all packets trying to traverse it. In order to provide insights into Cardio dynamic behavior, we analyzed the system at windows of 500 cycles and report, on the Y-axis, *the average latency incurred by all packets* generated during each analyzed window. In this experiment we considered discovery periods of 5,000, 10,000, and 20,000 cycles. To stress the interconnect with a moderate amount of traffic, we set the packet injection rate at 5%. Through native execution profiling, we measured that the time required for the distributed resource manager to recompute the routing tables is approximately constant at 10,000 cycles. We also estimated that each routing table requires 450 cycles to update, representing a serial write process for 15 routes of 15 hops each, writing 2 bits per hop [114]. In these experiments the network discovery period starts when the fault is injected to demonstrate the worst-case performance of our solution. In our evaluation, we disregarded the extra traffic introduced by core diagnostic messages, since their transmission frequency is three orders of magnitude lower than for the interconnect components [40].

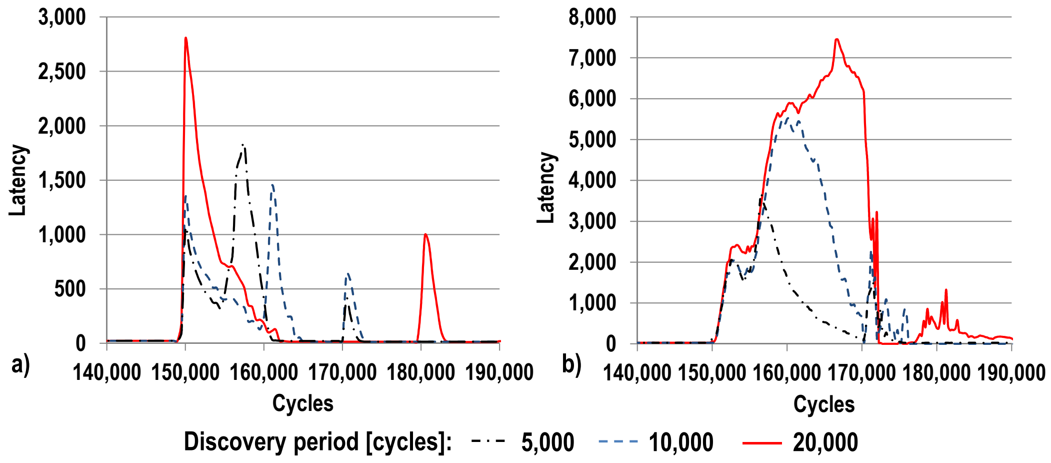


Figure 4.9 Effect of a dynamic fault on a link. This graph plots the average time necessary for a packet to reach its destination; packet latency is averaged between all packets generated in a window of cycles. In this scenarios the link is broken at cycle 150,000 and two fault models are considered: **a)** *drop-packet*; **b)** *hold-packet*.

Results from the *drop-packet* fault model are reported in Figure 4.9.a), where we distinguish a minimum of two and a maximum of three latency peaks, depending on the discovery period. The first peak is caused by the occurrence of the fault, and affects all packets that need to be re-transmitted due to the faulty link. After a certain amount of time, and directly related to the network discovery period, a first router detects the problem locally and consequently broadcasts the updated system state. The first interconnect reconfiguration process causes the network to temporarily stall, resulting in the second peak that is observable in the graph. The third peak shown in the graph is due to a second system reconfiguration and is triggered by a later detection of the fault by a second router.

The impact of the *hold-packets* fault model is more dramatic: a fault's effect is not limited to packets in transit between two nodes, but rapidly propagates to a vast portion of the CMP, as demonstrated by the much higher average packet latency reported. Indeed, this second fault model congests multiple links: input and output buffers at the routers that are connected through the broken link fill up and cause a domino effect to their neighbors and then to the rest of the network. As reported in Figure 4.9.b), the longer the period between hardware tests is the more dramatic the fault's effects on the overall system are.

4.5.6 Performance and Traffic Impact

In this section we study the impact of our solution on interconnect performance and its communication overhead. For this last study we report the extra execution time experienced when running SPECMPI benchmarks and the percentage of extra packets that must be transmitted for diagnostic purposes. During this experiment all topologies are fault-free. For most benchmarks, the performance impact is lower than 3% and almost uniform over all topologies. An interesting exception is the *104.milc* benchmark evaluated in the mesh topology, which suffers from significant performance loss. This is because each core in that benchmark relies on very frequent and very long data transfers to one process mapped to the core on the top left corner of the mesh. This core, therefore, has a much more limited bandwidth, and thus the performance impact measured in this scenario is particularly pessimistic. Furthermore, for most benchmarks and topologies, the extra packets due to Cardio's diagnostic messages are less than 10%, and this overhead varies greatly with the benchmark considered: the impact is higher for applications with little inter-core communication (*e.g. 128.GAPgeofem*).

As the number of nodes in CMPs is expected to increase, it is important to evaluate the scalability of our solution. Assuming a CMP with n cores, the maximum number of messages periodically that must be periodically exchanged in Cardio is $2n^3$ and $2n^2(n-1)$ for a torus and a mesh respectively. In order to overcome a network failure, the number of messages exchanged in a torus is $8+4n^3$ and is $8+4n^2(n-1)$ for the mesh. Finally, reconfiguration time in Cardio is constant. This is because Cardio detects failures locally and handles hardware reconfigurations in software.

4.5.7 Energy vs. Performance Trade-off

We previously measured the responsiveness of our technique to hardware changes. Here, we focus on measuring the impact of our adaptive technique on the performance and the power consumption of a complex, specialized hardware system. With this goal, we considered a SoC that embeds specialized hardware targeting computer vision algorithms and evaluate the performance of MEVBench benchmarks. In this set of experiments we used the gem5 full-system simulator to measure system performance and McPat to estimate the power consumption of this design. We evaluated a range of SoCs, between 2 and 32 cores. The characteristics of the SoCs under evaluation are listed in Table 4.1. Cores were organized in a mesh connected through a network-on-chip. Cardio allows the system to dynamically activate each processor, while also providing the capability of tuning each core's voltage and frequency. Hence, users can dynamically adopt the configuration that best fits their needs.

In this evaluation, we considered computational epochs 10 million cycle long. In order to deploy Cardio's adaptable solution, each core independently suspends its execution at the completion of an epoch to perform basic routine checks. In these experiments we disabled all online reliability features to eliminate their impact in our results.

In this system, each core periodically reassesses its available features, collects data about its current utilization and state (frequency, performance, and temperature), and broadcasts this information to the rest of the chip. The arrival of these messages triggers each core to execute a simple procedure of the distributed resource manager, which collects all messages

Table 4.1 Hardware Configuration

Feature	Configuration
Pipeline	1GHz-400MHz , 32 bit out-of-order, 4-way superscalar
Functional units	4 integer units, floating point units, 1 SIMD unit, 1 set of hardware accelerators
Vector registers	64 32bit single precision registers
L1 cache	32k 2-way assoc. instr. and data (1ns)
L2 cache	1MB unified non-inclusive (12ns)
Cache coherency protocol	MOESI
System interconnect	128-bit NoC@ 1GHz
System memory	2GB LPDDR2
Instruction set	ARM-v7
Technology node	45nm

received during the execution of one epoch. The end of an epoch is determined individually by each core through an instruction counter, which interrupts all current operations to execute the service routines of the distributed resource manager. These routines use the information collected to construct an updated graph of the features available in the system, and, if needed, can trigger system reconfiguration and notify the OS to optimize application execution to match the new hardware setup. If no reconfiguration is necessary, the execution of such procedures only requires approximately 10,000 cycles.

We evaluated the impact of Cardio on the energy consumed by this system. In our experiments we target systems that fit within a constrained power budget, thus the additional energy needed by our adaptive solution increases almost linearly with the decreased performance. For the considered configurations, our experiments report that Cardio increases power consumption between 70 and 210 mW on average. These costs are minimal when we consider the benefits provided by our design.

Figure 4.10 reports our findings. For this experiment, we consider a test sequence executing all eight MEVBench benchmarks. The SoC under evaluation can activate up to 32 cores in order to reach the targeted performance improvement indicated on the X-axis – performance is measured as relative speedup compared against a baseline 2-core configuration. On the Y-axis we report the percentage of energy saved by an adaptable system deploying

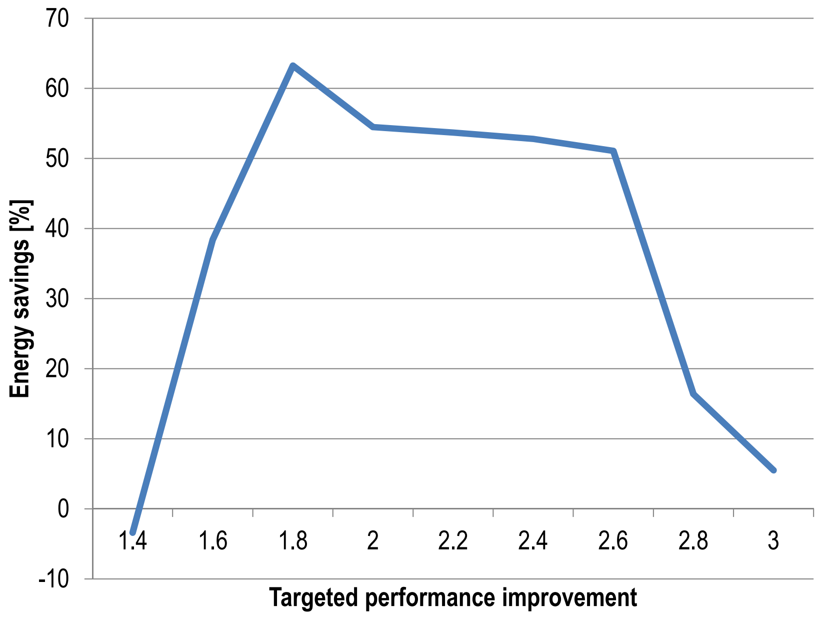


Figure 4.10 Energy savings obtainable thanks to our adaptable design. The plot reports on the Y-axis the reduction in energy consumed by an adaptable system deploying Cardio against a non-adaptable one. The graph shows in the X-axis the targeted performance improvements, measured as speedup compared against a baseline design with only 2 cores.

Cardio against a non-adaptable one. On one hand, we assume that the design augmented with Cardio can dynamically adapt to software demands, and, for each benchmark, activate the minimum number of cores necessary to achieve the targeted performance. On the other hand, the non-adaptable system must keep all cores active that are needed to achieve the targeted performance by the most demanding benchmark.

For a targeted performance improvement of $1.4\times$, we found that the system augmented with Cardio consumes slightly more energy. This is because all eight benchmarks require 4 cores to achieve this performance target. Therefore, our adaptable design cannot benefit the benchmark execution, while still requiring extra energy. However, as the targeted performance improvement increases, the benefits of our adaptable design become more evident. As shown in this plot, the energy savings made possible by our dynamic hardware resource assignment technique are vey significant, and peak at 63%. As the targeted performance improvement increases, energy savings achievable by our design decrease. This is because system performance saturates, and more and more benchmarks require the activations of all 32 cores in order to achieve the targeted performance, making less and less use of the adaptability features made available by Cardio.

4.5.8 Area Overhead

Compared to a typical CMP system with precomputed routing tables, Cardio requires the addition of a few hardware components throughout the interconnect. Considering the baseline design used in our evaluation, each network interface is enhanced with 10 buffers (Section 4.5.2) of 32 bytes each (size of 1 packet). In addition, we require 10 counters associated with the buffers to track timeouts, and each counter should be 20 bits wide to allow for a extended range of timeout values. Thus, the total storage overhead for each network interface is 345 bytes. Note that this last overhead is common to all solutions that require the ability to recover in-flight. Each router must store the IDs of nodes connected to each link: $4 * log(16) bits = 4 bytes$. Cardio's reconfigurable routing tables require $16 * log(4) bits = 2 bytes$. As already analyzed in the previous chapter, the area overhead introduced by the core self-test logic is approximately 1% [142]. Hardware additions for the routers, instead, increase their total area by roughly 12% (considering both reconfigurable routing tables and the BIST) [45].

4.6 Summary

This chapter introduced a solution that enables adaptability on hardware designs called Cardio, which consists of a low-cost technique that enables hardware adaptability. This technique can be leveraged to allow a chip to unlock significant improvements in computational efficiency, and promptly respond to environmental variations, changing application demands and hardware failures. This system-level solution is based on periodic exchanges of diagnostic messages among system components, which are handled by a distributed resource manager. Cardio can adapt to both design time and runtime hardware alterations, reconfiguring a hardware system in as little as 20,000 cycles.

While the previous chapter focused on solutions to provide our new architecture with reliability features, the technology presented here enables a design to be adaptable to hardware changes. Furthermore, this chapter also shows that the adaptability offered by this design can complement the reliability solution presented in Chapter 3, empowering a system to dynamically reconfigure around runtime hardware failures. So far, this dissertation has addressed two of the problems that we identified for future semiconductor technologies: the increasing number of hardware failures and the challenges connected with managing specialized functional units. In order to overcome the third challenge, lack of design modularity, the following chapter introduces a modular and distributed computer design. Lastly, the concepts and techniques developed throughout this chapter constitute an integral component of the holistic solution presented in Chapter 6.

Chapter 5
A Modular Computer Architecture

As detailed in the previous chapters, future semiconductor technologies are expected to allow further transistor miniaturization. This will enable computer chips to integrate tens of billions of transistors, thus empowering designers to include more and more features into a single silicon substrate. On one hand, this trend allows digital systems to improve performance while decreasing overall power consumption [23]. On the other hand, the lack of design modularity exacerbates the already high engineering costs [17, 18, 46, 97]. While the previous two chapters explored and evaluated solutions to make hardware designs reliable and adaptable, they do not mitigate the lack of design modularity that causes these high design costs.

In an effort to address this last challenge, the solution presented in this chapter prioritizes simplicity through the means of design modularity. Differently from prior designs, this architectural solution is characterized by a novel fully distributed control logic that makes it highly modular. Besides providing modularity, a property that can significantly reduce engineering costs, this novel execution paradigm can also boost reliability and adaptability, as we will discuss in Chapter 6.

5.1 Chapter Organization

Section 5.2 discusses the two main causes of lack of modularity in current processor designs: the presence of centralized control logic and the tight interconnection between hardware components. Moving forward, Section 5.3 describes the hardware organization of our new microarchitecture – Viper. This design uses a reconfigurable execution engine built from small, independent components guided by fully distributed control logic.

In Section 5.4 we utilize an example to demonstrate how our design dynamically reconfigures its computational fabric to execute programs. To complete the presentation of the execution model developed in this chapter, Section 5.5 explains how Viper handles exceptions and changes in the program flow. Further advantages of our execution model, possible optimizations, and a comparison of Viper against previously proposed architectures (reconfigurable CMPs, counterflow and dataflow machines) are discussed in Section 5.6. Finally, Section 5.7 presents the experimental evaluation of our microarchitecture and analyzes its performance, area, power consumption, and reliability.

5.2 Design Modularity

The lack of design modularity in current microprocessors is due to two reasons. First, complex and pervasive control logic is in charge of orchestrating the proper functioning of the entire processor [201]. This circuitry must manage all interactions among processor components, as well as handle external events. This logic is extremely complex, since design decisions must account for rare corner cases and infrequent interactions between asynchronous events, which can be difficult to foresee. Second, hardware components in modern microprocessors are tightly interdependent due to performance reasons. Such a design choice jeopardizes our ability to develop and test the functionality and behavior of individual components in isolation [46]. We must therefore address both the pervasive centralized control logic and the tight interconnection if we wish to develop a computer architecture that can be designed in a time- and cost- effective manner.

A modular system consists of distinct components that can connect, interact, or exchange information with each other through well-defined interfaces. An effective way to achieve modularity is to organize all components as self-contained entities that serve any request as an independent transaction. Non-modular designs, on the contrary, may not have clearly defined component interfaces, making it difficult to develop, test, and modify a component of the design independently from other components [132]. Furthermore, modular systems can be partitioned into a number of components that may be mixed and matched in various configurations [160]. Finally, these designs help reduce engineering costs because development of the different modules could evolve independently, incrementally, and in parallel [10, 128].

To this end, this thesis presents a new microarchitecture that decouples the functionality of a processor's parts from its control logic [147]. By removing the tight coupling between the different components of a processor, it becomes possible to build a modular and extendible microarchitecture. Our solution goes beyond organizing its hardware in separate modules with well-defined uniform interfaces. Indeed, a new execution paradigm is proposed for efficient execution on our modular architecture. A program is split into instruction bundles, each with a list of independent underlying tasks to be completed. Then, singular hardware components independently team up to complete these tasks. Our novel hardware organization and execution paradigm provides Viper with unprecedented design modularity.

5.3 Viper Hardware Organization

The design introduced in this chapter is based on a distributed execution engine that is dynamically configured to route instructions towards functioning hardware components. Viper is a service-oriented microarchitecture, where instructions are presented as clients that use hardware components to complete an ordered sequence of *services*. For instance, a sequence of such services for a simple `add` instruction - `add %al, [%ebx]` - could be: “fetch/decode instruction”, “retrieve value from registers”, “load memory value”, “add two operands”, “write the result back to a register” and, “compute the address of the next instruction”. From Viper’s perspective, an ISA consists of the set of services required by its instructions.

Instead of pushing instructions through paths defined at design time, as classic architectures do, Viper relies on a flexible fabric composed of hardware *clusters*. These clusters are loosely coupled via a reliable communication network to form a dynamic *execution engine*. Each cluster can accomplish one or more services and the number of clusters available in this system can change without affecting the rest of the system. Additionally, a cluster providing multiple services can be partially disabled and used solely for instructions that need its functioning services. Such a design is modular by construction, and each cluster is fully independent. In Viper, a program is always able to successfully execute as long as the working hardware clusters can, in aggregate, perform all the services required by its instructions.

Once an instruction is decoded, it is possible to know which remaining services it needs, and the instruction can be directed towards clusters available to provide those services. The set of clusters that contribute to the completion of an instruction form a *virtual pipeline*. Because clusters can be distributed across the chip, it may take many more clock cycles to transfer instruction information through these virtual pipelines than through a traditional hardwired pipeline. To mitigate this potential performance loss, Viper operates on larger collections of instructions called *bundles*, which, like basic blocks, typically end in control flow instructions. Bundles can successfully execute as long as at least one cluster can complete all their required services. Functional units within a cluster can service a bundle’s instructions out of order, and thus the maximum throughput achievable by a single cluster matches that of an out-of-order processor with an execution instruction window equal to the maximum number of instructions in a bundle.

In order to achieve design modularity, we partition an application execution flow into smaller bundles, which can be executed by the hardware modules as independent transactions. The advantage of this approach is that it enables hardware modules to be completely decoupled from one another. While bundles composed of single instructions would suffice for this purpose, the high reconfiguration overhead associated with assigning a virtual pipeline to each of these short bundles makes this choice prohibitive. Therefore, we must determine a feasible bundle size for our architecture.

Long bundles offer the opportunity to share virtual pipeline reconfiguration costs among numerous instructions. Unfortunately, including too many instructions on a single bundle is inefficient and wasteful. In the experiments presented in Chapter 3, we discovered that applications exercise hardware components in a bursty manner. Therefore, assigning all services needed to execute a long instruction sequence at once often results in very inefficient hardware utilization. Furthermore, since bundles commit atomically, the bundle size directly affects the responsiveness of the machine, as well as the amount of storage necessary to buffer speculative results. Lastly, an architecture deploying long execution bundles must include the rather complex logic for handling and correcting multiple misspeculated branches.

Given the above tradeoffs and constraints, four reasons guide our decision to partition dynamic execution in bundles that terminate with a control instruction (basic blocks). First, our preliminary studies performed with the Pin Dynamic Binary Instrumentation Tool [116] showed that the number of services required to execute all instructions within a single basic block is bound and can be used to efficiently assign subsets of functional units, as also shown in [216]. A second reason to organize execution at the basic block level is the fact that more than 95% of the dynamic basic blocks in typical applications are smaller than 16 instructions [147]. Therefore, a machine committing instructions at the basic-block level would be very reactive and only require moderate amount of memory to store speculative values. It is worth noting that applications may sporadically execute very long basic blocks. Since speculative storage is finite, our architecture will then split these long basic blocks into multiple bundles, each of which can fit into available speculative storage. Third, partitioning program execution into basic blocks allows our machine to take advantage of compiler optimizations and dataflow locality – previous research measured that when an instruction reads a value from the register file, there is a 92% chance that such value came from one of the last two producers of that value [187]. Fourth, an architecture working with instruction bundles that can only have a single entry- and exit-point can recover from branch missprediction simply by squashing all misspredicted bundles and restarting execution from the correct program counter.

In order to execute an application, Viper must be able to dynamically determine which hardware clusters will participate in each virtual pipeline. To avoid reliance on centralized control logic, Viper utilizes a collection of distributed and independent structures called *Bundle Scheduling Units (BSUs)*. Each BSU has some amount of storage that maintains information, such as the services that its bundle needs to complete, its virtual pipeline configuration, and the status of all operations performed on the bundle up to this point. This data is used to control the execution of a single bundle of instructions as it works its way to completion through its virtual pipeline. Every BSU also contains logic that is used to determine which hardware clusters will be used in its bundle's virtual pipeline. Clusters independently signal their ability to complete particular services to each BSU. Then, without consulting any centralized logic, the BSU then chooses which clusters will form the virtual pipeline that will service its bundle. This process is detailed in Section 5.4.

Viper's hardware can be logically partitioned into two parts:

1. A sea of redundant hardware clusters: hardware functional units connected through a homogenous communication medium. Each cluster can perform one or more of the services required to execute instructions in the ISA.

2. Bundle Scheduling Units: memory elements that contain the state of in-flight instruction bundles and store the data necessary to schedule and organize the hardware clusters that form a virtual pipeline. *A live BSU entry does not contain instructions or operands*, but only the information required to control the bundle's execution.

Figure 5.1 presents a simple Viper design organized in a mesh, where each colored service is replicated in multiple identical clusters. BSUs are connected to the sea of clusters through a crossbar, which allows each BSU to interact with all clusters in the execution engine. Clusters that need access to external modules are connected to them through dedicated links. For instance, clusters capable of "fetching instructions" are directly connected to the instruction cache, and the register file and load/store queues are placed near clusters that need fast access to these units. Finally, clusters that support the "write memory operations" service are connected to the load/store queue to allow stored values to be written to memory once the related bundles are committed. Depending on the layout of the hardware, note that such special links might not have uniform communication latency.

Viper is composable because the clusters are fully independent and only work on single instruction bundles, and thus individual clusters can be activated and deactivated without

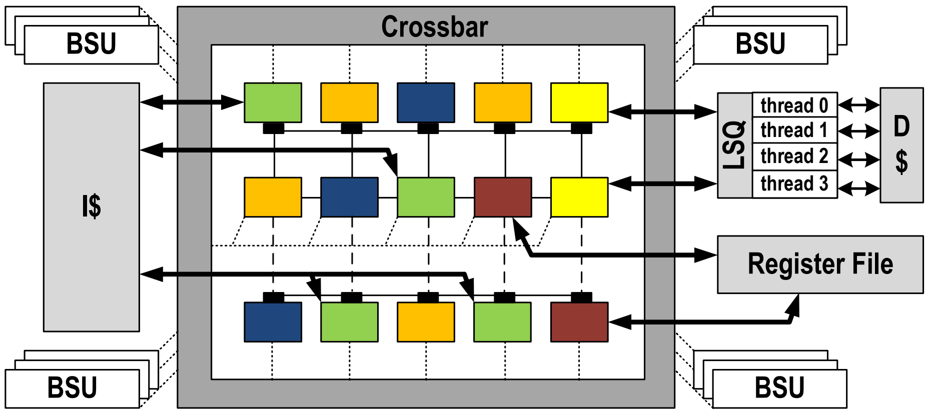


Figure 5.1 Organization of a Viper system with several redundant clusters that communicate through a mesh and that are connected to the BSUs through a crossbar. Some of the clusters, such as the ones capable of fetching instructions, have special connections to external hardware elements.

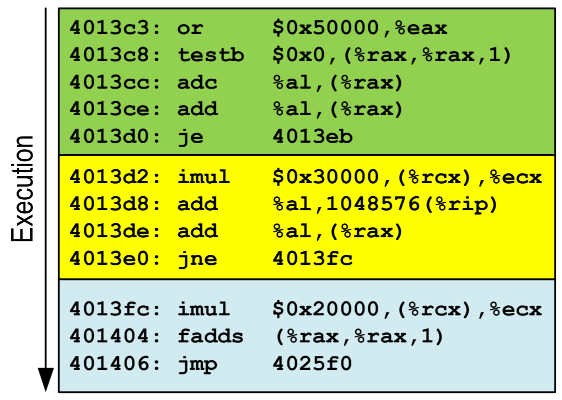
jeopardizing the design's ability to run software applications. Also, as long as all clusters and memory structures are redundant, Viper does not present any structural bottlenecks. In the following two sections we will illustrate how Viper executes a program using a running example. We first explain the steps necessary to execute a bundle of instructions and then detail how Viper handles special events such as branch mispredictions and exceptions.

5.4 Regular Execution in Viper

For the sake of simplicity, the Viper design used in our example provides only six services: "fetch", "decode", "rename", "execute", "commit", and "write-back and memory operations". Even though the services used in this example might resemble stages in a classic pipeline, it is important to stress that our architecture does not impose any constraint on how to partition services. This partition is an arbitrary design choice and should be guided by considering:

1. functionalities exposed by the ISA;
2. tasks accomplished by the underlying hardware;
3. degree of reconfigurability needed by the system.

The program stream is dynamically partitioned into bundles of instructions, which typically have basic block granularity. In Viper, each in-flight instruction bundle is associated with a live BSU entry. Figure 5.2 shows the three basic blocks that compose the simple program used throughout our example. As bundles are created in order, they are assigned a



```
4013c3: or      $0x50000,%eax
4013c8: testb   $0x0,(%rax,%rax,1)
4013cc: adc     %al,(%rax)
4013ce: add     %al,(%rax)
4013d0: je      4013eb
4013d2: imul    $0x30000,(%rcx),%ecx
4013d8: add     %al,1048576(%rip)
4013de: add     %al,(%rax)
4013e0: jne     4013fc
4013fc: imul    $0x20000,(%rcx),%ecx
401404: fadds   (%rax,%rax,1)
401406: jmp     4025f0
```

Figure 5.2 Simple program considered in our example.

sequential Bundle ID (BID). Each BID is paired with the thread ID of its process to form a unique bundle identifier throughout the entire machine.

In Figure 5.3.a we show the Viper design used in this example: it contains four redundant copies of the six different cluster types, each providing one of the services. In this section we illustrate how Viper can maintain correct program flow for the three bundles shown in Figure 5.2 (with BIDs 5, 6 and 7) and detail the generation of the virtual pipeline for the second instruction bundle - which starts and terminates with the instructions at addresses `0x4013d2` and `0x4013e0`, respectively. The color coding of the instruction bundles in Figure 5.3.a matches the hardware resources assigned to their execution and is maintained throughout all steps shown in Figure 5.3. New events and BSU updates are marked in red.

5.4.1 Bundle Creation

In this example, we assume that an instruction bundle (with BID 5) has already successfully determined the starting address of the next bundle. Therefore, program execution proceeds to the next basic block, which starts at address `0x4013d2`. Since a not-taken conditional branch concludes bundle 5, the "NPC" (Next Program Counter) field of its BSU stores the (correctly) predicted location, as shown in Figure 5.3.b. Because the PC of the next bundle is available, but no BSU has been assigned to it (the field "Next BSU" is empty), bundle 5's BSU assigns an available BSU entry to the following bundle, as shown in Figure 5.3.c. The mechanism for choosing the next BSU is described in Section 5.4.5.

When a bundle is first assigned to a BSU entry, the only two pieces of information available are: 1) the BSU entry number of the previous bundle and 2) the PC of the first instruction of the bundle. The former is needed because live BSU entries form a chain of in-flight bundles. This allows the system to track correct control flow and to commit bundles

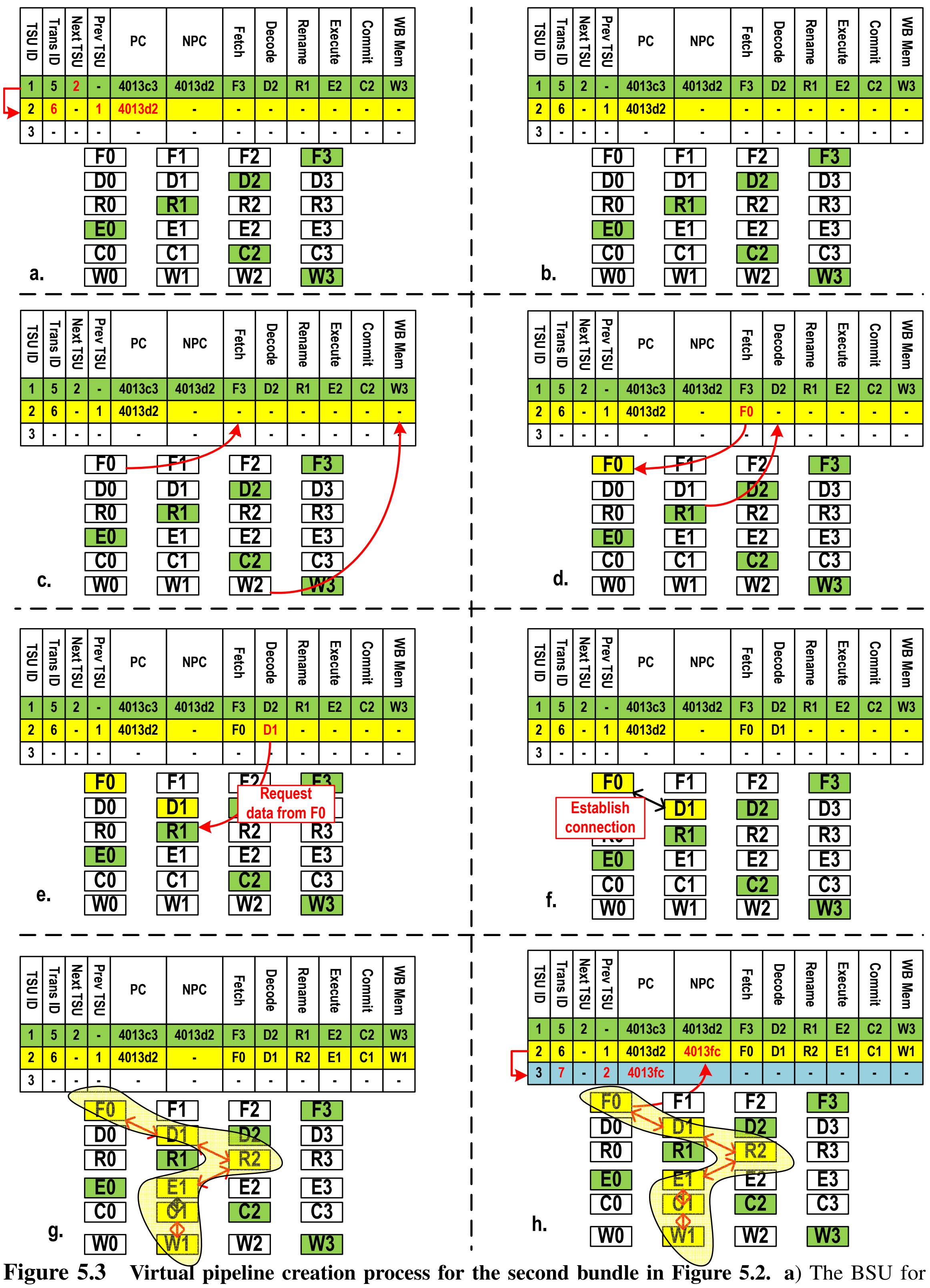


Figure 5.3 Virtual pipeline creation process for the second bundle in Figure 5.2. **a**) The BSU for bundle 5 creates the next bundle when the address of the following basic block becomes available in the next PC field. **b**) The new bundle is created in an available BSU. **c**) Functioning and available hardware clusters in the system propose their services to the new bundle. **d**) Cluster F0 is selected to become part of the new virtual pipeline. **e**) A subsequent proposal from D1 is accepted. Clusters are also notified by the BSU about the other clusters composing the virtual pipeline. **f**) The clusters are configured to establish communication paths. **g**) After the configuration, a virtual pipeline is formed. **h**) Finally, as F0 detects the last instruction in the bundle, it updates the BSU's NPC field, which allows the next bundle to begin.

in order. The latter information is needed by the fetch component, as we discuss shortly. Since a new set of clusters is needed to form the virtual pipeline for the new bundle, the newly assigned BSU marks all required services as unassigned. Figure 5.3.c shows that BSU 2 is assigned to keep track of a new bundle (with BID 6), and therefore the list of clusters assigned to its virtual pipeline is reset.

A similar process is also used to bootstrap Viper: when starting the system, a bundle with BID 0 is assigned to a BSU and its initial address is set to the reset address.

5.4.2 Virtual Pipeline Generation

A BSU entry assigned to control the execution of a new bundle is in charge of constructing a virtual pipeline capable of providing *at minimum* all services required by its instructions. Virtual pipeline generation consists of selecting which hardware clusters will collaborate in executing a bundle. Since using a centralized unit to perform this procedure would limit the modularity and thus the composability of our the system, Viper adopts a distributed mechanism to generate virtual pipelines. This mechanism is based on *service proposals*: clusters independently volunteer to execute services for a bundle in a live BSU.

Service Proposal

Several distributed mechanisms can be used to allow service proposals to reach the BSUs – solutions based on exchange of credits, token broadcasts, or service queues could all fit this purpose. For the sake of simplicity, and without losing generality, this example adopts a technique based on service queues. In such an implementation, a live BSU enrolls all needed services in queues accessible through a crossbar by both the hardware clusters and BSUs. BSUs requesting clusters are arranged in ascending order based on their BIDs, and service proposals from clusters are first forwarded to the oldest BID.

In our example, as the bundle with BID 6 has just been created, all six required services need to be assigned. Available clusters independently propose to service the BSUs that are enrolled in the service queue. Each cluster maintains a list of the virtual pipelines that have accepted its proposals, though a cluster can simultaneously be part of only a limited number of virtual pipelines.

We assume here that a cluster cannot propose its service to multiple BSUs: clusters F3, D2, R1, E0, C2, and W3 are already assigned to the previous bundle and therefore refrain from proposing their services to BSU 2. Nevertheless, any other available cluster (shown with a white background) can propose its services to the service queues, which redirect such proposals to needy BSUs. For instance, in Figure 5.3.d we show two clusters, F0 and W2, proposing their services to the bundle with BID 6. This may occur because clusters initiate the proposal negotiation independently, and therefore a BSU might receive multiple service proposals at the same time.

After submitting a service proposal, a hardware cluster changes its local status from "idle" to "pending" and waits for an award message from the BSU. A service proposal is not binding until a BSU notifies the proposing party; if no service award is received within a timeout period, the cluster considers its proposal rejected, and the service negotiation sequence is re-initiated.

Service Assignment

BSUs notify clusters accepted into the new virtual pipeline with an award message. In order to correctly build a new virtual pipeline, BSUs award clusters in the exact sequence as their services will be performed on the bundle. For instance, proposals for the "decode" service will not be accepted until the "fetch" service has been assigned to a cluster. In our example, the BSU cannot accept W2's proposal (shown in Figure 5.3.d) and cluster W2 therefore automatically returns to the "idle" state.

When BSU 2 chooses F0 to be included in its virtual pipeline, it records that this cluster will accomplish the "fetch" service for its bundle. Besides the notification that a proposal has been accepted, confirmation messages carry information needed by the clusters to perform their services. Such information consists of either data fields directly stored in the BSU or routing information needed to retrieve data from other clusters. The former case is shown in Figure 5.3.e: as the BSU sends a notification to F0 that its proposal has been accepted, it also forwards it the first memory address of the bundle with BID 6.

The other services are assigned to clusters in a similar fashion. Figure 5.3.e shows a service proposal sent by D1. Some of the services - such as "fetch" - are common to all bundles, while others can only be assigned once instructions in a bundle have been fetched and decoded. As the list of services is populated, functional hardware clusters are chosen in order to construct a complete virtual pipeline.

Configuring the Sea of Clusters

Clusters are dynamically selected to service bundles, and communication channels must be established between them to transfer information through the virtual pipeline. To perform this task, each cluster needs to know which clusters precede it in the virtual pipeline. In

Figure 5.3.f we show the BSU awarding its "decode" service to D1. This cluster is told which cluster will "fetch" the instruction bundle, in this case F0. D1 then establishes a connection with F0 through the reliable network, as shown in Figure 5.3.g.

All services are similarly assigned in an ordered fashion and, as the BSU service list is filled, the sea of clusters is configured to generate a complete virtual pipeline through the network, as shown in Figure 5.3.h. Viper can concurrently configure several independent active virtual pipelines, since the BSUs and execution clusters operate autonomously. Multiple virtual pipelines can work on a single program (as shown in our example), or can simultaneously execute multiple threads.

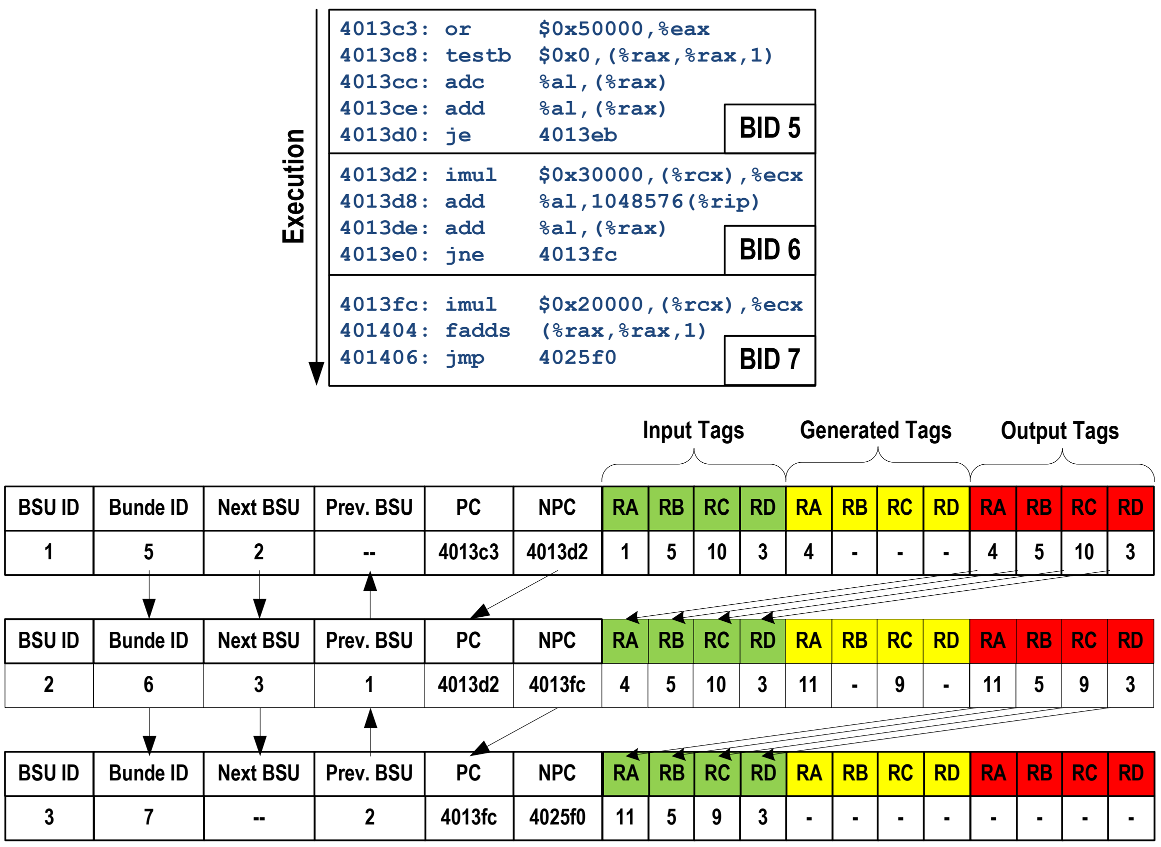
As faults accumulate in a device, a bundle might require a set of services that none of the execution clusters can provide alone. For instance, bundles 6 and 7 in our example can only execute on clusters that can service both the `add` and `mul` instructions. However, Viper can overcome this problem as long as at least one cluster can execute each one of the needed services. This case is addressed by canceling the execution of the unserviceable bundle and splitting it into multiple bundles, each consisting of a single instruction. While this technique reduces performance, as virtual pipeline creation overhead is not amortized across multiple instructions, it maximizes system availability by minimizing the set of services necessary to complete each bundle.

5.4.3 Operand Tag Generation

Viper does not enforce execution ordering on the different bundles, as long as 1) bundles belonging to the same thread commit sequentially and 2) cluster allocation avoids resource starvation. Thus, there is the opportunity for clusters to concurrently work on multiple bundles from the same program. For instance, in our example we show Viper concurrently executing BIDs 5 and 6.

Viper can improve performance by exploiting a program's ILP and capitalizing on the available hardware resources. However, this also creates inter-cluster data dependencies, as operands produced by clusters in one virtual pipeline might be needed by others. Viper utilizes operand tags to distribute values within the sea of clusters.

Adopting a centralized rename unit is not feasible, as this would undermine the composability of the system. Thus, we developed a BSU-based mechanism for generating and distributing tags to values produced by bundles. Because each bundle consists of an ordered sequence of instructions, only values live at a bundle's exit point can be used by following instructions. Thus, only live registers will have an associated tag: if multiple instructions in one bundle write to the same architectural register, only the last value that is produced is



```
4013c3: or     $0x50000,%eax
4013c8: testb  $0x0,(%rax,%rax,1)
4013cc: adc    %al,(%rax)
4013ce: add    %al,(%rax)
4013d0: je     4013eb
4013d2: imul   $0x30000,(%rcx),%ecx
4013d8: add    %al,1048576(%rip)
4013de: add    %al,(%rax)
4013e0: jne    4013fc
4013fc: imul   $0x20000,(%rcx),%ecx
401404: fadds  (%rax,%rax,1)
401406: jmp    4025f0
```

						Input Tags				Generated Tags				Output Tags			
BSU ID	Bunde ID	Next BSU	Prev. BSU	PC	NPC	RA	RB	RC	RD	RA	RB	RC	RD	RA	RB	RC	RD
1	5	2	--	4013c3	4013d2	1	5	10	3	4	-	-	-	4	5	10	3
2	6	3	1	4013d2	4013fc	4	5	10	3	11	-	9	-	11	5	9	3
3	7	--	2	4013fc	4025f0	11	5	9	3	-	-	-	-	-	-	-	-

Figure 5.4 Distributed rename table for an ISA containing four architectural registers - RA, RB, RC, RD. Tags assigned to the registers in previous bundles are used by the following to retrieve operands and solve data dependencies.

associated with a tag.

Each live BSU entry stores three tag versions for all the architectural registers in the ISA: “input”, “generated” and “output”. Compared to classical renaming schemes based on mapping architectural to physical registers, the “input” and “output” tags can be seen as two snapshots of a classic rename table: the first before and the second after the execution of the entire bundle.

In Figure 5.4, we illustrate how the tag generation and distribution process works for the three bundles used in our example for an ISA containing four architectural registers: RA, RB, RC, and RD. The first tags, “input tags,” are used to allow a bundle to retrieve its input operands: in our example in Figure 5.4, the cluster fetching register values for the instructions in the bundle with BID 5 will use tag “1” to retrieve the value of register RA from the physical register file, tag “5” to retrieve RB, and so forth.

The second set of tags, called “generated tags,” is used only if instructions in a bundle write to architected registers. In our example, two instructions update RA: `adc %al,(%rax)` first and `add %al,(%rax)` later. Because both of these instructions update the same register, only the operand computed by the last operation - `add %al,(%rax)` - needs to generate a new tag, 4. Tag generation could either be accomplished by the BSU or it can be serviced by the clusters. Tag generation for two sequential bundles must be serialized to guarantee program semantics. Solutions to allow our machine to univocally distinguish tags associated to operands are:

1. maintaining the set of free tags in a memory array protected by ECC;
2. piggybacking a list of available tags in the live BSUs;
3. generating a large number of tags with limited lifetime.

Finally, "output tags" associated with a bundle are used as "input tags" by the subsequent bundle. Output tags are produced by overwriting the input tags with any newly generated ones. Output tags of one bundle are provided as input tags of the next, as shown in Figure 5.4. Tags are generated wherever renaming is required – for all instructions, to enable out-of-order execution of the instructions within a bundle. However, the tag generator only reports to the BSU the tags for the operands that are live-out at the end of the bundle.

5.4.4 Bundle Execution

After an instruction is associated with an output tag, it can move to the functional units that will execute it. Instructions requiring only inputs produced by instructions within the same bundle will execute in the local modules, which are guaranteed by construction to contain all functional units required by the entire bundle. While the tag generation logic is very similar to the renaming logic of a processor such as the R10000 [213], the out-of-order execution window in Viper differs from a regular out-of-order machine. When a single bundle is running on Viper, the execution units are effectively working with a maximum execution window equal to the number of instructions in the bundle. However, things change significantly when multiple bundles from the same application execute at the same time on different functional units. Each cluster executing a bundle is effectively working with a maximum execution window equal to the number of instructions in its bundle. When multiple clusters are executing bundles from the same application, the effective execution window is equal to the sum of all instructions in flight for all bundles. The amount of ILP that can be extracted is limited by the total number and type of functional units within the executing clusters. Note that, since instructions from multiple bundles can have data-dependencies, an execution cluster may be required to forward an operand it has produced to an instruction executing in a different cluster.

Several solutions have been proposed to correctly deliver operands in a distributed execution engine. Some dataflow architectures solved this issue by broadcasting all produced operands to the entire machine. Other solutions, such as RAW [192], rely on compiler support to distribute operands to the different hardware modules. In Viper, we can use the BSUs to overcome this issue without relying on software support. In fact, the set of input tags in a BSU is annotated with the hardware cluster that produces the operand, thus avoiding the need to broadcast operand requests to the entire sea of clusters. A functional unit working on a bundle can retrieve this information and request the needed input operands by tag from both the register file and the clusters that executed the previous bundles. Finally, functional units assigned to execute a bundle request all output operands produced by the previous bundle in advance, since output tags of a previous bundle are known before the next one begins executing.

5.4.5 Bundle Termination

A bundle can terminate only if two conditions are met:

1. all clusters assigned to its virtual pipeline finish servicing its instructions;
2. all preceding bundles belonging to the same thread have already terminated.

If both these conditions are met, a bundle's instructions are then checked for exceptions. If no exceptions are detected, the bundle is terminated atomically, its instructions update the architectural register file and its "store" operations are committed to memory. In the example in Figure 5.3.h we show two bundles in-flight (with BID 5 and 6). As instructions need to commit in program order, bundle 6 is not allowed to terminate before its predecessor, bundle 5. Bundle 5, on the other hand, is the oldest bundle in flight (the "Prev BSU" field is empty) and can terminate as soon as all the clusters in its virtual pipeline complete their services.

It is worth noting that Viper does not need a reorder buffer, as program order is enforced by committing bundles in sequential order maintained by the linked-list formed by the BSUs.

Memory Operations

Our design includes one load and store queue for each supported thread. This structure is useful to enforce the correct order of memory accesses and to detect conflicts between load and store operations. Each entry in the load queue keeps track of the cluster that generated the memory requests, so as to deliver the data retrieved from memory to the correct destination. Each entry in the store buffer also maintains information about the bundle that originated the store instruction, as memory updates are committed or canceled at the bundle granularity. Before terminating, a bundle with pending store instructions signals that its memory operations can be committed to the store buffer associated with its thread. This signal will cause all store instructions in the bundle to update the memory state in program order.

Since multiple bundles from the same program can execute in parallel, the load and store queues might receive misordered memory requests. This could prove problematic, as the forwarding logic in the load and store buffer might mistakenly: 1) forward values to load instructions that are produced by younger stores or 2) receive a sequence of stores that does not reflect the program order. Because the memory queue cannot dynamically address these issues, they are resolved by clearing all entries in the thread's load and store queue and canceling the execution of the conflicting bundles. To ensure forward progress, the oldest canceled bundle is split into multiple bundles, each containing a single instruction. This replay mechanism is also used to handle exceptional events, such as page faults, and it is presented in Section 5.5.

Another solution to address this issue is proposed in [74]. This design includes a local store in the issue stage and a speculative store buffer in the execute/memory stage allowing delayed release of memory store operations. This store buffer also serves the purpose of keeping speculative stores from corrupting memory state. The store queue in the issue stage tabulates the outstanding store instructions, and their present states. These two structures are kept synchronized through a number of signals, and together they ensure that only stores on the correct path of execution update the memory [74].

Finally, a more radical solution to solving ambiguous memory dependencies on distributed systems is speculative versioning caches (SVC) [66]. This solution uses distributed caches to reduce latency and increase bandwidth. SVC conceptually unifies cache coherence and speculative versioning by using an organization similar to snooping bus-based coherent caches. SVC provides hardware support to break ambiguous memory dependences, thus solving the issue connected with aggressive parallel execution of sequential programs.

Managing Bundle Sequence

Each live BSU maintains starting addresses for both its bundle and the one immediately following. This latter value is provided by the clusters performing the "fetch" service, as they can recognize the end of a bundle when fetching a control flow instruction such as "jump". Such clusters communicate the starting address of the next bundle back to their BSU, as shown in Figure 5.3.g: even before bundle 6 terminates, F0 can predict the starting address of the following basic block - `0x4013fc` in our example - updating the "NPC" field of the BSU with this address. With this, the BSU can generate a new bundle (in our example with BID 7), and continue program execution.

BSU assignment is performed using the same mechanism that we deployed for cluster service negotiation. An active BSU needing to initiate a new bundle requests a new "initiate a new bundle" service, to the service negotiation system. Idle BSUs will then propose to accomplish this task, as previously detailed in Section 5.4.2.

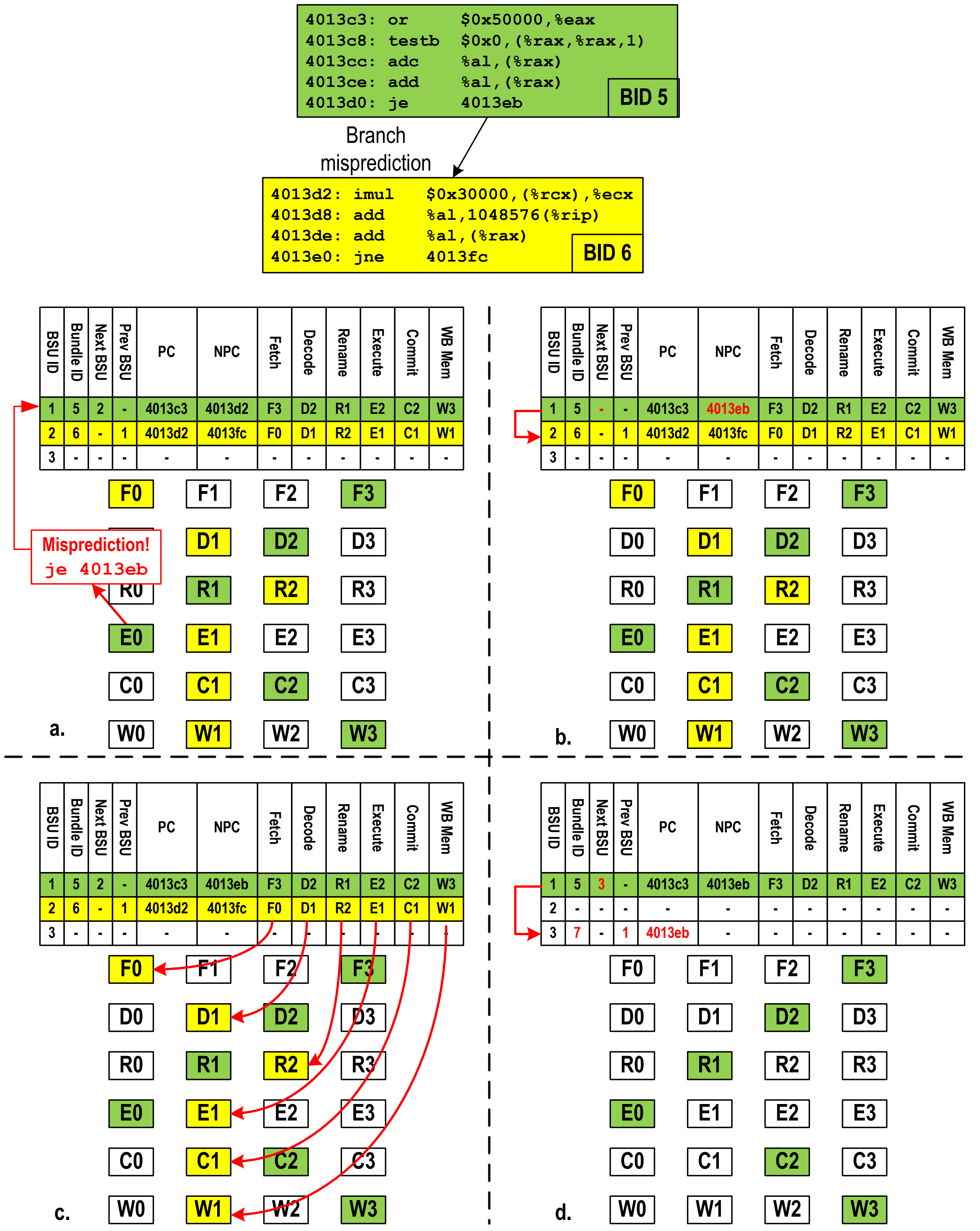
5.5 Handling Exceptional Events

Due to the fact that hardware clusters are fully decoupled, our architecture cannot rely on classic techniques - such as broadcasts of clear signals - to flush stale instructions from the system and correct an erroneous program flow. In this section, we specify how Viper can resolve such events through its BSUs.

5.5.1 Mispredicted Branches

Most processors require several cycles to resolve the target of an instruction that modifies the control flow. This delay might cause the system to start processing instructions from an incorrect execution path: these instructions need to be flushed as soon as a control flow misprediction is detected.

Similarly, Viper needs to cancel the execution of bundles generated by misspeculated program paths. We use the example shown in Figure 5.5 to illustrate how our architecture can tackle such events. We assume that bundle 6 is mistakenly predicted to follow bundle 5, and that all services required by both virtual pipelines are already assigned to clusters in the execution engine - Figure 5.5.a. Once a cluster in a virtual pipeline resolves a branch target, it reports the computed address to its BSU. This case is shown in Figure 5.5.a, where cluster E0 reports to bundle 5 that the correct initial address of the next basic block is `0x4013eb`. If this target address does not match the one stored in the NPC field of the BSU, a bundle misprediction is detected - Figure 5.5.b. All bundles generated from a mispredicted address - in our example bundle 6 starting from address `0x4013d2` - are canceled. The BSU notifies the clusters composing virtual pipelines of canceled bundles, so they can stop their work



```
4013c3: or     $0x50000,%eax
4013c8: testb  $0x0,(%rax,%rax,1)
4013cc: adc    %al,(%rax)
4013ce: add    %al,(%rax)
4013d0: je     4013eb
```

```
4013d2: imul   $0x30000,(%rcx),%ecx
4013d8: add    %al,1048576(%rip)
4013de: add    %al,(%rax)
4013e0: jne    4013fc
```

a.

BSU ID	Bundle ID	Next BSU	Prev BSU	PC	NPC	Fetch	Decode	Rename	Execute	Commit	WB Mem
1	5	2	-	4013c3	4013d2	F3	D2	R1	E2	C2	W3
2	6	-	1	4013d2	4013fc	F0	D1	R2	E1	C1	W1
3	-	-	-	-	-	-	-	-	-	-	-

b.

BSU ID	Bundle ID	Next BSU	Prev BSU	PC	NPC	Fetch	Decode	Rename	Execute	Commit	WB Mem
1	5	-	-	4013c3	4013eb	F3	D2	R1	E2	C2	W3
2	6	-	1	4013d2	4013fc	F0	D1	R2	E1	C1	W1
3	-	-	-	-	-	-	-	-	-	-	-

c.

BSU ID	Bundle ID	Next BSU	Prev BSU	PC	NPC	Fetch	Decode	Rename	Execute	Commit	WB Mem
1	5	2	-	4013c3	4013eb	F3	D2	R1	E2	C2	W3
2	6	-	1	4013d2	4013fc	F0	D1	R2	E1	C1	W1
3	-	-	-	-	-	-	-	-	-	-	-

d.

BSU ID	Bundle ID	Next BSU	Prev BSU	PC	NPC	Fetch	Decode	Rename	Execute	Commit	WB Mem
1	5	3	-	4013c3	4013eb	F3	D2	R1	E2	C2	W3
2	-	-	-	-	-	-	-	-	-	-	-
3	7	-	1	4013eb	-	-	-	-	-	-	-

Figure 5.5 Exception and branch misprediction handling in Viper. The cluster that detects a branch misprediction updates the program counter of its virtual pipeline (**a**), which consequently clears the state of the following bundles (**b**) and resets the virtual pipelines for the misspeculated bundle (**c**). Finally, program flow is steered towards the correct execution path (**d**).

and clear their states - Figure 5.5.c. Finally, the most recent non-speculative bundle recovers program execution, creating a new instruction bundle starting at the correct basic block address - Figure 5.5.d.

5.5.2 Exception and Trap Handling

Interrupts, exceptions, traps, and page faults must be handled with particular attention. Without modifying the bundle termination procedure, these events can cause the system to deadlock. For instance, an instruction triggering a page fault might prevent its entire bundle from terminating. To overcome this issue, a bundle affected by one or more of these special events is canceled and split into multiple bundles, each including a single instruction from the original basic block. The bundle containing the faulty instruction will then guide program execution to the correct software handler. Other cases where bundles must contain only a single instruction are system calls and uncacheable memory accesses.

5.6 Discussion

The modular and composable execution model proposed in this chapter offers several advantages compared to classic micro-architectures and can be enhanced through numerous features. This section briefly presents other benefits of our distributed-control architecture, lists possible enhancements to our design, and compares Viper with other exotic designs.

5.6.1 Additional Advantages

Viper's distributed and decoupled execution engine enables several technologies can decrease costs, improve reliability, limit power consumption, and enable massive parallelism. First, production yield could be boosted thanks to the fact that chips that have enough functional clusters to support, in aggregate, all instructions in the ISA are still marketable. Second, as each hardware cluster is fully independent, fault isolation and online testing is greatly simplified: clusters can be singularly disabled without pausing program execution. Third, as fine-grained clock scaling for small portions of a digital system becomes feasible, clusters can be clocked at different frequencies depending on their physical characteristics. This would allow fine voltage and frequency tuning to mitigate the effects of process vulnerability [47]. Finally, our architecture provides the advantage of supporting an arbitrary number of simultaneous threads, dependent only on the number of BSU entries available in the machine. As all the information needed to control program execution is already bundle-based (PC, renamed registers, thread-id, *etc.*), no further effort is needed to manage multiple threads.

5.6.2 Possible Optimizations

Compared with execution on classic microarchitectures, programs running on Viper incur the cost of dynamically generating virtual pipelines. A series of optimizations can be adopted to reduce such overhead:

- Avoid virtual pipeline generation for complete seas of clusters: if no clusters are disabled in a sea of clusters, BSUs could automatically assign a default set of clusters to bundles avoiding service proposal overheads.
- Early virtual pipeline generation: service proposal negotiation can be performed ahead of time, for instance while clusters are already working on previous bundles, thus reducing their idle time.
- Non-blocking instruction migration: instructions can be transferred from a cluster to the next in the virtual pipeline as soon as they complete, instead of waiting for the whole bundle to be serviced.
- Trace caches and multiple bundle prediction: as Viper works at the basic block granularity, our architecture can improve ILP through trace caches and sophisticated look-ahead predictions [156, 165].
- Loop detectors: so far we have assumed that all bundles need to accomplish some services such as "fetch instructions" and "decode". Loop detectors could recognize tight program loops and replicate bundles reducing reliance on some services [195].
- Fast exception handling: to handle program exceptions, the model introduced in this chapter clears all bundles in-flight and restarts execution in a safe but slow serial fashion. The ability to dynamically split in flight bundles would allow our system to clear only the instructions that need to be re-executed, without forcing an entire bundle to re-execute.

5.6.3 Comparison to Previous Work

In this section we compare Viper against three other design approaches: configurable CMPs, counterflow architectures, and dataflow machines.

Configurable Chip-Multiprocessors

Several previous works focused on developing configurable chip-multiprocessors that could reconfigure their hardware in order to maintain performance. In this section we will discuss three types of configurable CMPs: the first strives to maintain ISA compliance at the chip level, the second relies on reconfiguring fully instruction-set-architecture (ISA) compliant modules in order to improve performance, and StageNet, which uses small hardware modules to achieve chip-level ISA compliance and improve performance.

Chip-level instruction-set compliance – The authors of core salvaging observed that even if some individual cores in a CMP cannot execute certain operations, the natural cross-core redundancy may still make the whole chip ISA compliant. In this design, applications executing on a core that cannot execute certain operations will then migrate to a different one that can execute them [149]. Necromancer extends this idea by exploiting partially functioning cores to improve system throughput by supplying hints regarding high-level program behavior. CMP cores in this design are grouped into sets, each of which shares a lightweight core that can be substantially accelerated using these execution hints from crippled cores [6]. Similar to these works, Viper enables seamless dynamic thread migration to maintain performance within the constraints of available resources. Although both designs share with Viper the concept of chip-level ISA compliance, our design allows for finer reconfiguration granularity and enables automatic dynamic resource allocation.

Core composition – Two works, Composable Lightweight Processors (CLPs) and Core Fusion, proposed flexible chip-multiprocessor architectures composed of composable cores. These designs share the same goal: effectively adapting parallel resources to varying number of threads. CLPs allow simple, low-power cores to be aggregated together dynamically, forming larger, more powerful single-threaded processors [100]. Like CLPs, Core Fusion allows multiple dynamically allocated processors to share a single contiguous instruction window [93]. This is achieved through a reconfigurable chip multiprocessor architecture where groups of fundamentally independent cores can dynamically morph into a larger CPU, or they can be used as distinct processing elements, as needed at run time by applications. The advantage of Core Fusion over CLPs is that it exploits conventional RISC or CISC ISAs. In order to provide this advantage, some structures (e.g. register renaming) must be physically shared, limiting its scalability to 8-wide issue. Instead, CLPs share no physical resources, so they can form cores that can issue up to 64 instructions in one cycle. Unfortunately, CLPs rely on a non-standard EDGE ISA to achieve this modularity. These solutions and Viper share the characteristic that more hardware components can be aggregated in order to improve single-thread performance. However, since both these solutions are designed to improve performance, it is not clear how they can improve the reliability of a system.

Fine-grained reconfiguration – StageNet is one of the first distributed architectures targeting reliability as a key design criterion. The authors propose the use of a reconfigurable fabric to connect multiple hardware modules, each of which can perform the tasks associated with one stage of a typical in-order pipelined processor [72, 73, 74]. Hardware stages are partitioned into islands to allow the solution to scale, but this constrains the system's connectivity and reconfigurability. This design relies on interconnection flexibility, micro architectural innovations, and compiler directed instruction steering to merge pipeline resources for high single-thread performance. The same flexibility enables it to route around broken components, achieving sub-core level defect isolation. Together, the resulting fabric consists of a pool of pipeline stage-level resources that can be allocated for accelerating single-thread performance, computing throughput, or tolerating failures. Unlike StageNet, Viper allows out-of-order execution and empowers engineers to decide the granularity of the reconfigurable modules at design time; StageNet reconfiguration granularity is always bounded to pipeline stages of an in-order processor core.

Counterflow Architectures

This design was first introduced by Sproull *et al.* in 1994 to demonstrate the opportunities of asynchronous execution and modular design [115, 179]. Here, we briefly examine the functioning of an optimized non-stalling counterflow machine [9, 123]. This design builds upon the basis of the original counterflow architecture [179] to develop a substantially faster and more scalable machine. Both the original and the optimized design rely on the same modular control algorithm and on local, asynchronous clocks.

Counterflow processors present a competitive alternative to the approach adopted by out-of-order superscalar designs. Classic high performance pipelines schedule instructions and operands through complex control logic. Instead, the execution engine of a counterflow machine is composed of two parallel and opposing pipelines, one pushing operands in one direction and a second pushing incomplete instructions in the opposite direction. The basic intuition is that, as incomplete instructions flow in one direction, they can match and copy the value of the operands passing in the opposite direction. This technique simplifies a processor design, as it allows out-of-order execution without requiring a common data bus.

Unlike classic out-of-order superscalar pipelines, counterflow stages do not perform any operations. Instead, these pipeline stages are responsible solely for scheduling instructions to the functional units and resolving data dependencies. As a result of these design choices, components in a counterflow architectures only need local controls and do not require complex centralized control logic. Furthermore, the structure of this machine is very regular and modular, therefore easing design verification. Hence Viper and the Counterflow architecture share some of the same objectives.

However, the differences between this type of machines and Viper are significant. First, counterflow designs only focus on restructuring the execution engine of a processor, maintaining an unaltered front-end, reorder-buffer, and register file. Second, while this architecture is modular, communication patterns between its modules must be defined at design time. Finally, hardware components in this system are arranged in one dimension, and therefore scaling up the number hardware components increases the length of its pipelines and consequently worsens average instruction execution latency.

Dataflow Machines

Dataflow machines do not obey the serial execution model adopted by classic control-flow architectures [7, 197]. In fact, an instruction in this system executes as soon as all of its inputs become available. Dataflow machines offer significant advantages over regular control flow architectures. Since they are completely data-driven, their performance is only inhibited by true data dependencies. Furthermore, components in a dataflow system are stateless, and therefore side-effect free. These characteristics make this architecture highly modular. Although these machines had historically limited commercial success, many researchers have developed and analyzed these architectures. For instance, Multiscalar [174], RAW [191], TRIPS [27], and Wavescalar [187] are all examples of recent projects inspired by this execution model. Since Viper's hardware clusters are loosely coupled to form a dynamically reconfigurable fabric of hardware clusters, our hardware organization might appear similar to that of dataflow machines.

Unfortunately, dataflow architectures are characterized by a series of shortcomings that tend to overshadow their advantages. First, they typically rely on dedicated compilers to extract operand and data dependencies and map them into the hardware. Such compilers are effective only on a very narrow set of applications and legacy code cannot take advantage of the vast amount of logic available in these machines. Programs running in any of these designs require binaries annotated with architectural-specific information and thus cannot execute legacy binaries. Among these designs, Wavescalar [187] is the only one that could dynamically avoid assigning instructions to faulty processing elements. However, it still relies upon a centralized hash table to manage instruction scheduling. Second, there are several technical obstacles in the way of designing efficient data-flow machines due to the need to dispatch, distribute, and match tokens for both instructions and operands. Another problem with this execution model is its inefficiency in handling data structures. Since operands exchanged in this machine are scalar values expressed through tokens, efficiently representing large data structures represents a serious challenge.

TRIPS is an interesting example of dataflow machine that supports one main characteristic: direct instruction communication [27], where the functional units deliver the result of an instruction directly as an input to the consumer instructions, rather than writing it back to a shared namespace, such as a register file. Using this direct communication from producers to consumers, instructions execute in dataflow order, with each instruction executing as soon as its inputs become available. In comparison, Viper uses its BSUs to keep track of the tags assigned to instruction bundles. The advantages of TRIPS include higher exposed concurrency and more efficient execution. Unfortunately, TRIPS uses a very peculiar instruction set, and applications must be recompiled in order to execute on this architecture. Viper, on the other hand, does not modify the interface between hardware and software, and therefore can execute any legacy application.

5.7 Evaluation

We simulated a Viper implementation that uses the x86-64 ISA to evaluate Viper's performance. We compared Viper against a similarly sized CMP design comprised of 2-wide out-of-order cores. We choose to compare against out-of-order cores because they are the natural successors to in-order CMP cores currently used in many-core machines [22, 166], which currently adopt in-order processors [15, 164].

We first studied the effects of bundle size on Viper's performance and follow this with an evaluation of Viper's area overhead. Next, we compared the power consumption of Viper against traditional CMPs comprised of out-of-order cores. After, we evaluated single-threaded performance of workloads from the SPEC CPU2006 benchmark suite [83]. Finally, we briefly discussed the reliability of our design.

Processor characteristics		x86-64ISA 1GHz 64-bit 2-way superscalar, 32nm technology node
Pipeline	Fetch	3 cycles
	Decode	3 cycles
	Tag Generation	3 cycles
	Execute	2 integer ALUs, 2 FPUs, variable latency (min 1 cycles)
	Commit and load/store logic	2 cycles
L1 caches		32k 8-way assoc. instr. and data, 1ns latency
Inter-cluster switches		128×128 crossbar
Communication buffers		16 instructions
Cache coherency protocol		MOESI
System interconnect		128-bit Bus@ 1GHz with fast snoop unit
System memory		512MB, 30ns latency

Table 5.1 Viper Configuration

5.7.1 Hardware Model

The Viper architecture we evaluated in this work offers only six services: “fetch”, “decode”, “tag generation”, “execute”, “commit” and “write to memory”. Our modeled Viper design includes five types of clusters. The first four services are each executed by four different kinds of clusters, each capable of performing a single service. The fifth type of cluster can accomplish both the “commit” and the “write to memory” services.

The sea of hardware clusters is organized in a mesh connected through 128-bit wide links. Routes in the interconnect can be warmed up before transmitting the data packets, so we modeled the cluster-to-cluster latency as one extra cycle of delay per hop, with data transmission between clusters fully pipelined [122]. Communication between the BSUs and the clusters requires very little bandwidth, since it is limited to a few control bits. For these connections, we used a crossbar with a latency of 4 cycles [206].

The Viper design we modeled adopts a number optimizations to improve efficiency and utilization. Among the most significant are:

1. Early virtual pipeline generation: service proposal negotiation can be performed ahead of time; for instance, while clusters are working on previous bundles, thus reducing their idle time.

2. Non-blocking instruction migration: instructions can be transferred from a cluster to the next in the virtual pipeline as soon as they complete, instead of waiting for the whole bundle to be serviced.

Both Viper and the out-of-order core are modeled to fetch, decode, execute and commit up to two instructions per cycle and are clocked at 1 GHz. Each core contains 2 integer pipelines, 2 FP units, 1 load/store unit and 32KB of L1D and L1I. In order to fairly compare Viper's performance against classic processors, each of the "execute" clusters in our design has functional units identical to those in the out-of-order core. Finally, in our performance evaluation, we compare Viper against a baseline OoO processor with a comparable instruction window - 32 ROB entries and 5 RS entries per functional unit. Both the out-of-order machine and Viper's "execute" clusters can issue up to 5 instructions per cycle to the functional units. The most important hardware characteristics of the modeled system are listed in Table 5.1.

5.7.2 Simulation Infrastructure

We developed a microarchitectural model of our design in the gem5 simulator [19], relying on full timing simulations in system-call emulation mode. The Viper system that is modeled is based on the C++ implementation of the out-of-order core provided by the original gem5 distribution. Building on this model, we organized the system in fully decoupled clusters and augmented it with the required communication infrastructures (inter-cluster mesh and crossbar) and the BSUs. Timing models for all hardware components have been modified to better match the deeper pipelines typical of modern CISC processors [78]. The number of cycles for each logical stage are listed in Table 5.1, and sum to a minimum of 12 cycles. The amount of storage needed for each BSU entry is reported in Table 5.2, and adds up to 123 bytes. In order to measure both the area and power of our design we used a customized version of McPAT 0.8, a tool which provides integrated power, area, and timing models and enables comprehensive architectural exploration for multicore and manycore processors [108].

5.7.3 Design Choices

We first analyzed the benchmarks' performance as a function of the number of available BSU entries. Because these entries hold the dynamic state of bundles that are running in parallel, their number directly affects the maximum ILP achievable by the execution engine. We found that single-threaded performance reaches a plateau for a system composed of 4 BSU entries for every full set of hardware clusters. The minimum number of operational entries needed by our proposed microarchitecture is 2. However, we estimated that a Viper system with only 2 entries operates 24.4% slower on average than one with 4. Since the system analyzed in this section supports up to four concurrent threads, a model with 16 BSUs is used for all further experiments.

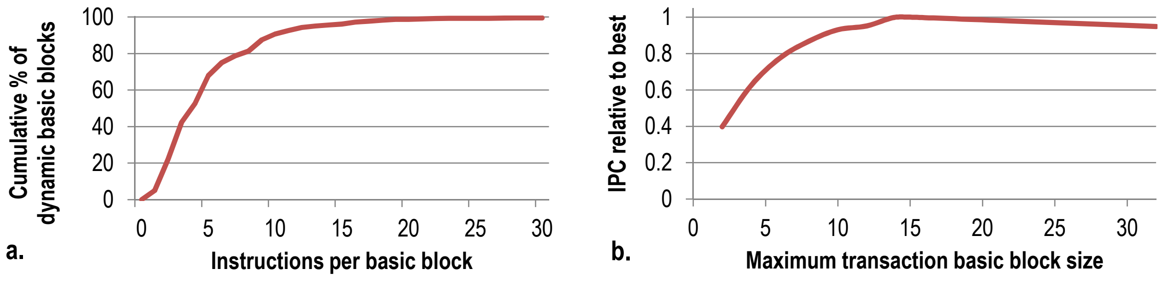


Figure 5.6 Effects of basic block size on Viper's performance. **a**) Cumulative distribution of the basic block size in our benchmarks. **b**) Sensitivity study on the maximum number of instructions allowed in Viper's bundles.

Another parameter to select is the maximum number of instructions allowed in a single bundle. On one hand, bundles with a large number of instructions have the potential to depend less on operands produced by clusters in other virtual pipelines. This can provide a significant advantage, as it reduces instruction reliance on long latency inter-cluster operand requests. On the other hand, partitioning program execution in smaller bundles allows more clusters to execute instructions concurrently. This leads to a performance tradeoff, which we analyzed through Pin [116] by gathering statistics on the distribution of basic block sizes in our benchmarks. Our finding are reported in Figure 5.6.a. More than 95% of the dynamic basic blocks in our applications are smaller than 16 instructions. We performed a sensitivity study on how this parameter affects Viper's performance, and report our results in Figure 5.6.b. A slight performance slowdown is shown for bundles larger than 16 instructions, mostly due to the higher probability of conflicts between instructions in the load/store queue and to the higher costs of recovering from canceled bundles.

5.7.4 Area

In this section we estimate the area of the hardware overhead due to our novel architecture, and detailed results about our estimations are reported in Table 5.3. All area estimates are reported in mm^2 and are computed assuming a system built in 32nm technology. Beyond their functional logic, each type of Viper cluster includes buffers for inter-cluster communication (input and output buffers) and the circuitry necessary to manage the virtual pipeline configurations. All these components and their connections are shown in Figure 5.7. Clusters exchange information about instructions and operands through a homogenous data network composed of routers, and all clusters and BSUs communicate via a single control network router. If we consider a Viper architecture comprising four copies of all clusters, four register files, and four load store units, the total area occupied by such a system sums to 66.552 mm^2. Through McPat we measured that a similarly sized classic CMP design composed of four out-of-order cores occupies an area of 61.998 mm^2. Therefore, our design occupies only 7.34% more area than a similarly sized traditional CMP. While these figures consider a number of factors that were previously neglected, they confirm previous area overhead estimations (7.2%) [147].

It is worth briefly discussing our results and comparing our design against a similar solution based on dynamic reconfiguration, StageNet [73]. A StageNet design composed of five in-order cores enhanced with reconfigurable logic and instruction buffers is measured to occupy an extra 16.3% of silicon area. Several factors contribute to the relatively lower area overhead of our design. First, while we add a number of new components to a classic design, we also remove a significant number of large, centralized, and multi-ported modules, such as the ROB and the RAT. For instance, McPat reports that the ROB of our baseline system measures 0.98535 mm^2, or about 6.3% of the area of one out-of-order core (15.4995 mm^2). Furthermore, area cost for the StageNet architecture is relative to a simple 5-stage in-order core, which logic typically employes less than 0.5 million gates. Differently, our baseline is a modern, superscalar, out-of-order core, which typically employs several million logic gates.

In Table 5.2 we list and report the estimated area of the additional hardware components

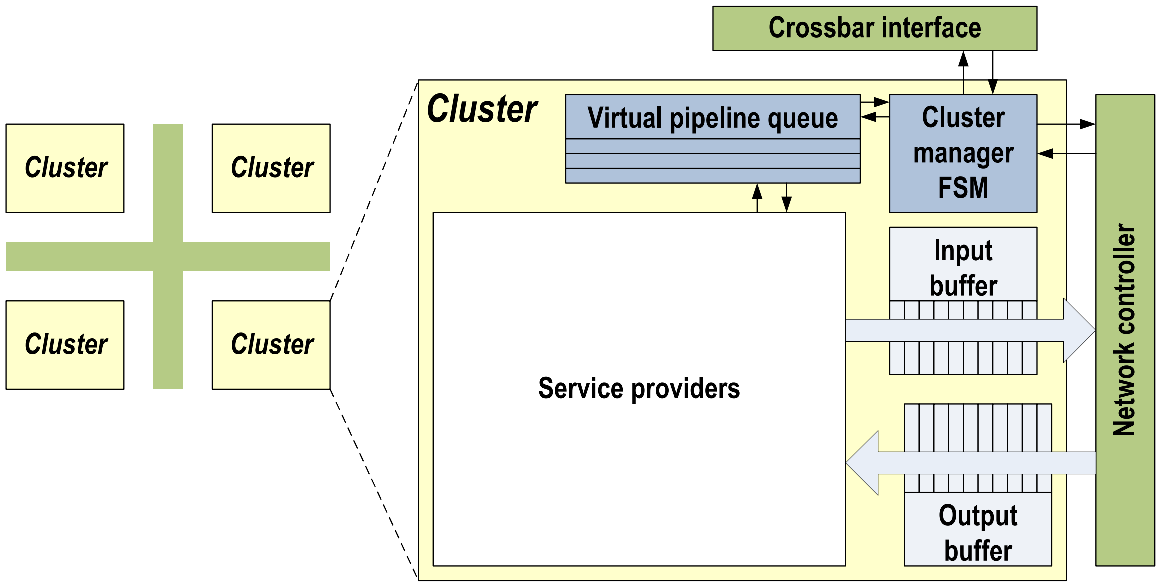


Figure 5.7 Diagram of each Viper cluster listing all additional required components. The logic used by the cluster to execute instructions, reported as "Functional logic" in the figure, is surrounded by a series of additional components that allow it to communicate and exchange data with the rest of the system.

Content	**Size [bits]**
Bundle ID	16
Basic block program count.	64
Next basic block program count.	64
Branch prediction data	4
Previous bundle	4
Next bundle	4
Virtual pipeline	7*6
Input tags	16*24
Output tags	16*24

Table 5.2 BSU storage requirements. Since there are only 16 BSU in our implementation, only 4 bits are needed to index other BSUs. Six services are present in our design, each requiring 2 control bits (assigned, proposal pending) and 5 bits to index the assigned hardware cluster. Finally, 16-bit tags are maintained for the 24 registers of the x86 architecture (9 "general purpose", 6 "segment pointers", 8 "MMX" and 1 for execution flags).

needed by our design. More detailed results are presented in Table 5.3.

Functional units - Every cluster includes input and output buffers for moving bundle data during execution, which adds an area footprint of $0.011028mm^2$. Additionally, as the data for inter-cluster communication is already buffered, every cluster in Viper's execution engine is enhanced with a 5×5 128-bit wide router, each of which occupies 0.11849 mm^2.

BSU - We modeled a BSU with 16 entries in these experiments. The storage required for each entry is reported in Table 5.2, and each entry is enhanced with 58 bits of ECC, for a total storage of 1,536 bytes. The silicon area for this structure is estimated to occupy 0.4173 mm^2.

Crossbar - Our design requires a router connecting 20 clusters with the BSU unit containing all 16 BSU entries. Our estimation for a 21×21 32-bit wide crossbar report that this component should occupy 0.33179 mm^2. We believe that scalability of our design is not jeopardized by this communication element since a much larger crossbar, such as 128×128, still yields a reasonable area footprint of 6.5 mm^2 in 32 nm technology [138].

5.7.5 Power

We used our customized version of McPat to estimate the extra power consumed by our design. Our results are reported in Table 5.3 in detail. Assuming an operating frequency of 1 GHz, we measured that the logic of four complete sets of Viper clusters consumes 20.98169 W of dynamic power, an increase of 4.11% compared against four baseline out-of-order cores, where dynamic power envelope sums to a total of 20.15396 W. Accounting for the communication logic increases the peak dynamic power of Viper to 20.98169 W, a 10.70%

Cluster	**Cluster component**	**Area** [mm^2]	**Dynamic power** [W]	**Static power** [W]
Fetch	Output buffer	0.009369	0.004667	0.000160
	VP configuration logic	0.001567	0.000847	0.000055
	Instruction cache	2.893400	0.189884	0.022981
	Instruction TLB	0.023182	0.007446	0.001359
	Branch target buffer	0.264722	0.014437	0.014830
	Branch predictor	0.076094	0.009695	0.005755
	Total	3.268334	0.226976	0.045140
Decode	Input buffer	0.009461	0.004079	0.000192
	Output buffer	0.009369	0.004667	0.000160
	VP configuration logic	0.001567	0.000847	0.000055
	Instruction decoder	0.939544	0.159214	0.276804
	Total	0.959941	0.168807	0.277211
Tag generation	Input buffer	0.009461	0.004079	0.000192
	Output buffer	0.009369	0.004667	0.000160
	VP configuration logic	0.001567	0.000847	0.000055
	Integer dependency checker	0.001135	0.079607	0.000122
	FP dependency checker	0.001135	0.079607	0.000122
	Integer RAT	0.101413	0.065165	0.005143
	Floating point RAT	0.300066	0.183625	0.012208
	Total	0.424146	0.417597	0.018002
Execute	Input buffer	0.009461	0.004079	0.000192
	Output buffer	0.009369	0.004667	0.000160
	VP configuration logic	0.001567	0.000847	0.000055
	Instruction scheduler	0.111932	0.049163	0.001773
	Integer ALUs	0.274674	0.327811	0.109861
	Floating point units	4.71141	0.820121	0.471102
	Complex ALUs (Mul/Div)	0.329609	0.098700	0.131833
	Results broadcast bus	0.016268	0.543961	0.024771
	Total	5.464290	1.849349	0.739747
Commit	Input buffer	0.009461	0.004079	0.000192
	VP configuration logic	0.001567	0.000847	0.000055
	Trap logic unit	0.232391	0.095265	0.046474
	Total	0.243419	0.100191	0.046721
Register file	Integer RF	0.661799	0.212449	0.004525
	Floating point RF	0.661799	0.159337	0.004525
	Total	1.3236	0.371786	0.009050
Load store queue	Data cache	0.189884	0.189884	0.022981
	Load queue	0.116727	0.017306	0.001122
	Store queue	0.116727	0.017306	0.001122
	Data TLB	0.072523	0.018560	0.002209
	Total	3.199377	0.243056	0.027434
Bundle scheduling units	Memory elements	0.417337	0.014785	0.000024
	Total	0.417337	0.014785	0.000024
Data network router	Crossbar 5×5 (128-bits)	0.0700973	0.111741	0.1358324
	Arbiter	0.000001	0.001064	0.000001
	Links	0.048389	0.076361	0.012354
	Total	0.118487	0.189166	0.148187
Control network router	Crossbar $21 \times 21 (32 - bits)$	0.331788	0.099887	0.705380
	Arbiter	0.000020	0.015924	0.000020
	Links	0.182456	0.127197	0.102889
	Total	0.514264	0.243008	0.808289

Table 5.3 Area and power estimations for all components of the modeled Viper design

increase compared against the baseline processor (20.15396 W). The difference of the estimated static power is higher, since McPat estimates it to reach 9.16886 W, an increase of 53.10% compared against the 5.98876 W reported for the out-of-order processor. The total peak power of Viper then reaches a total of 30.15055 W, increasing the total power consumption estimated for our baseline machine by 15.33% (26.14272 W). We attribute this significant increase in high static power consumption to the extra memory elements and the routers added in the Viper design. McPat's attributes a significant static power consumption for the routers, a total of 3.77203 W – an estimation that we also verified through another publicly available tool, Orion version 2.0 [30]. In summary, these results show that Viper moderately increases overall power consumption, mostly due to the additional communication components.

5.7.6 Performance

We measured the performance and reliability of small CMPs with comparable transistor counts. According to our estimations, an area slightly larger than 60 mm^2 can approximately fit 4 complete sets of Viper clusters or 4 out-of-order processors. Figure 5.8 reports performance figures for single-threaded benchmarks. These performance figures report the MIPS of the SPEC2006 benchmarks that could correctly execute on our simulator. In order to reduce simulation time, we used a faster and less accurate model to fast-forward our simulations for 500 million instructions. We then disabled this faster core, reset all statistics, and activated the detailed cycle-accurate models for 100 million cycles. Viper loses an average of only 22.36% performance compared to the out-of-order core, as reported in Figure 5.8. This is primarily due to the overhead of generating virtual pipelines. We believe that with further analysis and engineering effort it would be possible to recover most of the performance loss.

5.7.7 Faulty Behavior

To better compare the reliability and performance of Viper against previous works, we measured the expected throughput of four different designs for a chip of 2 billion transistors. In Figure 5.9 we compare Viper against the following designs: in-order CMP, Bulletproof, and StageNet. As the graphs show, performance of the unprotected solution, though initially higher, quickly degrades as the number of faults increase. Performance degradation for both Bulletproof and StageNet is more graceful, but their reliance on centralized control logic affects performance as the number of hardware errors grow. On the other hand, thanks to

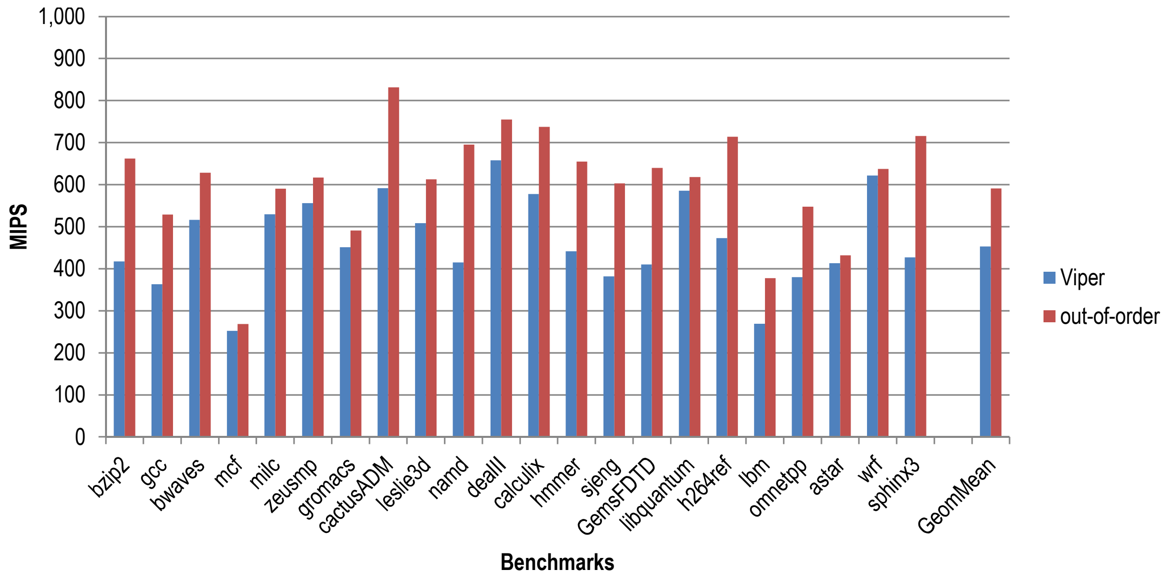


Figure 5.8 MIPS achievable by fault-free configurations of the out-of-order core and Viper.

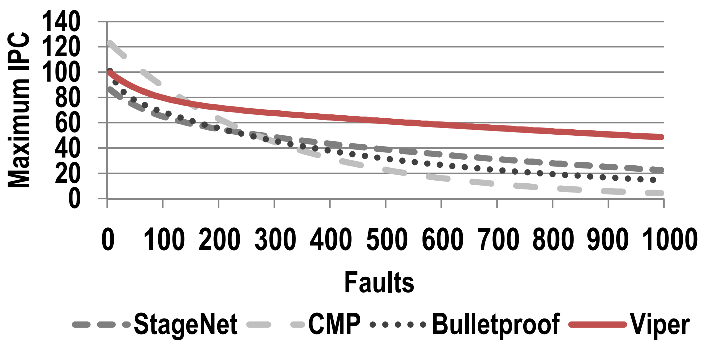


Figure 5.9 Comparison between performance of unprotected in-order cores (CMP), Bulletproof pipelines, StageNets and our solution, Viper.

its distributed control logic, Viper is capable of maintaining higher performance even on silicon substrates tainted by hundreds of permanent faults.

5.8 Summary

In this chapter we developed and evaluated Viper, a new distributed microarchitecture. The design we propose targets modularity as the major driver to limit engineering costs. The studies presented in Chapter 3 inspired the design choices that drove our novel hardware organization. We observed that an application rarely uses all processor's components at the same time. Instead, the majority of applications present a phasic behavior, where programs rely on small subsets of hardware modules for relatively long periods of time. Additionally, we analyzed the organization of current microprocessors and identified the two main characteristics that limit design modularity: their extensive and complex control logic and the tight interconnection of hardware modules.

These insights pushed us to seek an innovative solution that could provide design modularity while maintaining performance and without altering the hardware/software interface. To achieve our goals, Viper uses a reconfigurable execution engine built from independent components guided by a fully distributed control logic. Instructions in this system are viewed as clients that require a number of services. These clients are served by the available hardware components. A program can successfully terminate as long as its required services can be executed by the available components. We showed that Viper is a completely distributed design and is modular by construction and explained how this microarchitecture executes programs and manages exceptions. We also evaluated the performance impact of our solution, and showed that its modular hardware fabric affects performance only by 22.36% on average compared to an out-of-order core. While a baseline system must include several new components, a number of large centralized structures can be removed from the design (for instance the ROB and the RAT). Hence, we estimated the overall additional components of our solution to increase a chip's size by roughly 7.34%. We also measured that our architecture would increase total power consumption by 15.33%.

This chapter concludes the presentation of the three techniques developed in this research to make future architectures reliable, adaptable, and modular – A^2Test, Cardio, and Viper. The next chapter will put together these three solutions to build a comprehensive, adaptable, and distributed architecture: Cobra. This design can overcome transistor failures through Application-Aware self-testing techniques, and adapt to hardware and software changes through the protocols developed in Cardio. Finally, its hardware is based on the modular microarchitecture developed in Viper and presented in this chapter.

Chapter 6
Putting It All Together

The previous chapters described how the three solutions developed so far contributed to enable designs that are adaptive, reliable and distributed. They address what we recognize as being the three major issues posed by future semiconductor technologies: hardware failures, the challenges connected to managing specialized functional units, and the lack of design modularity. This chapter presents Cobra, a holistic solution for future semiconductor technologies that coheres and embodies all the reliable, adaptive and modular features we have proposed throughout this thesis [144].

The complete design presented in this chapter targets highly parallel workloads and represents the pinnacle of this research. First, Cobra's reliability is entrusted to the fault detection mechanisms presented and evaluated in Chapter 3. Second, this design utilizes the distributed protocol introduced in Chapter 4 to allow application execution to dynamically adapt to the available hardware resources. Third, Cobra builds on the novel modular microarchitecture illustrated in Chapter 5. Similarly to the execution model adopted by Viper, Cobra organizes hardware components into a reconfigurable fabric of small, stateless units, operating on bundles of instructions instead of single ones. This chapter also discusses how the traits of the distributed microarchitecture previously introduced can also significantly enhance a system's reliability and adaptability.

6.1 Chapter Organization

Section 6.2 overviews the distributed microarchitecture that constitutes Cobra's backbone. This section also presents the challenges that must be addressed in order to build a comprehensive architecture that can succeed in the scenario imposed by future semiconductor technologies. Following this short introduction, Section 6.3 describes how the communication protocol and techniques for adaptability developed in Cardio in Section 4.3 are deployed in our complete solution. Following, Section 6.4 illustrates further benefits obtained through these techniques that are not available in the baseline distributed design presented in Chapter 5.

With the goal of protecting software applications from hardware failures, Cobra deploys a variety of fault detection techniques, presented in Section 6.5. Applications that strive to maintain fast fault detection latency at a low performance impact can opt for solutions that monitor software symptoms and actively protect only the most vulnerable portions of a program, as presented in Section 3.3. Instead, software requiring maximum reliability can either make use of fully redundant execution or of the application-aware online hardware tests introduced in Section 3.4. Finally, software that does not require any correctness guarantee can disable all online reliability mechanisms for a performance benefit. In Cobra, each application can employ any reliability feature independently from other workloads.

Finally, Section 6.6 presents the experimental evaluation of our complete architecture, analyzing its adaptability to various hardware configurations. This section also measures the ability of our design to handle hardware failures and provides a more formal reliability analysis of our architecture.

6.2 A Reliable, Adaptive Distributed Architecture

Cobra's microarchitecture builds on Viper's execution model, which was detailed in the previous chapter. Several traits make this design modular. First, it organizes its hardware in a sea of loosely connected and independent components, where each component can perform one or more of the services offered by the ISA. Second, these components are fully decoupled from the ones that control program flow and allocate resources. Finally, component interaction occurs solely through packetized point-to-point interconnects and our design treats each instruction bundle as an independent transaction.

While modularity is a static design property of this architecture, adaptability is the ability to change system behavior to match dynamic requirements. In order to provide this second property, resource allocation is managed at runtime through a distributed protocol. While running a program, the sea of units is dynamically configured to provide all the services required by its instructions, a task accomplished by allowing hardware units to independently negotiate their available services with the bundles awaiting execution.

Besides providing modularity and adaptability, our distributed architecture can achieve an unprecedented degree of reliability. In fact, its hardware fabric allows seamless fault isolation and can automatically work around defective components. Each component is decoupled from all others and communicates through a well-defined request/response message protocol. This prevents faults in one component from corrupting the behavior of other hardware modules. When diagnosed, a defective module is disabled and inhibited from executing any instruction bundle without affecting the rest of the system.

Cobra adopts an execution model very similar to the one introduced in Chapter 5. A program is dynamically partitioned into bundles of instructions. Each bundle executing on this system is uniquely identified through a sequential ID generated by increasing the ID of the bundle immediately preceding it. Every bundle in-flight is associated with a scheduling unit, which is responsible for managing and tracking the hardware resources operating on its instructions. Scheduling units do not store operations or values, but only information related to the execution progress: allocated hardware units, pointers to operand locations, data to manage program flow, and instruction sequence. Scheduling units are connected to form a linked list that maintains program order. Once the address of the next instruction bundle is computed, a new scheduling unit is allocated to coordinate its execution. Even though bundles – and even instructions within a bundle – can execute out-of-order, bundles commit their results sequentially, following the order enforced by their IDs [147].

In this architecture, instructions can be scheduled to utilize any available hardware unit that suits them – in contrast to classic pipelines, which push instructions through paths established at design time. On one hand, the baseline modular architecture presented in Chapter 5 offers unparalleled opportunities to isolate defective hardware components and to automatically adapt to hardware changes. On the other hand, this distributed execution model suffers from serious bottlenecks that limit its performance, scalability, and robustness.

The modular distributed architecture proposed by Viper avoids reliance upon centralized microarchitectural structures and tight component connections by construction. These characteristics enable such design to easily work around hardware failures and maintain performance even when affected by a high number of hardware defects. Regrettably, the same properties that make this design more dependable also limit the system's ability to efficiently diagnose hardware malfunctions and restore software state upon fault detection. In fact, bundles from the same application could potentially execute on any of the hardware modules in the system. Since a large number of components might be used to execute a portion of a program, or computational epoch, the periodic fault diagnostic techniques proposed in Chapter 3 cannot pinpoint which process might have been corrupted by a newly discovered hardware failure. Hence, Viper should periodically suspend the entire system to test all components. Furthermore, in case of fault detection, all software applications running on the system will need to be restored from a safe checkpoint.

In terms of adaptability, the original Viper design offers very fine temporal reconfiguration granularity (up to a single basic block). Unfortunately, for highly parallel machines,

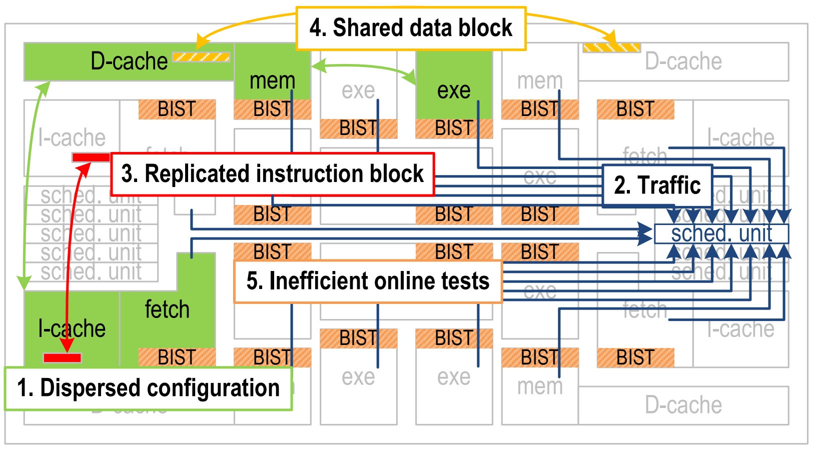

4. Shared data block
D-cache
mem
exe
BIST
I-cache
fetch
3. Replicated instruction block
2. Traffic
sched. unit
5. Inefficient online tests
1. Dispersed configuration

Figure 6.1 Limitations of an unoptimized distributed control architecture: 1) dispersed resource assignments lead to high communication latencies between units (in green) leading to *dispersed hardware configurations*; 2) *hardware configuration setups* entail a large communication overhead (in blue); 3) *scattered workload executions* penalize the performance of stateful resources, such as caches (red); 4) *unordered accesses to memories* affect memory operations' performance and correctness (in yellow); 5) *inefficient hardware tests* affect system's performance (in orange).

the limitations of this approach overshadow its benefits. Since the hardware reconfiguration algorithm does not account for the physical distance of the components in a configuration, runtime may suffer from long data transfers from one end of the chip to the other. Furthermore, because Viper triggers hardware reconfigurations every basic block, its negotiation protocol causes a proliferation of proposal requests and responses. Lastly, Viper's frequent hardware reconfiguration may also hurt overall system performance. In fact, different hardware clusters may service instruction bundles from the same program, possibly causing unordered and scattered memory accesses affecting both a program's correctness as well as performance. In order to tackle all these challenges, Cobra develops a novel reconfiguration algorithm that uses the insights and techniques developed in Cardio to find an optimal tradeoff between adaptability, performance, and communication overhead.

Figure 6.1 briefly illustrates all the issues listed above on a sample distributed architecture.

6.3 Optimized Hardware Adaptation

This section first addresses the latency-agnostic component reconfiguration adopted by Viper in Chapter 5. Solving this first issue allows us to build a system that can use its hardware resources more effectively and succeed in executing highly parallel workloads. Secondly, we propose a technique to extend the lifespan of Cobra's hardware configuration to optimize its performance without jeopardizing its adaptability.

6.3.1 Creating a Localized Hardware Configuration

The first issue that Cobra strives to overcome is the physical dispersion of hardware configurations. The resource assignment algorithm proposed in Section 5.4.2 focuses solely on fulfilling instructions demands. This may negatively affect a system's performance, as such naïve assignment policy does not account for the time spent to transfer instructions and operands across the system. As the number of components in a system grows – and therefore the average distance among them increases – this issue is greatly exacerbated, undermining the scalability of a distributed architecture. Cobra overcomes this issue by leveraging a simple but cost-effective algorithm, which preserves locality without compromising either modularity or adaptability. Note that a straightforward solution would consist in limiting the reach of a service's advertisement broadcasts to a small region of the chip; however, this approach would reduce Cobra's reliability, adaptability and modularity.

When a scheduling unit assembles a hardware configuration, Cobra evaluates the physical location of the units that advertise their availability. In order to do so, components are logically organized in a mesh, so that the distance between any pair of hardware units can be easily computed as their Manhattan distance. While Viper greedily allocates resources as soon as they become available, Cobra's scheduling units store service providers' (hardware units') proposals for a certain number of cycles so as to choose the best among several alternative configurations. Each scheduling unit then generates a hardware configuration based on the services required, taking into account the distances between units in its candidate pool. The target is to minimize communication latency by reducing the overall distance among all the hardware elements forming a configuration. Various algorithms providing different performance/complexity tradeoffs can be employed to this end. An optimal solution to this problem would require the prohibitive (in hardware) application of Dijkstra's algorithm. Instead, in Cobra we opted for a simpler approach that fits our purpose while imposing very small overhead.

Once a new bundle is initiated, its associated scheduling unit accepts proposals from the available hardware units. In order to avoid starvation and deadlocks, the service assignment policy prioritizes the oldest bundle in flight and then assigns resources in the same order as they are utilized by the instructions in a bundle. Differently from the design presented in the previous chapter, Cobra's scheduling units do not include components in their configurations as soon as their services become available. Instead, our algorithm starts from the location of the scheduling unit and adds each required service provider, one at a time (in the sequence they are used by instructions), selecting the available unit that is closest to the previously allocated one. To accomplish this, for a preset number of cycles a scheduling unit stores every service proposal that i) provides forward progress towards the generation of a complete hardware configuration and ii) is physically closer than other proposals received within the preset time window. If resources are assigned in a strictly sequential order (as in this case), this technique quickly converges to generate a configuration that minimizes communication latency. Our solution requires minimal additional hardware – location and type of the next-best candidate and a cycle counter for the search time-window.

Figure 6.2 provides an example of this process. The location of each hardware unit is represented by a pair $<X,Y>$ of coordinates (there can be multiple units at the same location). Components' physical locations are encoded so that distances can be calculated by computing the Hamming distance between coordinates in each dimension and adding them together. Specifically, we use as many bits as the size in the corresponding dimension (6 for X coordinate, 5 for Y, in Figure 6.2), and encode the position by setting a corresponding number of bits to 1. Note that this approach can be extended to three-dimensional layouts such as in 3D stacking [56].

Our buffering of proposals may cause overhead in the hardware configuration setup process. Therefore, we need to find a suitable trade-off between costs and benefits of this approach: we study this aspect in Section 6.6.4.

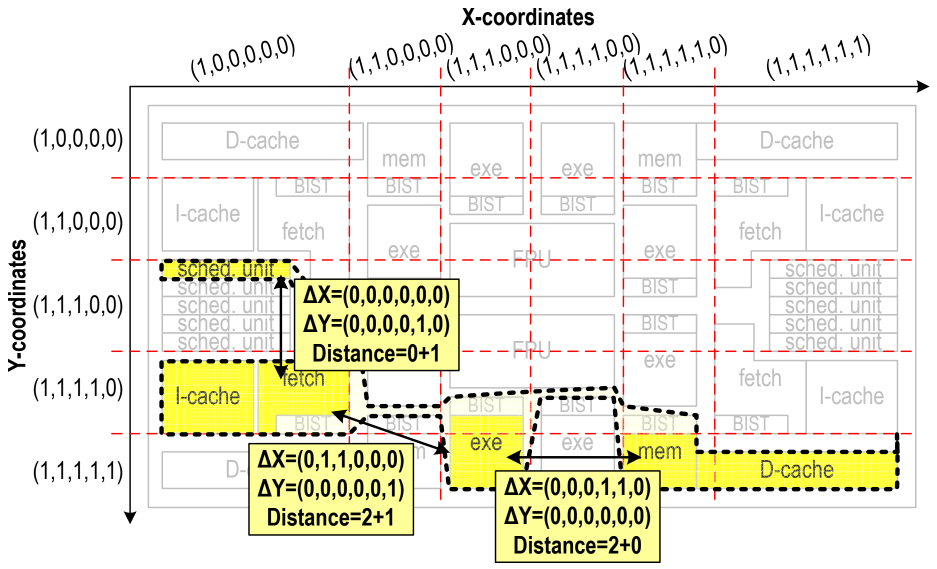


Figure 6.2 Location coordinates used to setup Cobra hardware configurations. Each coordinate is always encoded as a sequence of bits, whose length corresponds to the maximum number of segments it spans. Each segment is identified by the number of "1s" in the leading bits of its coordinate. The distance between two components in a hardware configuration (in yellow) is then determined by summing each direction's hamming distances.

6.3.2 Hardware Configuration Lifespan

The service-based architecture presented in Chapter 5 offers an incredible degree of reconfigurability and adaptability. Viper allows hardware components to independently advertise their availability to any scheduling unit that is attempting to execute an instruction bundle [147]. Components whose services are included in a hardware configuration are then notified by the relative scheduling unit. This process dynamically allocates the hardware resources that fit a program's requirements without relying upon centralized control structures. However, this mechanism leads to message proliferation due to the large number of advertisements and notifications exchanged among the hardware components, causing a significant communication overhead.

To understand the magnitude of this problem, consider for instance a distributed architecture composed of 5 services, 4 providers for each service and 2 active bundles (groups of instructions). Assume also that each bundle is already mapped to a scheduling unit. The maximum number of messages exchanged between the hardware units and the scheduling units is: $5(services) \times 4(providers) \times 2(bundles) = 40$ service proposals and $5(services) \times 2(bundles) = 10$ acceptance notifications. Considering that bundles consist of a single basic block, and more than 90% of them are up to 16 instructions long, the number of messages exchanged to execute a rather short portion of the program is significant. Such overhead grows linearly with the number of service providers and in-flight instruction bundles, severely hindering a system's scalability. This problem is further exacerbated for workloads requiring redundant execution for reliability purposes.

This overhead occurs because of the short life-span of a hardware configuration: one for each instruction bundle. In practice, however, a system rarely needs to be reconfigured because of either a newly discovered fault or for changes to a workload's hardware-resource needs. For instance, we have empirically shown in Chapter 3 that application execution presents a phasic pattern, and long portions of a program often rely on a small subset of the available hardware features [142]. Furthermore, in Chapter 4 we demonstrated how we can achieve effective hardware adaptability simply by allowing processes to notify each other about hardware state and utilization at the end of every computational epoch.

For these reasons it thus makes sense to use the same hardware configuration over longer execution periods, so as to amortize configuration costs among many groups of instructions. The remainder of this section introduces the mechanisms that we developed in Cobra to extend the lifespan of a configuration to an entire computational epoch.

Hardware Configuration Transferring

Cobra allows an instruction bundle to pass on its hardware configuration to the following bundle (through the linked list of scheduling units). Compared with the design in Chapter 5, which generates a hardware configuration for each new instruction bundle, this enhancement does not need any major modifications, as it only requires transferring the already established configuration to the scheduling unit assigned to manage the next bundle. This simple operation greatly reduces the cost of the service negotiation procedure, as – in the common case – it only consists of notifying the hardware components servicing the current bundle that they will also be employed by the following bundle. If the instructions in this latter bundle require additional services, the associated scheduling unit will initiate a service negotiation procedure to acquire new providers only for the newly needed services. It is worth noting that Cobra can extract less ILP from a single application than Viper. This is because extending hardware configuration lifespan forces all bundles in a thread to execute on the same set of modules, basically serializing bundle execution. Therefore, in Cobra, functional units are always working with an instruction window equal to the number of instructions in the bundle in-flight. As we show in Section 6.6.4, the benefits of this solution outweigh the costs for multi-threaded workloads.

Figure 6.3 illustrates this approach with an example. The units in the yellow area belong to a hardware configuration that has completed the execution of a bundle. When the next bundle begins execution, its corresponding scheduling unit (marked in blue) receives the set of hardware units from the previous one. This setup information can be piggybacked on the notification that allocates a new bundle to a scheduling unit. Moreover, the bundle associated with the blue scheduling unit requires the services of an FPU, which is then added to the set of hardware units servicing the bundle.

Hardware Configuration Tearing Down

While our configuration transferring approach greatly reduces communication overhead, at times it is beneficial to tear down a hardware configuration and create a new one from scratch. This is the case, for instance, when a configuration becomes under-utilized because some of its previously assigned resources are not needed by the more recent bundles. This situation becomes critical when bundles from other threads require some of those resources that are no longer used. Moreover, a fault could manifest in one of the units participating in an active hardware configuration. When any of the above situations occurs, the corresponding scheduling unit simply prevents the transfer of the hardware configuration to the following bundle.

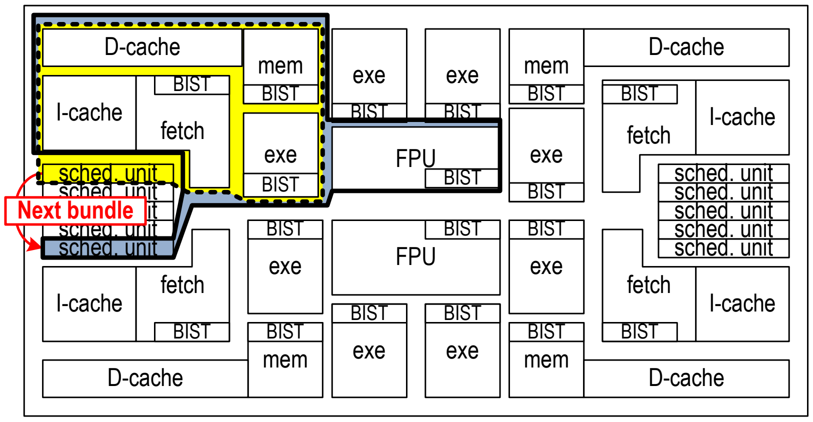


Figure 6.3 Hardware configuration transferring used in Cobra. When a scheduling unit and its associated hardware configuration (yellow area) complete the execution of a bundle, the hardware configuration is transferred to the scheduling unit servicing the following bundle (blue area), adding more units if needed (an FPU in the Figure).

Our design uses the same approach developed in Chapter 4 to track the relevant events related to resource utilization. However, while Cardio relies on data structures stored in memory to keep track of hardware state and utilization, Cobra's scheduling units are augmented to store the list of service components acquired and used – via a single-bit flag per service. Such a list is transferred from one scheduling unit to another in order to preserve memory of the hardware components used during a computational epoch. Flags are propagated from a bundle to the next, updated every time a new group of instructions is fetched from memory, and reset periodically to keep information relevant. It is worth noting that updating service utilization flags does not increase the communication traffic over previous distributed architectures, since information about the services needed must be available even in a baseline system.

Maintaining information about hardware utilization is also beneficial for the management of scarce resources. Scheduling units that cannot obtain availability from one or more service providers can request the scheduling units that reserved these resources not to forward their hardware configuration – thereby forcing a new service negotiation procedure. This mechanism guarantees forward progress while allowing sharing of scarce resources. Note that this procedure relies on the same protocol used by the instances of Cardio's distributed resource manager.

As in Cardio, the end of a computational epoch can trigger a series of introspective tests meant to assess whether the underlying hardware is healthy and utilized effectively. However, while software applications handle resource availability in the CMP system considered in Chapter 4, resources in Cobra's reconfigurable computational fabric are managed through scheduling units. At the end of each epoch, Cobra's hardware configurations are torn down and tested. Scheduling units that do not benefit from a hardware configuration transfer will proceed to setup a new hardware configuration. It is worth noting that faulty units or units under test will simply not advertise their services, and thus they will not be included in any new configuration.

6.4 Scalable Performance

The benefits of maintaining longer hardware configurations are not limited to a more efficient hardware utilization but also allow significant advantages in on-chip memory utilization. This section will briefly discuss how Cobra can improve on-chip memory utilization.

6.4.1 Temporary Data Persistence

Workloads running on a dynamically configured architecture may experience scattered execution due to the fact that consecutive bundles belonging to the same process can be executed by different hardware units. This severely affects the effectiveness of units that leverage temporary information storage to enhance performance. For instance, caches, which could exploit both temporal and spatial locality, are not able to do so if groups of instructions from the same thread are constantly swapping caches. This also holds true for other performance-enhancing features, such as branch predictors and TLBs.

In Cobra, this problem is solved as a by-product of our hardware configuration transferring approach. Indeed, when the same hardware configuration is used to service a large number of subsequent bundles, temporary data is naturally and effectively maintained in these memory structures. Our experimental evaluation quantifies this benefit over the unoptimized modular distributed-control architecture presented in the previous chapter.

6.4.2 Boosting Memory Access Performance

Managing data memory operations is particularly challenging in a distributed architecture because program semantics expect memory operations to be issued and completed in order. This expectation may not be easy to meet when instructions from distinct bundles execute on different hardware configurations. Moreover, distinct bundles (possibly one logically following the other) might need to update the same memory location, leading to multiple "store"-service hardware units requesting exclusive access to the same cache line – a rather expensive procedure, as it requires all caches to invalidate their local copy. This problem is exacerbated in distributed architectures supporting memory-to-memory instructions, such as x86, where memory accesses are even more frequent.

In the previous chapter we proposed a single memory access point for the entire system. While this is effective in boosting the single cache hit rate, it comes at a high impact to memory access time and leads to poor system scalability. In contrast, our aim is to keep a multitude of caches in the system, which are geographically distributed and relatively small, in order to boost performance. To this end, Cobra maps caches in the system to a set of threads: each thread in the set shares the cache assigned to it exclusively with the other threads in the same set. This is achieved by mapping each Load-Store Queue (LSQ) to only one data cache. Since each hardware configuration (and thus, each thread) can only include one LSQ, this guarantees that memory ordering and data locality benefits can be attained within the set. At the same time, this solution provides system scalability, since overall, a system can still leverage multiple memory access units and memory structures.

An available LSQ unit is assigned to the first bundle generated by a new thread – this is performed through the same negotiation mechanisms used to allocate any other service in the system. Machines that want to maximize throughput and availability can share a LSQ among multiple threads. The mapping between a thread and a LSQ unit is recorded in the relevant scheduling units, and is propagated from one bundle to the next, releasing the LSQ only when a thread terminates or is de-scheduled by the OS. Note that a bundle can be associated to a hardware configuration with multiple execution units; however, the configuration would still have only one LSQ unit. In addition, memory operations ordering is enforced by using the bundle sequence ID. This can also be used to detect mis-speculations of memory values.

6.5 Reliability

One of Cobra's key objectives is to maximize system availability in the face of hardware failures. In this section we detail how distributed architectures can detect and manage runtime faults. As already mentioned in Chapter 3, overcoming these events is a three-stage process consisting of:

1. fault detection;
2. hardware reconfiguration;
3. system state restoration.

First, a comprehensive reliable system must be able to dynamically detect errors and diagnose faulty components. Several techniques are available for this purpose (redundant execution, symptom-based detection, online testing), each of which can trade fault coverage or detection latency for performance overhead. Unfortunately, none of them have been tailored to distributed architectures, so here we focus on enabling these fault detection mechanisms for this novel design paradigm, particularly for Cobra.

Second, the distributed processor architecture developed in Cobra empowers a system to automatically reconfigure itself around hardware errors. Hardware units deemed faulty can be selectively turned off, thus preventing them from advertising their services to the rest of the system.

Third, independent from the fault detection mechanism deployed, upon fault detection, all bundles in flight are flushed through a system-wide broadcast signal to all scheduling units. The hardware failure is then diagnosed, and the newly discovered faulty component disabled. Both architectural state and memory system is restored to a previous safe checkpoint through techniques such as ReVive or SafetyNet [150, 176], and each checkpointed program is restored and mapped to an available scheduling unit. In Cobra program state is distributed, because the information necessary to manage program execution is stored in the scheduling units while the architectural register values are located in the reconfigurable fabric: these two pieces of information are synchronized to form a unique checkpoint state when instruction bundles commit.

It is worth noting that handling faults in the scheduling units does not require advanced mechanisms. The vast majority of the area in these units consists of storage elements, which can be protected through ECC, while their relatively small logic can be made resilient through hardware duplication [147]. Finally, in this work we do not account for failures on caches and interconnect, as these subsystems can be effectively protected by other techniques [4, 55].

The remainder of this section considers three classes of fault detection solutions previously proposed for classic pipelined processors: full redundancy, selective redundancy and periodic online testing. While the first solution is already commonly adopted in mission critical systems, this thesis already evaluated the effectiveness of the two latter techniques in Chapter 3.

6.5.1 Full Redundancy

The first technique, full redundancy, is both the simplest and fastest mechanism for detecting any type of hardware errors – due to permanent, intermittent and transient failures. Each instruction executes multiple times on different hardware components, and the results are then compared to detect and possibly correct errors that may have occurred. Although highly effective, this mechanism is very performance- and resource-intensive, and only applications that value reliability as a key requirement adopt it. Corrupted results are recognized immediately, and a faulty hardware component is diagnosed by comparing the outcomes of multiple executions.

To deploy this fault detection mechanism in Cobra, we must enhance the scheduling units to accommodate multiple hardware configurations, one for each redundant execution – these configurations must contain non-overlapping sets of service providers. In addition, they must include a voting unit to compare results produced by the redundant configurations. This check can be done at a fine granularity, comparing each result produced by every instruction, or at the bundle level, comparing only the results that are transferred to subsequent bundles. In addition, the scheduling unit must allow results to be committed to memory only after they have been successfully checked.

Bundles to be executed redundantly are tagged by a special flag, so that all units servicing such bundles are aware that the instructions within require special handling: they are not allowed to alter process state or program flow until after they have been deemed fault-free – in our system we maintain a single software process for all redundant executions. Furthermore, they cannot update the register renaming table of a process (or the tags associated with an operand), and redundant store instructions sent to the LSQ are not committed to memory but are instead discarded once the bundle completes. Finally, redundant memory loads are not allowed to receive values forwarded by store operations in the other executions of a same bundle.

6.5.2 Selective Redundancy

Most programs do not require the degree of protection guaranteed by a fully redundant execution. In fact, we have already shown through our experiments with CrashTest in Chapter 3 that the majority of hardware malfunctions (up to 90% of permanent faults and 60% of transient faults) can be detected at zero-cost by simply monitoring software anomalies such as kernel panics, fatal traps and illegal memory accesses [53, 107, 203]. We do not address these diagnostic technique in this dissertation since several works provide low-cost solutions to diagnose faulty components by analyzing these software malfunctions, and any of them could be used for this purpose [81, 107].

Unfortunately, even though this approach provides a significant fault coverage for most hardware failures, it is not very effective in detecting faults that silently corrupt an application's outputs. Cobra provides an ad-hoc solution for checking the portions of a workload that are particularly susceptible to silent data corruptions (SDC). Our experiments in Section 3.3 show that particular types of instructions – floating point operations, divisions, multiplications and SIMD instructions – are particularly prone to SDCs [107, 148]. However, the service-based microarchitecture we develop can help to address these insidious hardware failures. In fact, the scheduling units of our distributed system maintain detailed information about the services required by the executing bundles, they can dynamically flag the portions of a program that are vulnerable to SDCs. Therefore, we allow processes to request "selective redundancy": Cobra will then activate redundant executions only for the bundles that include operations vulnerable to SDCs. This technique represents a good compromise for processes that need high fault coverage but cannot afford the cost of a fully redundant execution, and it has been shown to be very effective in exposing permanent failures [130].

6.5.3 Periodic Online Testing

We already advocated in Chapter 3 that periodic online testing can effectively protect software from permanent hardware failures. This approach assumes that the results generated by a processor cannot be trusted until its underlying hardware components have been tested by periodic self-tests (for instance, with a built-in-self-test unit, BIST). Only after all tests succeed, a process is allowed to commit its results to memory and to I/O devices. With this approach, execution time is partitioned in epochs, which are typically a few million instructions long, and hardware tests are executed at the end of each epoch [41]. We previously showed that periodic hardware testing is both very economical (as low as 1% hardware overhead [142]) and effective (up to 100% fault coverage [41, 110]). Handling faults in the test logic is a problem common to all solutions that adopt this fault detection technique, and in this work we assume that a faulty self-test logic causes its related hardware component to be marked as non-functional.

In Viper, the bundles of a process could potentially execute on any service provider (hardware unit) in the system. Therefore, a large number of units (possibly all) could be exercised during an epoch's execution, requiring many tests to be completed before the epoch's results can be deemed fault-free – causing performance hiccups. In addition, a single faulty unit could impact many processes within one epoch, requiring many or even all processes to be rolled back to their previous checkpoint. In contrast, in a traditional CMP, only a single process would be affected.

Cobra addresses both of these issues. As long as a process does not tear down a hardware configuration for the duration of an epoch, applications running in Cobra maintain exact knowledge of which components were exercised. Therefore, the health of a process during an epoch can be assessed by testing only a small subset of hardware units. Compared against previous solutions, our design provides an additional advantage to online testing, as it does not require interrupting program execution. Two solutions are possible in case of tear-downs occurring during an epoch:

1. if infrequent, the system might force tests on all hardware in the torn down configuration;
2. if frequent, the scheduling units can store the list of all units used in an epoch and, at the end, run tests on all the exercised hardware.

Finally, upon detecting a fault, only the processes that exercised the defective hardware unit need to be restored from their checkpoints. Before presenting our experimental results, we summarize in Table 6.1 below the problems that Cobra strives to address and the solutions developed to tackle each of them.

Problem	**Solution**
1. Physical dispersion of HW configuration	Localized HW configuration
2. Large communication overhead	HW configuration transferring
3. Scattered execution of a workload	HW configuration localized & transferring
4. Unordered accesses to memory	New cache design, HW configuration transferring
5. Efficiency in self-testing	HW configuration transferring

Table 6.1 Summary of the problems addressed by Cobra and the techniques developed to solve them.

6.6 Evaluation

We modeled a Cobra system that implemented the x86-64 ISA and evaluated both its performance scalability and its ability to endure hardware failures. To this end, we compared Cobra against two similarly sized designs: a classic CMP comprising 2-wide out-of-order cores and the baseline modular, distributed architecture presented in the previous chapter, Viper [147]. We chose the former because it matches the characteristics of modern CMPs [22, 64], and the latter because it represents a state-of-the-art distributed architecture. In order to measure Cobra's scalability, we considered systems which can execute 1, 2, 4, 8 and 16 threads, and whose processor logic (caches excluded) occupies 20M, 40M, 80M, 160M and 320M transistors, respectively.

We first analyzed Cobra's performance and capability to adapt to hardware changes compared to the CMP and the unoptimized Viper design. Therefore, we measured the impact of the solutions developed in Cobra on the throughput of systems of different sizes, running multiple instances of the SPEC2006 benchmarks. No faults were injected in these first simulations. We then evaluated Cobra's robustness to an increasing number of permanent failures and compared it against a classic CMP solution. Finally, we investigated the performance impact of deploying the online fault detection mechanisms presented in Chapter 3 on Cobra.

6.6.1 Hardware Model

All three architectures evaluated in this work are clocked at a frequency of 2.0GHz. Cores in the classic CMP configuration have an execution windows of 32 instructions, can commit up to 2 instructions per cycle, and can have up to 5 in flight: 2 in integer pipelines, 2 in FP units, 1 in the load/store unit. Each OoO core uses dedicated 32KB L1 data cache and L1 instruction cache, while all service providers in Cobra and in the baseline distributed architecture can make use of similarly-sized data and instruction caches – one for each thread executed on the machine. In order to ensure program correctness, both distributed architectures tie a process to only one data cache before starting execution, as discussed in Section 6.4.2.

Service	Hardware unit	Number of units	Test cycles	Transistors
Fetch bundle	Fetch	4	1.25M	4M
Generate next PC				
Decode bundle	Decode	4	1.25M	2.5M
Tag generation	Tag	4	1.25M	3M
Integer ALU	Execute	8	1.25M	1.6M
Load & Store		4		951K
Integer mult. & div.		8	327K	1.27M
FP ALU		8	230K	635K
FP mult. & div.		8	230K	635K
SIMD		4	1.57M	635K
Update register file	Commit	4	1.25M	1M
Commit stores				
Create new bundle	Scheduling unit	32	–	15K

Table 6.2 Characteristics of the distributed architectures evaluated: Cobra and Viper [147]. Both architectures provide 13 services using 6 different hardware units. For each unit type, we report the number of instances available in the configuration executing four threads, the length of a complete test and the area in transistors of each unit [142].

To setup both of the distributed architectures under analysis, Cobra and Viper, we partitioned the x86-64 ISA into 13 different services, which are provided by six different hardware units, as listed in Table 6.2. In our fault model, a failure hitting a multi-service unit disables only one service in that unit. For instance, an "Execution" unit, which is hit by a fault in its "FPU ALU" service provider will no longer be able to execute any bundle that requires that service, but can still provide its other services. Periodic self-test are scheduled independently on each unit after it has executed a certain number of instructions, 20M in our case [142]. In order to utilize the A^2Test technique introduced in Chapter 3, we embed an instruction counter in each hardware unit in order to trigger self tests: once the limit is reached, the hardware tests are serially performed on all the services still available. With reference to our example, the "Execution" unit would skip testing the hardware of its already faulty "FPU ALU" service. Table 6.2 reports several characteristics of the distributed architectures considered. Configurations supporting a different number of programs are scaled proportionally (the 16-thread configuration will have 16 fetch units). The last two columns of the table report the number of cycles needed to test each service and an area estimate for the unit. Connectivity between scheduling units and service providers is established through a crossbar, which has a point-to-point latency of 4 cycles [206]. Both distributed architectures arrange the "Fetch", "Decode", "Tag", "Execute" and "Commit" units in a mesh, each unit connecting to its neighbors with 256-bit wide links. The communication latency between two adjacent nodes is 1 cycle [147].

In our analysis, we focus on evaluating the relative throughput measured in instructions per cycle (IPC). Although we have not thoroughly investigated the impact of our novel design on the critical paths of the system, our changes primarily affect the length of the pipeline, as for similar solutions [73]. Hence, we do not expect a negative influence on the cycle time in Cobra compared to the baseline. Moreover, while classic pipelined processors rely on long wires to connect distant pipeline stages, interactions among Cobra's components occur through packetized point-to-point interconnects, which allow aggressive pipelining and high operating frequencies. In summary, our assumption that Cobra's operating frequency matches the one of the baseline processor considered is rather conservative.

6.6.2 Software Benchmarks

We used the SPEC CPU2006 benchmark suite [83] to evaluate Cobra's performance. Due to the detail and complexity of our simulations, we could not run all benchmarks to completion. Instead, we evaluated their performance when they reached a steady execution state. The benchmarks in this suite were run with the "test" input set. We evaluated performance for both single– and multi– programmed workloads, executing independent copies of the same benchmark. In order to measure Cobra's scalability we evaluated a system executing 1, 2, 4, 8 and 16 programs, and scaled the resources available accordingly. As for the other experiments, for reasons of space, we focused our attention on a medium-sized system executing four programs, and analyzed its performance and reliability in more detail. We decided to run multiple copies of the same benchmark at the same time – instead of a mix of benchmarks – because we wanted to stress the system by creating high contention on the hardware components exercised by their instructions. These results are therefore somewhat conservative, as mixes of workloads might decrease resource contention. Finally, since we focus on systems capable of executing multiple programs at the same time, reliability estimations are reported as throughput on the multiprogrammed workloads running four copies of the same program.

6.6.3 Simulation Infrastructure

The microarchitectural simulation platform adopted for this work is based on the gem5 simulator [19], employing full timing simulations in system-call emulation mode. The model of the OoO core provided with gem5 was modified to match the deeper pipelines typical of modern high-performance processors. The minimal execution latency of an instruction in such design is 12 cycles: 3 to fetch, 3 to decode it, 3 to rename the architectural registers, at least 1 for execution and 2 cycles for committing the instruction [78].

Fault Model – gem5 does not natively include a fault injection model. Thus we augmented the baseline simulator with: 1) a parameter representing transistor count for each hardware component – so that we could create a uniform distribution over the chip's area when injecting faults; 2) a fault injector capable of randomly triggering a fault in one of the hardware components. Faults are injected with a uniform distribution in all service providers listed in Table 6.2, proportionally to the area occupied by each of them. In order to gain statistical confidence in our results, each fault injection experiment was repeated 20 times, each one with a different random selection of fault locations. Once a service is affected by at least one fault, it is considered no longer operational. Note that we did not consider defects in memory arrays since those can be easily avoided through ECC and redundant entries. Finally, we did not inject faults into the intra-chip interconnect, as several techniques are already available to protect the communication subsystem from hardware failures [55].

6.6.4 Hardware Adaptability & Fault-Free Throughput

In the first set of experiments, we compared the throughput of a CMP design and distributed architecture against Cobra, and evaluated the impact of each of the techniques discussed in Sections 6.3 (localized hardware configurations) and 6.3.2 (hardware configuration transferring). We compared 1, 2, 4, 8, and 16 concurrent processes to gauge Cobra's adaptability to hardware modifications. Our findings are reported in Figure 6.4, where we compare the throughput of Cobra (3 solid lines), against that of a similarly sized CMP design and Viper.

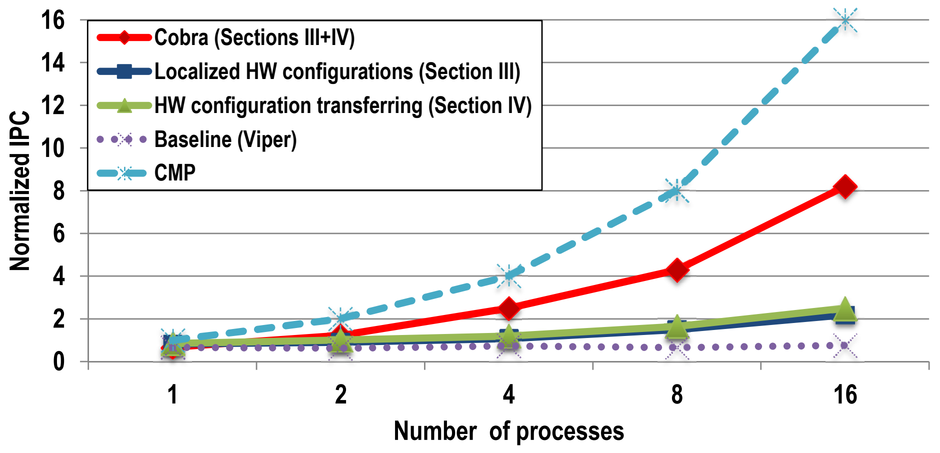


Figure 6.4 Throughput vs. system size. The plot compares Cobra (solid lines) against a CMP and a baseline distributed architecture.

It can be noted that our two performance-boosting techniques, "localized HW configurations" and "HW configuration transferring" enable a distributed-control solution to approach the performance and the scalability of a classic CMP system. We analyze this result in more detail below.

Localized Hardware Configuration – To avoid starvation, scheduling units must assign hardware units to services in an ordered fashion, thus we proceed top-down through the services listed in Table 6.2. The unoptimized Viper design in our experiments allocates each required service to the first hardware unit available in a greedy fashion. In contrast, Cobra uses our localized configuration approach, buffering service proposals for a number of cycles (in our experiments we used 10, slightly more than twice the crossbar transmission latency). Figure 6.5 plots the contribution of this technique over a baseline distributed architecture (Viper) in a 4-threaded configuration for each SPEC2006 benchmarks, showing that by itself it introduces an average performance improvement of 23%.

Hardware Configuration Transferring – We then evaluated the impact of allowing a bundle to directly transfer its hardware configuration to its successor. Hardware configurations are torn down every 20 million instructions (a reasonable length for a computational epoch, as shown in [41]). This technique brings an average performance improvement of 42% on a 4-threaded system, as indicated in Figure 6.5. Correspondingly, we observed a significant reduction in the number of messages through the crossbar – 61% and 52% for single process and multi-programmed benchmarks, respectively.

Cobra – The two techniques combined contribute an overall performance boost of 79% over a baseline distributed design (see Figures 6.4 and 6.5). When compared to a CMP system, the performance of Cobra falls short by only 21% – a small incidence when considering the benefits in reliability and flexibility.

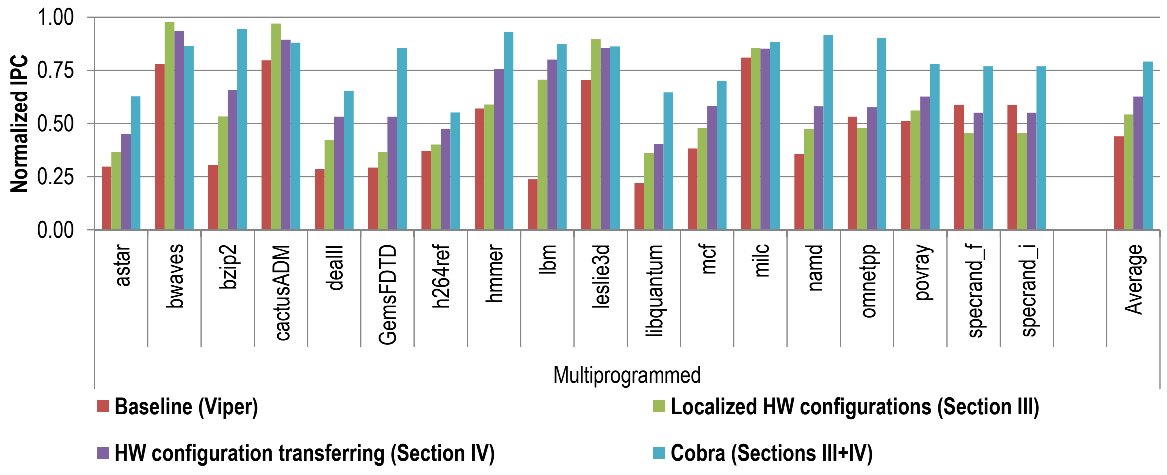


Figure 6.5 Contribution of Cobra's performance features to overall IPC for a 4-threaded system. IPC is normalized to that of a similarly-sized CMP.

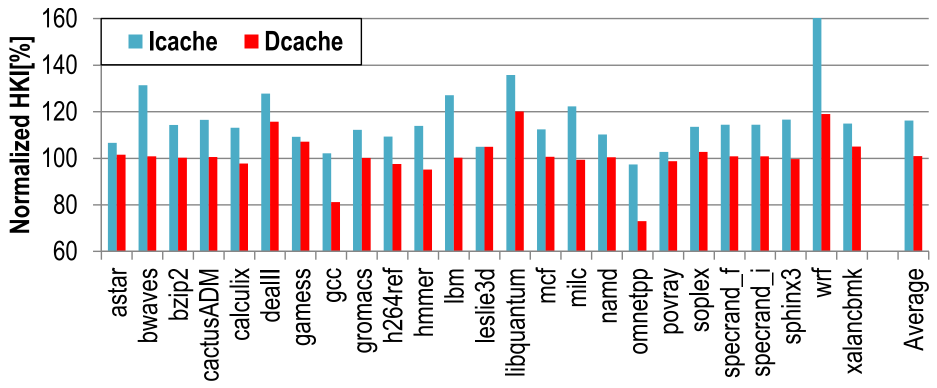


Figure 6.6 Cache hit improvement – normalized hits per thousand instructions (HKI) – due to Cobra's solution for memory accesses. Results for both data and instruction caches over the multiprogrammed workloads.

We also evaluated the ability of Cobra to boost memory access performance: Figure 6.6 plots the improvement in cache hits per thousands instructions (HKI), relatively to the unoptimized distributed system. Note that data caches are affected by only a minimal performance difference, since both Cobra and Viper map one cache per process to maintain correct execution. In contrast, the instruction caches greatly benefit from Cobra's approach, since they can attain much better utilization: the multiprogrammed workloads experience an average of 16% more cache hits.

Next, we performed a sensitivity study that correlates the number of the BSUs with the performance of the machine. This experiment has been performed on a single, very CPU-intense synthetic benchmark (overall branch predictor accuracy is 83%). The maximum bundle size is set to 16 instructions, and the large majority of the bundles (43%) contain 14 instructions. Only one process is executed on this machine. We swept the number of

BSUs available in the system from 2 entries (the minimum required for a process to make forward progress) to 128. Since performance improves only marginally for configurations containing more than 16 BSU entries, Figure 6.7 does not report them.

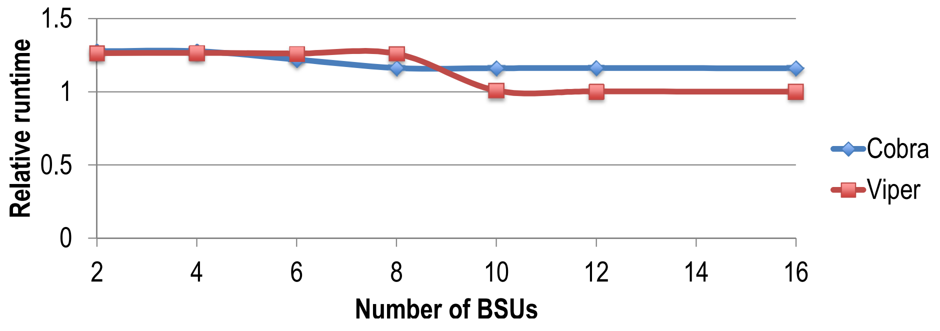


Figure 6.7 Cache hit improvement – normalized hits per thousand instructions (HKI) – due to Cobra's solution for memory accesses. Results for both data and instruction caches over the multiprogrammed workloads.

We report performance as the ratio between the runtime of each configuration and the baseline – the best Viper configuration, containing 128 BSUs, is used as the baseline. The closer the ratio is to 1, the better that configuration performs. As shown in Figure 6.7, what is important for the system is to have enough BSUs to keep as many hardware modules occupied as possible (this design contains 4 copies of 5 different types of hardware modules). Once the process maximizes its hardware utilization, there is very little advantage in increasing the number of BSUs. This threshold is 8 BSUs for Cobra and 10 for Viper. This difference is due to the fact that Viper can use multiple virtual pipelines to execute the same program (so it needs more BSUs to maintain runtime information about the bundles in flight), while Cobra cannot (execution is serialized by the hardware configuration transferring mechanism). This is also the reason Viper performs better than Cobra on single-threaded benchmarks – in our experiments we measured an improvement of 15.8%. Finally, note that Viper performs slightly worse than Cobra for smaller numbers of BSUs (2, 4, and 6). This is mainly because this system cannot parallelize benchmark execution, and cannot take advantage of localized virtual pipelines (this problem significantly worsens on multi-programmed workloads).

6.6.5 Reliability

The next set of results evaluate Cobra's reliability and measures the effects of hardware failures on its performance. For these experiments we only considered the multiprogrammed SPEC2006 benchmarks and injected faults at the beginning of each simulation. Since our reliability-boosting techniques are unaffected by the scale of the system, we only provide results for 4-threaded systems and workloads.

Survivability – We first report our survivability, in terms of probability that a system can still execute all the instructions in its ISA, over a range of permanent hardware failures. In Figure 6.8, we plot our findings for both Cobra and the CMP design. After 7 faults, the probability of not being able to execute all instructions for the CMP design is 0.5 – correspondingly Cobra reaches this same level of exposure at 20 faults – one permanent fault per 5 million transistors.

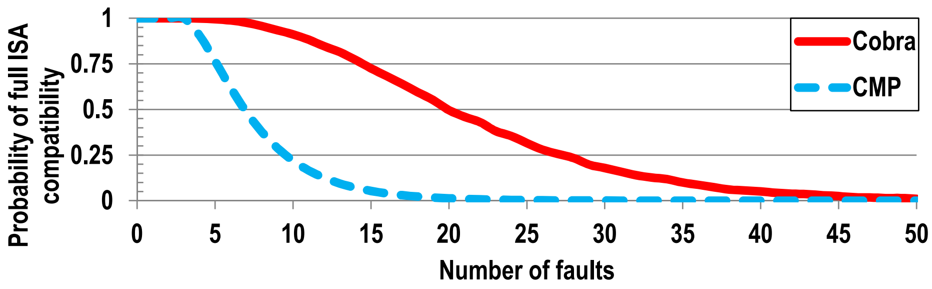


Figure 6.8 Survivability of Cobra vs. CMP. Probability that the system can execute all ISA instructions in the face of increasing faults.

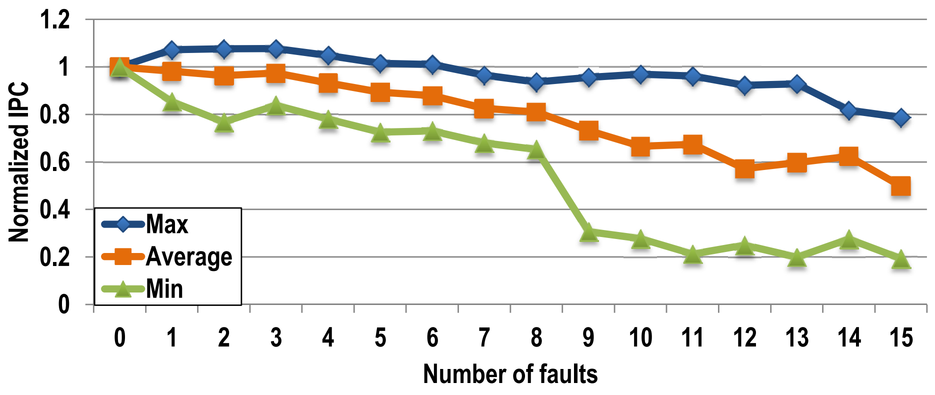


Figure 6.9 Cobra's performance degradation in presence of failures, averaged over 24 SPEC2006 benchmarks. Results are normalized to a fault-free Cobra solution. The throughput increase reported in the Max curve shows that small perturbations to a distributed architecture may occasionally enhance our resource assignment algorithm.

Performance in presence of faults – We also measured the performance degradation of our system when subjected to permanent failures as reported in Figure 6.9. For this study we only considered faulty systems that can still execute all ISA instructions, and we averaged our results over all SPEC2006 benchmarks, running 20 distinct simulations per benchmark to gain statistical confidence.

It can be noted that Cobra's average throughput degrades gracefully with increasing faults, roughly halving at 15 faults. Figure 6.9 reports max and min relative IPC, in addition to the average overall all 20 runs for the 24 benchmarks. We observed that injecting a moderate number of faults (between 1 and 5) can occasionally lead to a performance boost

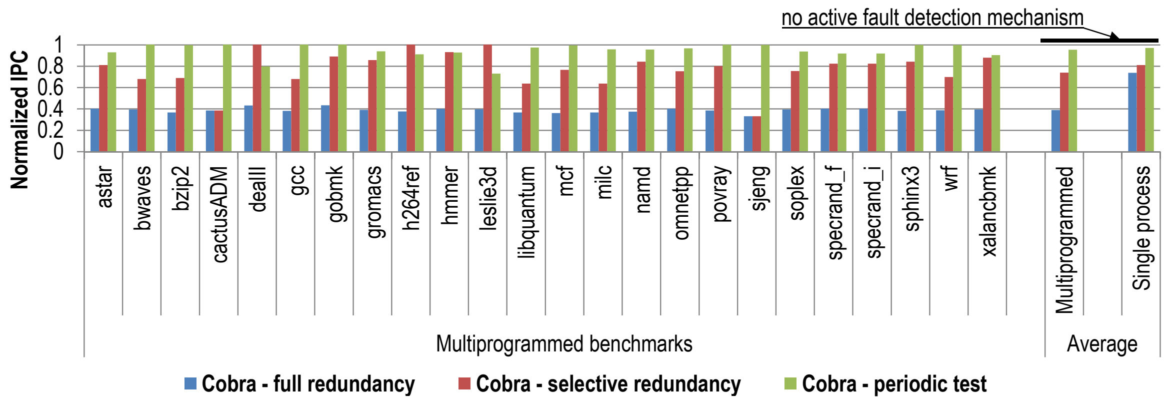


Figure 6.10 Performance cost of the three fault detection mechanisms for Cobra. IPC is normalized to a Cobra system with no active fault-detection mechanism. Note that the periodic test technique cannot actively protect against transient faults.

(as it can be noted from the max line in Figure 6.9). This is due to the fact that our resource assignment algorithm is based on algorithms that search for a local optimum. Thus, small variations in the available resources may lead to a better-performing solution.

Finally, Figure 6.11 compares Cobra's performance against that of a CMP design in presence of faults. Values are reported relative to the performance of a single fault-free OoO core, and the graph compares performance for area-neutral designs. The CMP design considered in these experiments is not affected by the interactions between the different cores; thus, in its fault-free state, its performance is 4 times that of the reference OoO core. The curves reporting Cobra's performance are obtained by disabling faulty components at the hardware unit or service granularity (see Table 6.2 for a list of units and services). Note that under fault-free conditions, Cobra is outperformed by the CMP by more than a factor of 2, due to the overhead of setting up and managing the dynamic hardware configurations. However, as faults increase, the margin of benefit is reduced quickly with Cobra providing better performance after only 4 faults – corresponding roughly to 1 fault every 20M transistors.

Note that our performance evaluation of Cobra is fairly conservative, as we omitted a number of optimizations for single-threaded workloads as discussed in Section 5.6.2.

Online Fault Detection

Lastly, we evaluate the performance impact of deploying online fault detection mechanisms in Cobra. Table 6.3 compares fault coverage and performance impact of the techniques discussed in Section 6.5. Figure 6.10 compares the performance cost of each of the fault detection mechanisms relatively to a Cobra solution with no active fault detection. Note

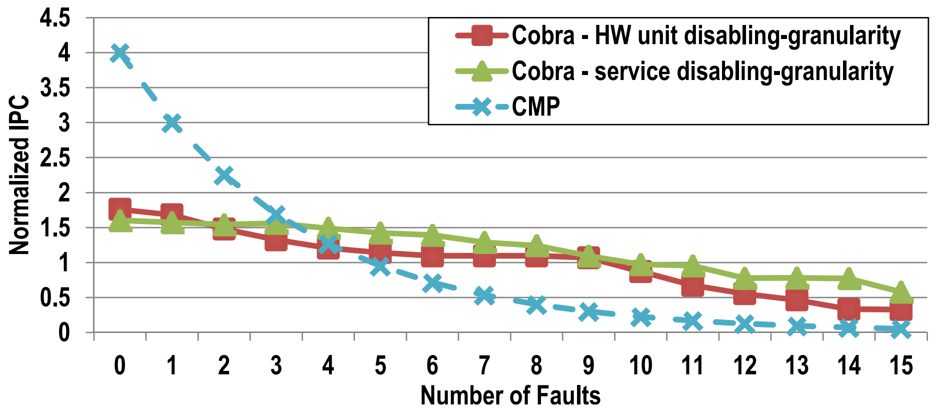


Figure 6.11 Cobra's performance degradation in presence of failures vs. a CMP. When measuring area-neutral throughput, Cobra outperforms the CMP beyond 4 faults.

how the full redundancy approach experiences the highest performance costs of 26% for single-process benchmarks and 61% for multiprogrammed ones. The reason for the smaller impact on single-process programs lies in the multiplicity of hardware units available that can be used to hide the additional computation required. Note that on multi-process benchmarks the cost is even above 50% – what one would expect when Cobra uses twice as many resources per bundle to implement dual redundancy: this is due to overhead in checking redundant results and resource overbooking.

The performance cost of selective redundancy is less steep: 19% and 26% for single- and multi- process benchmarks, respectively. For this solution we protected with dual redundancy only bundles including FPU and SIMD operations and mult/div instructions. The three outliers presenting a performance improvement are explained in a similar way to Figure 6.11: occasionally a small perturbation on available resources may lead to better local optima configurations.

Finally, the performance impact of periodic testing is only 3%. We implemented this solution by performing a periodic self-test of all hardware units (not yet known to be faulty) every 20M instructions – a typical interval for processor self-tests [41, 142]. The performance we measured is much better than that of similar approaches in pipelined architectures (we reported 16% in our experiments in Chapter 3). Such limited impact is due to the fact that in Cobra it is straightforward to take a unit off-line temporarily for testing purposes, without affecting execution on the rest of the system.

Mechanism	Detection Latency	Fault Coverage		Overhead	
		Permanent	Transient	CMP	Cobra
Full redundancy	≤ 100 cycles	100%	100%	50%	61%
Selective redundancy	$\leq 10M$ cycles	100%[130]	98%[80]	10%	26%
Online testing	$\leq 20M$ cycles	95.5%[142]	N/A	8%	3%

Table 6.3 Summary of the characteristics of the three fault detection mechanisms considered. We compare Cobra against an ideal CMP on fault detection latency, fault coverage and performance overhead.

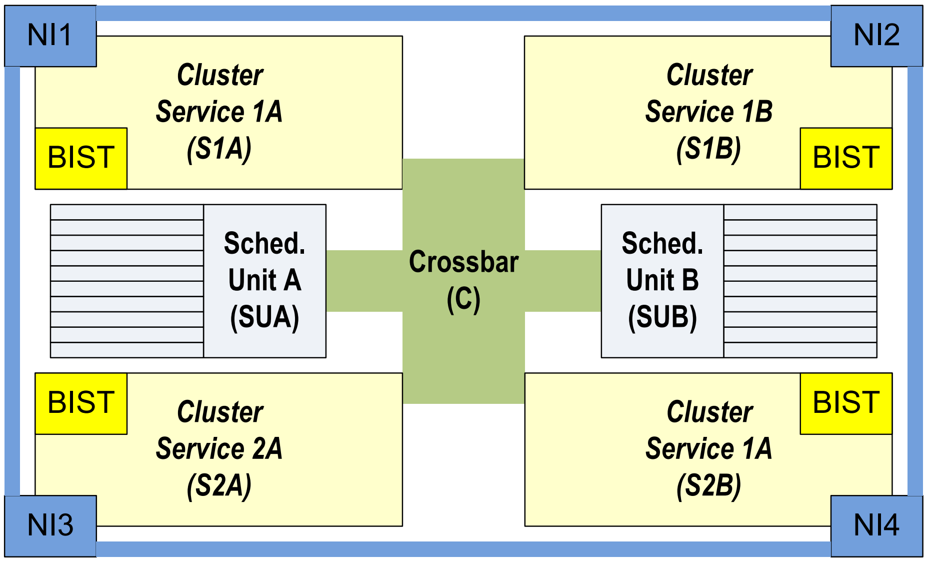


Figure 6.12 Example of Cobra configuration which partitions the ISA in two services. This model contains two copies of each service provider and two scheduling units. The control interconnect, which is needed to establish the hardware configurations, is shown in green, while the data interconnect, used to transmit instructions and operands, is shown in blue.

6.6.6 Reliability Analysis

In this section we study the reliability bottlenecks of Cobra, and evaluate which of its components must be strengthened in order to obtain a single-point-of-failure free system. With this goal, we rely upon a deductive failure analysis in which component failures are combined using Boolean logic. This methodology, called fault-tree analysis (FTA), is commonly adopted by reliability engineers to determine how single component failures propagate to trigger critical problems at the system level [104].

In order to study our design from a general point of view, we analyze a model of a simple, yet complete, Cobra architecture that provides two type of services to the instructions executing on the system. For the sake of simplicity, we call these services “1” and “2”, respectively. We show the system used in our case study in Figure 6.12. In this example, two hardware components can execute each service, and two scheduling units are available to synchronize and control program execution. Components exchange information through two communication subsystems: the control interconnect, shown in green in Figure 6.12, and the data interconnect, shown in blue.

We start by analyzing the events that would prevent our system from executing a program that relies on both services. We recognize that four distinct events could, separately, cause this critical failure: the disconnection of the control interconnect, the unavailability of at least one type of service provider, the unavailability of both scheduling units, and the disconnection of the data interconnect. To assess the low-level causes of these events, the remainder of this section analyzes each of them in more detail. We also report a graphic representation of our analysis in Figure 6.13. In this study we first assume that every fault is known a priori, and that a hardware component deemed faulty is disabled and will no longer provide any information about its state and availability to the rest of the system. Although we initially do not consider cases where an undetected byzantine fault silently corrupts system state and/or produces incorrect output, we discuss this scenario for the components that may be affected by these subtle failures.

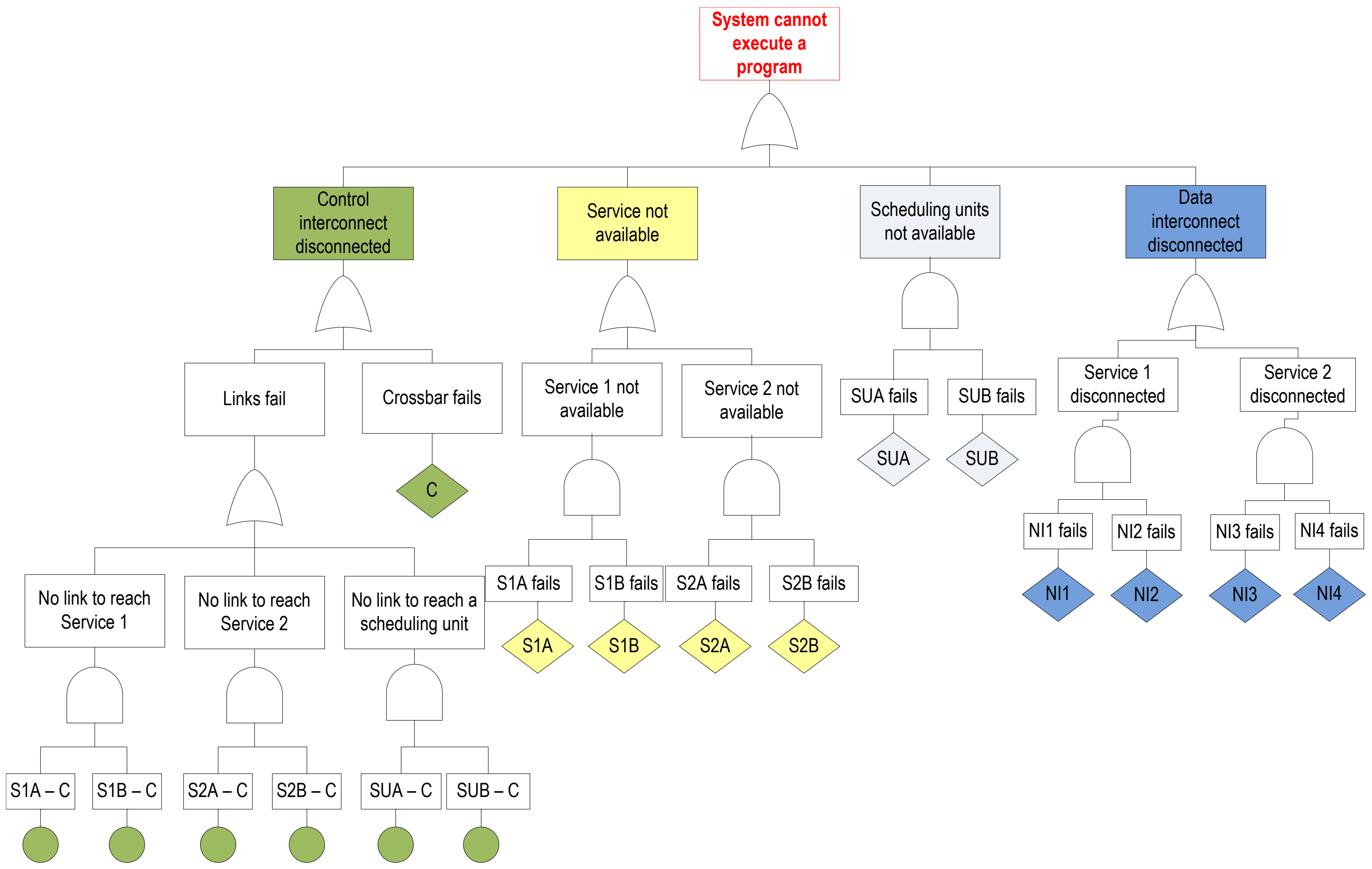


Figure 6.13 Fault tree analysis of the simple Cobra configuration adopted in our reliability study. We used a top-down methodology to assess how individual component failures affect the behavior of the entire architecture.

Disconnection of the Control Interconnect

All service providers and scheduling units are connected through a crossbar. In a fault-free system, each component can interact freely with any other one. The disconnection of the control interconnect prevents components from interacting with each other to establish a hardware configurations that can execute an instruction bundle. Two distinct cases could trigger this scenario:

1. all links connecting a type of service provider or the scheduling units fail;

2. the crossbar fails.

Since service providers and scheduling units are redundant, the first case can occur only when multiple failures affect all the links connecting the redundant copies of a hardware module (service 1, service 2, or the scheduling units). The second case requires more attention, since a single failure in the crossbar could endanger the functionality of the entire control interconnect, hence hindering Cobra's ability to execute any instruction bundle. With the goal of acquiring a better understanding of the reliability limitations of this component, we studied Vicis, a recent solution for reliable network-on-chips that can maintain performance even in the face of numerous hardware failures [55]. We are interested in the two techniques developed by Vicis to protect the crossbar from failing when subjected to permanent failures: i) the crossbar bypass bus and ii) error correcting codes. The first technique consists of an ancillary bus, parallel to the crossbar, which allows network packets to bypass a faulty crossbar and reach their destination. This solution provides a second, slower medium for the control interconnect, effectively trading off performance to maintain correct operation in case of hardware failures in the crossbar. The latter solution, reliance upon error correcting codes, allows crossbar datapath components to sustain a number of hardware faults without corrupting in-flight packets. Empirical evidence demonstrated that the combination of these two solutions protects a crossbar from single failures and enables a router to maintain performance even when it is subjected to a significant number of hardware defects [55]. Finally, users that can afford larger area budgets could replicate the entire crossbar, thus avoiding performance penalties in case of hardware failures.

Unavailability of Hardware Services

From Cobra's perspective, an ISA consists of the union of the services required by all its instructions. In our design, a program is always able to successfully execute as long as the working hardware clusters can, in aggregate, perform all the services required by the program's instructions. Therefore, a scenario that could prevent our design from executing a program is the unavailability of all providers of a determined type of service. In our case-study, this is possible when two or more failures affect both S1A and S1B or both S2A and S2B, and neither case would be possible due to a single hardware failure.

Nevertheless, service providers are very complex hardware components, and we have already shown in Chapter 3 that undetected hardware failures may silently corrupt the output of a hardware module. In our previous experiments we assumed that 100% of the faults manifesting at runtime can be detected through testing techniques orthogonal to our architecture [41, 110]. Still, it is possible that faults may also occur in the self-test logic embedded in every hardware module – this is an issue that all fault-tolerant designs must face. Since the self-testing units are local to each component in Cobra, in our experiments we assumed that a fault in this logic would cause the corresponding component to be marked as non-testable and therefore diagnosed as faulty. However, it is important to consider the scenario when hardware defects might also affect the self-test logic. Four cases are possible in this circumstance:

1. the service provider is not faulty but its broken self-test circuitry diagnoses it as so;
2. the service provider is not faulty and its broken self-test circuitry diagnoses it as non-faulty;
3. the service provider is faulty and its broken self-test diagnoses it as so;
4. the service provider is faulty but its broken self-test diagnoses it as non-faulty.

In the first case, the fault in the self-test circuitry is effectively reported as a fault in the service provider, therefore causing the entire unit to shut down. While, practically, the hardware unit is still functional, it is conservatively disabled due to its faulty self-test logic – our previous experiments modeled this scenario. Cases two and three do not alter the normal behavior of the system, since the diagnostic results provided by the self-test logic are accurate. The fourth case is the most interesting: even though the service provider is faulty, its fault is masked by the crippled self-test circuitry. This case can only occur when we consider that multiple failures may be present in the system, and the only way to avoid this occurrence is to strengthen the self-test circuitry. Because our solution relies heavily on these self-test mechanisms, it is important to discuss how to ensure the integrity of the self-test logic. All manufacturing defects present in self-test units can be diagnosed after fabrication. Built-in-self-test units are power-gated during normal operations to protect them from wear out. Therefore, since the self-test logic is used much more infrequently than other hardware modules, the occurrence of runtime failures is unlikely [55]. Nevertheless, users concerned about these failures can protect the relatively small self-test logic through traditional reliability mechanisms such as dual-modular redundancy [143].

Unavailability of Scheduling Units

Scheduling units allow Cobra components to generate hardware configurations that can execute instruction bundles. Each scheduling unit is composed of two components: i) an array of memory elements and ii) sequential logic to assign and schedule hardware components to in-flight instruction bundles.

The memory elements consist of an array of entries, each of which can store data necessary to manage the execution of an instruction bundle: the list of services required, hardware configurations, operand tags, program counters, and other relevant information. These entries are redundant and their memory elements are protected from hardware failures through ECC.

Scheduling units also contain a modest amount of logic to match component proposals with program demands, and initiate or respond to these communications. The hardware required to accomplish these tasks is rather simple, since it consists of decoding logic, bit masks, and counters that trigger an ordered sequence of events. Therefore, this circuitry can be effectively protected through dual or triple module-redundancy [147].

Disconnection of the Data Interconnect

Lastly, there is also the possibility that service providers may not communicate with each other due to failures in the data interconnect. This could be fatal when none of the providers of a needed service can receive or transmit instructions or operands to the others. In our case study, this happens when two distinct failures partition the system horizontally, either breaking both NI1 and NI2, thus disconnecting both service 1 providers, or because both NI3 and NI4 fail, disconnecting all service 2 providers.

It is worth noting that the interface connecting a service provider might experience failures that do not completely hinder its functionality, but only sporadically corrupt its correct operation. Previous works analyzed this type of failures, and proposed to overcome these scenario through: i) the exploitation of the natural redundant storage available; ii) replication of the few most sensitive parts of the interface [57]. Beyond avoiding such byzantine failures, these techniques would also enable a network interface to survive a number of permanent faults, therefore extending its lifetime.

It is worth noting that larger Cobra configurations may require higher connectivity to increase the total bandwidth available to the service providers. In this case, multiple network interfaces would be connected through routers to form a full-fledged network-on-chip. In such design, each provider would be paired with a network interface and a switch, and both these components could be enhanced with the reliability features detailed above [55, 57].

Reliability Analysis Summary

In summary, this section presented a reliability analysis of our architecture and studied which component failures may prevent the system from executing a program. We recognize that the only critical component in the system is the crossbar connecting all components to form the control interconnect, since a single fault in its logic could jeopardize the functioning of the entire machine. Hence, we proposed two solutions that can strengthen its logic and protect it from hardware failures. While all other components can be made redundant and therefore expendable, we also analyzed all possible events triggered by defective self-test circuitry and proposed mechanisms to overcome them.

6.7 Summary

In this Chapter we presented Cobra, a holistic, reliable, adaptable distributed-control architecture that brings together all the solutions discussed in this dissertation. Furthermore, we also demonstrated that this comprehensive design can greatly enhance a system's reliability and adaptability, without jeopardizing fault-free performance. We also studied the reliability of our new architecture, and discussed solutions that can protect its critical components from hardware failures.

Our design enables the distributed architecture developed in this dissertation to take advantage of the low-cost fault detection mechanisms presented in Chapter 3. In addition, Cobra proposes a novel memory organization and relies on the algorithm developed in Cardio in Chapter 4 to maintain adaptability without hindering instruction execution runtime. As a result, Cobra not only provides high system dependability and adaptability, but also greatly boosts the performance of distributed-control architectures when running multiprogrammed workloads. By analyzing Cobra's reliability, we found that it outperforms a traditional CMP design beyond the occurrence of 4 faults – corresponding roughly to 1 fault per 20M transistors in our setup, and that the performance cost of online fault detection is only 3%. We also systematically analyzed the reliability bottlenecks of our design, and proposed solutions to strengthen the components that, if faulty, may endanger system functionality.

Chapter 7
Conclusions

This dissertation presented a number of solutions to build adaptive and distributed architectures that can address the challenges facing future generations of computer systems. Transistor features are rapidly shifting. Future silicon manufacturing processes are expected to integrate a massive number of tiny and, unfortunately, fragile switching devices. While these new technologies could enable higher performance and lower energy consumption, current microprocessor designs cannot take advantage of them. Indeed, current technological trends have already indicated three new barriers that will limit advancements of current computer architectures: i) increasing fragility of their switching elements, ii) challenges associated with managing the heterogeneous components necessary for efficient computing, and iii) lack of design modularity.

Due to the scenario imposed by these three constraints, current architectures and design methodologies are no longer adequate. This creates the compelling demand for novel computer architectures capable of tackling these challenges. With these goals, this research has proposed various solutions that address these impending issues by promoting reliability, adaptability, and modularity as principal design foci. The previous chapters presented and analyzed a number of solutions that overcome these challenges, enabling future computers to provide more reliable performance and unlocking opportunities to increase computational efficiency.

7.1 An Adaptive, Reliable, and Distributed Architecture

The objective of this thesis is to develop novel architectures designed to overcome all three challenges imposed by future semiconductor technologies. Chapter 3 addresses the increasing need for fault-tolerant computing with an original adaptive mechanism to diagnose hardware failures in processor components. Our solution, called **A2Testing**, was driven by the insights obtained through analysis of the fault injections performed with **CrashTest**. We injected thousands of hardware failures into an industrial-grade microprocessor running unmodified software applications. Such experiments revealed that a large portion of hardware faults do not alter software execution, especially when applications do not directly exercise crippled hardware components. In light of this observation, we developed an adaptive framework that optimizes testing routines and improves hardware testing efficiency by monitoring hardware utilization and focusing testing efforts on the components that, if faulty, are more likely to have corrupted a software application.

Chapter 4 focuses on providing mechanisms to achieve adaptability and to solve the challenges related to managing the heterogeneous components necessary for efficient computing. We first study the advantages provided by specialized hardware through a practical case-study that targets hardware specialized to execute computer vision algorithms. Our experiments show that hardware specialization can improve processor efficiency up to 27 times compared to a baseline embedded processor. While specialized hardware can greatly improve hardware performance and reduce power consumption, a system must only dynamically activate specialized components when needed. With the goal of providing this capability, we developed **Cardio**. This software-based self-introspective mechanism relies on message broadcasting to quickly reconfigure a chip to match software demands and system requirements. While Cardio's response time is very limited, only a few thousand cycles per event, we also demonstrate that it scales well and affects performance for as little as 3.5%.

The third and last pillar of this dissertation is the modular architecture presented in Chapter 5: **Viper**. Modularity is a system property that allows a design to scale in size and performance while maintaining flexibility and reconfigurability. Viper is a fully distributed microarchitecture that breaks apart the classic concept of a hard-wired pipeline, dissolving instead the processors components into a sea of redundant hardware clusters. These units, which are connected via an on-chip network, are dynamically grouped into virtual pipelines capable of executing instructions. Virtual pipelines are generated to match the execution needs of the upcoming instructions through a distributed control mechanism that requires no software changes.

Finally, Chapter 6 completes this research with the presentation of **Cobra**, the holistic design that coheres all the contributions presented in this thesis. Similarly to Viper, Cobra organizes hardware components into a reconfigurable fabric of small, state-less units. This design targets highly parallel workloads and promotes reliability, adaptability, and modularity as top-priority design foci. Cobra relies on A2Testing to tolerate a large number of defects and its performance gracefully degrades as faults accumulate in its hardware. Finally, we rely on Cardio to manage dynamic hardware reconfiguration. While Chapter 4 presented a software-based technique for adaptive computing, Cobra deploys this protocol in hardware. In our experiments we show that this system provides reliable and flexible computing with very low overhead. Before exploring possible future extensions of our work, we briefly detail the individual contributions of this thesis.

7.1.1 Reliability

Unmanaged hardware failures are extremely dangerous for computer users. On current systems, these events sporadically cause system crashes or corrupt program output. While reliability is only a secondary requirement for most computers, technology experts warn that it will soon become a primary concern due to the degradation in transistor robustness foreseen in future technology nodes. This thesis proposes a novel low-cost mechanism to diagnose hardware failures that is based on the periodic assessment of the health of each component.

In order to understand the limitations of current microprocessors, this dissertation first investigates the repercussions of hardware failures on modern industrial-grade designs. Prior methodologies for fault analysis could not be employed for this investigation since they are either detailed but slow or fast but inaccurate. In order to obtain an infrastructure that could perform accurate fault analysis without compromising performance, we developed a complete, customizable FPGA-based framework: **CrashTest**. In our evaluation we measured that such infrastructure can perform reliability analysis six orders of magnitude faster than software-based fault injectors. Combining the accuracy and the performance offered by this system enabled us to analyze the behavior of a complex microprocessor design, such as the Sun OpenSPARC T1, when it is subjected to hardware failures.

These analyses have shed light on the effects of software applications executing on faulty hardware. First, we observed that programs rarely stress all hardware uniformly, and microprocessor's utilization varies even within the execution of one application. Second, we noticed that program execution is often divided in phases, each of which typically relies on only a few hardware components. These observations led to the development of a low-cost adaptive reliability technique, called **A2Testing**, which leverages application behavior to tune runtime self-tests and achieve higher efficiency and better protection against hardware failures. Periodic hardware tests are organized per functional module, and the activity of each module is monitored through counters. These counters are then used to selectively activate self-tests only on the hardware features that the software exercised.

7.1.2 Adaptability

We propose hardware adaptability as the solution to allocate resources and reconfigure the system to work around hardware failures. Dynamic resource allocation is a fundamental challenge imposed by the new design methodology intended to tackle the dark silicon era. Since general purpose processors are inherently inefficient, a number of researchers have advocated the deployment of hardware accelerators specialized in speeding up particular software functions. The main challenge in the adoption of this execution paradigm is the dynamic management of these various hardware components. In order to save power, only those needed to complete the current applications should be activated. Adaptability can also improve reliability, since faulty hardware modules can be turned off so the rest of the system can work around them. In this thesis we envision future computer architectures as dynamic machines that can conform their operations to changing program demands, environmental characteristics, and hardware availability.

In order to provide this characteristic, this thesis presented **Cardio**, a software-based solution that automatically manages computer resources. All components in a Cardio-enabled chip dynamically exchange information about their condition and utilization. These diagnostic messages are collected by a distributed resource manager, which reconfigures the design to match hardware functionalities and application needs without relying on a centralized controller. In this dissertation we show that the reactivity of this design in responding to changes is as low as 20,000 cycles, and we measure its performance impact to be as low as 3.5%.

7.1.3 Modularity

All modern state-of-the-art processor designs are characterized by extremely high non-recurring engineering costs. As future semiconductor technologies are expected to increase component integration even further, this problem can only worsen. The two design characteristics that have the largest impact on the overall engineering effort are: i) the tight interconnection between hardware components and ii) the extensive control logic needed to synchronize and manage the interaction among a processor's modules. To reduce both these costs, our research developed a microarchitectural solution that promotes simplicity through modularity as a key design guideline.

Viper is our solution to overcome this issue. Modular systems can be designed faster than non-modular ones because engineers are able to focus their attention on developing and testing individual components. Our solution organizes hardware modules into a reconfigurable fabric of small, state-less units. Each module can accomplish one or more services towards the execution of a portion of a program. Such a design greatly simplifies hardware organization, since each component is autonomous and the number of available service providers does not affect the operations of the rest of the system.

The original execution model of the modular design developed here constitutes the backbone of our complete final adaptive and distributed architecture, **Cobra**. This final design targets highly parallel workloads and integrates all the contributions of this thesis into a comprehensive distributed architecture. A design such as this enables unprecedented robustness and adaptability, as demonstrated in this dissertation. Hence, it has the opportunity to reduce overall engineering cost, extend system lifetime, and leverage specialized functional units to increase computational efficiency.

7.2 Future Research Directions

The novel distributed execution paradigm explored in this thesis opens new frontiers for research in computer architecture. There are a number of new and stimulating research topics that can stem from this research:

What is the best dynamic configuration granularity for a distributed architecture?

Our novel way to approach computer architectures enables unmatched flexibility in terms of runtime hardware adaptability. This execution model supports components of any size and with any capability, and offers very short reconfiguration times. Hence, such architecture can benefit a large range of applications and design constraints. Given all the possible design choices laid out by our distributed execution model, we believe that architects will find a plethora of applications and systems that can take advantages of these distributed architectures.

Which characteristics of a distributed architecture can be exposed to software developers?

In our research we did not change how programs perceive the underlying hardware, therefore allowing unmodified legacy software to run on our architecture. Nevertheless, distributed architectures introduce some interesting features that could be presented to software programmers. For instance, the fabric connecting the different hardware units can be exploited to enable fast low-cost inter-thread communication. Another example of a semantic change of programs that is worth investigating is the relaxation of the strict sequentiality of memory operations in order to enhance hardware efficiency and improve instruction parallelism. These are just two examples of how the interface between software and a distributed architecture can be relaxed to decrease power consumption, increase performance, or limit hardware complexity.

What are the best ways to achieve high performance on a distributed architecture?

Even though our research presents several solutions to improve the proposed design, deeper analyses on distributed architectures are needed to identify more opportunities and better optimize performance and efficiency. A multitude of hardware resource assignment policies can be developed for this purpose, and one example of tradeoff worth investigating is between fairness and performance.

7.3 Summary

This dissertation has presented a number of solutions to build adaptable and distributed computer architectures that can address the three obstacles that undermine the adoption of future semiconductor technologies: increasing transistor fragility, challenges connected to the management of heterogeneous components, and lack of design modularity. In this work we have shown that these designs are effective in providing reliability, adaptability, and modularity to a hardware system. Beyond these already significant achievements, our architectures offer a revolutionary opportunity to rethink computer organization in broader terms, offering further advantages that can extend far beyond the ones presented in this dissertation.

Bibliography

[1] E. Ackerman. Google gets in your face. IEEE Spectrum, 2013. `http://spectrum.ieee.org/consumer-electronics/gadgets/google-gets-in-your-face`.

[2] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *International Symposium on Computer Architecture*, jun. 2000.

[3] A. Alaghi, N. Karimi, M. Sedghi, and Z. Navabi. Online NoC switch fault detection and diagnosis using a high level fault model. In *International Symposium on Defect and Fault Tolerance*, sep. 2007.

[4] A. R. Alameldeen, I. Wagner, Z. Chishti, W. Wu, C. Wilkerson, and S.-L. Lu. Energy-efficient cache design using variable-strength error-correcting codes. In *International Symposium on Computer Architecture*, jun. 2011.

[5] Amazon Web Services Discussion Forums. Intermittent internal connectivity failures. World Wide Web, 2008. `http://developer.amazonwebservices.com/connect/thread.jspa?threadID=21401&tstart=15`.

[6] A. Ansari, S. Feng, S. Gupta, and S. Mahlke. Necromancer: enhancing system throughput by animating dead cores. In *International Symposium on Computer Architecture*, jun. 2010.

[7] Arvind and D. Culler. Dataflow architectures. *Annual Review of Computer Science*, 1(1):225–253, jun. 1986.

[8] A. Baghdadi, D. Lyonnard, N.-E. Zergainoh, and A. Jerraya. An efficient architecture model for systematic design of application-specific multiprocessor SoC. In *Design Automation and Test in Europe*, mar. 2001.

[9] P. Balaji, W. Mahmoud, E. Ososanya, and K. Thangarajan. Survey of the counterflow pipeline processor architectures. In *Proceedings of the Southeastern Symposium on System Theory*, mar. 2002.

[10] C. Baldwin and K. Clark. *Design Rules*, volume 1 of *MIT Press Books*. The MIT Press, aug. 2000.

[11] C. Barnes, D. Fleetwood, D. Shaw, and P. Winokur. Post irradiation effects (PIE) in integrated circuits [MOS]. In *European Conference on Radiation and its Effects on Devices and Systems*, sep. 1991.

[12] W. Bartlett and L. Spainhower. Commerical fault tolerance: a tale of two systems. *IEEE Trans. on Dependable and Secure Computing*, 1, jan.-mar. 2004.

[13] M. Bass and C. Christensen. The future of the microprocessor business. *Spectrum, IEEE*, 39(4):34 –39, apr. 2002.

[14] R. Baumann. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In *International Electron Devices Meeting*, dec. 2002.

[15] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - processor: a 64-core SoC with mesh interconnect. In *International Solid-State Circuits Conference*, feb. 2008.

[16] A. Benso, A. Bosio, P. Prinetto, and A. Savino. An on-line software-based self-test framework for microprocessor cores. In *International Conference on Design and Test of Integrated Systems in Nanoscale Technology*, sep. 2006.

[17] B. Bentley. Validating the Intel Pentium 4 microprocessor. In *Design Automation Conference*, jun. 2001.

[18] B. Bentley. Validating a modern microprocessor. In *International Conference on Computer Aided Verification*, jul. 2005.

[19] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, aug. 2011.

[20] N. Binkert, A. Davis, N. Jouppi, M. McLaren, N. Muralimanohar, R. Schreiber, and J. Ahn. Optical high radix switch design. *IEEE Micro*, 32(3):100–109, may 2012.

[21] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, nov.-dec. 2005.

[22] S. Borkar. Thousand core chips: a technology perspective. In *Design Automation Conference*, jun. 2007.

[23] S. Borkar and A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, apr. 2011.

[24] S. Borkar, N. Jouppi, and P. Stenstrom. Microprocessors in the era of terascale integration. In *Design Automation and Test in Europe*, mar. 2007.

[25] P. Bose, D. Albonesi, and D. Marculescu. Guest editors introduction: power and complexity aware design. *IEEE Micro*, 23(5):8–11, sep.-oct. 2003.

[26] F. Bower, P. Shealy, S. Ozev, and D. Sorin. Tolerating hard faults in microprocessor array structures. In *International Conference on Dependable Systems and Networks*, jun. 2004.

[27] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, jul. 2004.

[28] M. Bushnell and V. Agarwal. *Essentials of Electronic Testing for Digital Memory and Mixed-Signal VLSI Circuits*. Springer, 2000.

[29] A. Chang and W. Dally. Explaining the gap between ASIC and custom power: a custom perspective. In *Design Automation Conference*, jun. 2005.

[30] L. Chen and T. Pinkston. NoRD: node-router decoupling for effective power-gating of on-chip routers. In *International Symposium on Microarchitecture*, dec. 2012.

[31] L. Chen, S. Ravi, A. Raghunathan, and S. Dey. A scalable software-based self-test methodology for programmable processors. In *Design Automation Conference*, jun. 2003.

[32] X. Chen, Y. Yang, G. Gopalakrishnan, and C.-T. Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Proceedings of the Formal Methods in Computer Aided Design*, nov. 2006.

[33] Y.-K. Chen. Challenges and opportunities of Internet of things. In *Asia and South Pacific Design Automation Conference*, feb. 2012.

[34] G. Chrysos. Knights Corner, Intel's first many integrated core (MIC) architecture product. In *Hot Chips*, aug. 2012.

[35] J. Clemons, A. Jones, R. Perricone, S. Savarese, and T. Austin. EFFEX: an embedded processor for computer vision based feature extraction. In *Design Automation Conference*, jun. 2011.

[36] J. Clemons, A. Pellegrini, S. Savarese, and T. Austin. EVA: an efficient vision architecture for mobile systems. In *International Conference on Compilers Architectures and Synthesis for Embedded Systems*, oct. 2013.

[37] J. Clemons, H. Zhu, S. Savarese, and T. Austin. MEVBench: a mobile computer vision benchmarking suite. In *International Symposium on Workload Characterization*, nov. 2011.

[38] K. Compton and S. Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, jun. 2002.

[39] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman. Architecture support for accelerator-rich CMPs. In *Design Automation Conference*, jun. 2012.

[40] K. Constantinides, J. Blome, S. Plaza, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. BulletProof: a defect-tolerant CMP switch architecture. In *International Symposium on High-Performance Computer Architecture*, feb. 2006.

[41] K. Constantinides, O. Mutlu, T. M. Austin, and V. Bertacco. Software-based online detection of hardware defects: mechanisms, architectural support, and evaluation. In *International Symposium on Microarchitecture*, dec. 2007.

[42] E. Cota, P. Mantovani, M. Petracca, M. Casu, and L. Carloni. Accelerator memory reuse in the dark silicon era. *IEEE Computer Architecture Letters*, 99, jul. 2012.

[43] E. Czeck and D. Siewiorek. Effects of transient gate-level faults on program behavior. In *International Symposium on Fault-Tolerant Computing*, jun. 1990.

[44] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah. A 22nm IA multi-CPU and GPU system-on-chip. In *International Solid-State Circuits Conference*, feb. 2012.

[45] A. DeOrio, D. Fick, V. Bertacco, D. Sylvester, D. Blaauw, J. Hu, and G. K. Chen. A reliable routing architecture and algorithm for NoCs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 31(5):726–739, apr. 2012.

[46] A. DeOrio, J. Li, and V. Bertacco. Bridging pre- and post-silicon debugging with BiPeD. In *International Conference on Computer-Aided Design*, nov. 2012.

[47] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tzchanz, V. Erraguntla, N. Borkar, V. De, and S. Borkar. Within-die variation-aware dynamic-voltage-frequency-scaling with optimal core allocation and thread hopping for the 80-core TeraFLOPS processor. *IEEE Journal of Solid-State Circuits*, 46(1):184–193, jan. 2011.

[48] M. Dimitrov and H. Zhou. Unified architectural support for soft-error protection or software bug detection. In *International Conference on Parallel Architectures and Compilation Techniques*, sep. 2007.

[49] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *International Reliability Physics Symposium*, apr. 2011.

[50] T. Dorta, J. Jimenez, J. Martin, U. Bidarte, and A. Astarloa. Overview of FPGA-based multiprocessor systems. In *International Conference on Reconfigurable Computing and FPGAs*, dec. 2009.

[51] N. Durrant and R. Blish. Semiconductor device reliability failure models. World Wide Web, 2000. `http://www.sematech.org/`.

[52] H. Esmaeilzadeh, E. Blem, R. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *International Symposium on Computer Architecture*, jun. 2011.

[53] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Shoestring: probabilistic soft error reliability on the cheap. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, mar. 2010.

[54] A. Ferrari, A. Filipi-Martin, and S. Viswanathan. The NAS parallel benchmark kernels in MPL. Technical Report CS-95-39, NASA, dec. 1995.

[55] D. Fick, A. DeOrio, J. Hu, V. Bertacco, D. Blaauw, and D. Sylvester. Vicis: a reliable network for unreliable silicon. In *Design Automation Conference*, jul. 2009.

[56] D. Fick, R. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, T. Mudge, D. Blaauw, and D. Sylvester. Centip3De: a cluster-based NTC architecture with 64 ARM Cortex-M3 cores in 3D stacked 130 nm CMOS. *IEEE Journal of Solid-State Circuits*, 48(1):104 –117, dec. 2013.

[57] L. Fiorin, L. Micconi, and M. Sami. Design of fault tolerant network interfaces for NoCs. In *Euromicro Conference on Digital System Design*, sep. 2011.

[58] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Conference on Fundamentals of Computation Theory*, aug. 1983.

[59] M. Flatte. Spintronics. *IEEE Transactions on Electron Devices*, 54(5):907 –920, may 2007.

[60] S. Frehse, G. Fey, and R. Drechsler. A better-than-worst-case robustness measure. In *Workshop on Design and Diagnostics of Electronic Circuits and Systems*, apr. 2010.

[61] G. Gielen, P. De Wit, E. Maricau, J. Loeckx, J. Martín-Martínez, B. Kaczer, G. Groeseneken, R. Rodríguez, and M. Nafría. Emerging yield and reliability challenges in nanometer CMOS technologies. In *Design Automation and Test in Europe*, mar. 2008.

[62] D. Gizopoulos, M. Psarakis, S. Adve, P. Ramachandran, S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera. Architectures for online error detection and recovery in multicore processors. In *Design Automation and Test in Europe*, mar. 2011.

[63] B. Goldwater and G. Hart. Recent false alerts from the nation's missile attack warning system. Technical report, United States Senate, 1980.

[64] R. Golla and P. Jordan. T4: a highly-threaded, server-on-a-chip with native support for heterogenous computing. In *Hot Chips*, aug. 2011.

[65] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante. Soft-error detection using control flow assertions. In *International Symposium on Defect and Fault Tolerance*, nov. 2003.

[66] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *International Symposium on High-Performance Computer Architecture*, feb. 1998.

[67] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The GreenDroid mobile application processor: an architecture for silicon's dark future. *IEEE Micro*, 31(2):86 –95, mar.-apr. 2011.

[68] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, and J. Haid. Modular fault injector for multiple fault dependability and security evaluations. In *Euromicro Symposium on Digital Systems Design*, sep. 2011.

[69] R. Guo, S. Mitra, E. Amyeen, J. Lee, S. Sivaraj, and S. Venkataraman. Evaluation of test metrics: stuck-at, bridge coverage estimate and gate exhaustive. In *Proceedings of the VLSI Test Symposium*, apr. 2006.

[70] P. Gupta and A. Kahng. Manufacturing-aware physical design. In *International Conference on Computer-Aided Design*, nov. 2003.

[71] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. Adaptive online testing for efficient hard fault detection. In *International Conference on Computer Design*, oct. 2009.

[72] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. StageWeb: interweaving pipeline stages into a wearout and variation tolerant CMP fabric. In *International Conference on Dependable Systems and Networks*, jun. 2010.

[73] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. The StageNet fabric for constructing resilient multicore systems. In *International Symposium on Microarchitecture*, nov. 2008.

[74] S. Gupta, S. Feng, A. Ansari, and S. Mahlke. Erasing core boundaries for robust and configurable performance. In *International Symposium on Microarchitecture*, dec. 2010.

[75] G.Yoh and F. Najm. A statistical model for electromigration failures. In *International Symposium on Quality Electronic Design*, mar. 2000.

[76] T.-J. Ha, D. Akinwande, and A. Dodabalapur. Hybrid graphene/organic semiconductor field-effect transistors. *Applied Physics Letters*, 101(3):033309 –033309–3, jul. 2012.

[77] T. Hacker, F. Romero, and C. Carothers. An analysis of clustered failures on large supercomputing systems. *Journal of Parallel Distributed Computing*, 69(7):652–665, jul. 2009.

[78] T. Halfhill. Intel's tiny Atom. *Microprocessor Report*, apr. 2008.

[79] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Comp.*, 30(9):79–85, sep. 1997.

[80] S. Hari, S. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *International Conference on Dependable Systems and Networks*, jun. 2012.

[81] S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. Adve. mSWAT: low-cost hardware fault detection and diagnosis for multicore systems. In *International Symposium on Microarchitecture*, dec. 2009.

[82] A. Hartstein and T. Puzak. The optimum pipeline depth for a microprocessor. In *International Symposium on Computer Architecture*, may 2002.

[83] J. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, sep. 2006.

[84] T. Hey. Quantum computing: an introduction. *Computing Control Engineering Journal*, 10(3):105 –112, jun. 1999.

[85] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein. Scaling, power, and the future of CMOS. In *IEEE International Electron Devices Meeting, 2005.*, dec. 2005.

[86] M. Hrishikesh, D. Burger, S. Keckler, P. Shivakumar, N. Jouppi, and K. Farkas. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *International Symposium on Computer Architecture*, may 2002.

[87] B. Huang, A. Schmidt, A. Mendon, and R. Sass. Investigating resilient high performance reconfigurable computing with minimally-invasive system monitoring. In *International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, nov. 2010.

[88] C.-C. Huang and A. Kusiak. Modularity in design of products and systems. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 28(1):66–77, jan. 1998.

[89] M. Huebscher and J. McCann. A survey of autonomic computing–degrees, models, and applications. *ACM Computing Surveys*, 40(3):7:1–7:28, aug. 2008.

[90] IBM Corp. IBM 1961 BRL report. World Wide Web, 2013. `http://www.ed-thelen.org/comp-hist/BRL61-ibm1401.html`.

[91] Intel Corp. Intel processor comparison. World Wide Web, 2012. `http://www.intel.com/content/www/us/en/processor-comparison/compare-intel-processors.html`.

[92] Internet World Stats. Internet users in the world by years. World Wide Web, 2012. `http://www.internetworldstats.com/`.

[93] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture*, jun. 2007.

[94] M. Jacome and G. de Veciana. Design challenges for new application specific processors. *Design Test of Computers, IEEE*, 17(2):40–50, apr.-jun. 2000.

[95] P. Jamieson, W. Luk, S. Wilton, and G. Constantinides. An energy and power consumption analysis of FPGA routing architectures. In *International Conference on Field-Programmable Technology*, dec. 2009.

[96] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *International Symposium on High-Performance Computer Architecture*, feb. 2003.

[97] D. Josephson. The good, the bad, and the ugly of silicon debug. In *Design Automation Conference*, jul. 2006.

[98] E. Karl, D. Blaauw, D. Sylvester, and T. Mudge. Reliability modeling and management in dynamic microprocessor-based systems. In *Design Automation Conference*, jul. 2006.

[99] A. Kent. A Texas Instruments application report: MOS programmable logic arrays. Bulletin CA-158., oct. 1970.

[100] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. Keckler. Composable lightweight processors. In *International Symposium on Microarchitecture*, dec. 2007.

[101] A. KleinOsowski and D. Lilja. MinneSPEC: a new SPEC benchmark workload for simulation-based computer architecture research. *IEEE Computer Architecture Letters*, 1, jan.-dec. 2002.

[102] J. Kurose and K. Ross. *Computer Networking: a Top-Down Approach*. Addison-Wesley Publishing Company, USA, 5th edition, 2009.

[103] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, jul. 1982.

[104] W. Lee, D. Grosh, F. Tillman, and C. Lie. Fault tree analysis, methods, and applications – a review. *IEEE Transactions on Reliability*, R-34(3):194–203, aug. 1985.

[105] M.-L. Li, P. Ramachandran, R. Karpuzcu, S. Hari, and S. Adve. Accurate microarchitecture-level fault modeling for studying hardware faults. In *International Symposium on High-Performance Computer Architecture*, feb. 2009.

[106] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *International Conference on Dependable Systems and Networks*, jun. 2008.

[107] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient systems design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, mar. 2008.

[108] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *International Symposium on Microarchitecture*, jun. 2009.

[109] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *International Symposium on High-Performance Computer Architecture*, feb. 2007.

[110] Y. Li, S. Makar, and S. Mitra. CASP: concurrent autonomous chip self-test using stored test patterns. In *Design Automation and Test in Europe*, mar. 2008.

[111] Y. Li, O. Mutlu, and S. Mitra. Operating system scheduling for efficient online self-test in robust systems. In *International Conference on Computer-Aided Design*, nov. 2009.

[112] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: a high-performance DSP architecture for software-defined radio. *Micro, IEEE*, 27(1):114 –123, jan.-feb. 2007.

[113] A. Ling, D. P. Singh, and S. D. Brown. FPGA technology mapping: a study of optimality. In *Design Automation Conference*, jun. 2005.

[114] I. Loi, F. Angiolini, and L. Benini. Synthesis of low-overhead configurable source routing tables for network interfaces. In *Design Automation and Test in Europe*, apr. 2009.

[115] P. Lucassen and J. Udding. On the correctness of the Sproull counterflow pipeline processor. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, mar. 1996.

[116] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *International Conference on Programming Language Design and Implementation*, jun. 2005.

[117] T. May and M. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, jan. 1979.

[118] J. McCalpin. STREAM: sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 2007. `http://www.cs.virginia.edu/stream/`.

[119] H. McIntyre, S. Arekapudi, E. Busta, T. Fischer, M. Golden, A. Horiuchi, T. Meneghini, S. Naffziger, and J. Vinh. Design of the two-core x86-64 AMD "Bulldozer" module in 32 nm SOI CMOS. *IEEE Journal of Solid State Circuits*, 47(1):164 –176, jan. 2012.

[120] M. Mehrara, M. Attariyan, S. Shyam, K. Constantinides, V. Bertacco, and T. Austin. Low-cost protection for SER upsets and silicon defects. In *Design Automation and Test in Europe*, apr. 2007.

[121] D. Menasce' and J. Kephart. Guest editors' introduction: autonomic computing. *IEEE Internet Computing*, 11(1):18–21, jan.-feb. 2007.

[122] G. Michelogiannakis, D. Pnevmatikatos, and M. Katevenis. Approaching ideal NoC latency with pre-configured routes. In *International Symposium on Networks-on-Chip*, may 2007.

[123] M. Miller, K. Janik, and S.-L. Lu. Non-stalling counterflow architecture. In *International Symposium on High-Performance Computer Architecture*, feb. 1998.

[124] S. Mitra, M. Zhang, T. Mak, N. Seifert, V. Zia, and K. S. Kim. Logic soft errors: a major barrier to robust platform design. In *International Test Conference*, nov. 2005.

[125] M. Muller, K. Kalyanasundaram, G. Gaertner, W. Jones, R. Eigenmann, R. Lieberman, M. VanWaveren, and B. Whitney. SPEC HPG benchmarks for high-performance systems. *Journal of High Performance Computing and Networking*, pages 162–170, jan. 2004.

[126] S. Murali, T. Theocharides, N. Vijaykrishnan, M. Irwin, L. Benini, and G. De Micheli. Analysis of error recovery schemes for networks on chips. *IEEE Design Test of Computers, IEEE*, 22(5):434 – 442, sep.-oct. 2005.

[127] E. Musoll. Mesh-based many-core performance under process variations: a core yield perspective. *ACM SIGARCH Computer Architecture News*, pages 27–34, sep. 2009.

[128] P. Neumann. Principled assuredly trustworthy composable architectures. Technical Report 11459, SRI International, dec. 2004.

[129] E. Nightingale, J. Douceur, and V. Orgovan. Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In *EuroSys*, apr. 2011.

[130] S. Nomura, M. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam. Sampling + DMR: practical and low-overhead permanent fault detection. In *International Symposium on Computer Architecture*, jun. 2011.

[131] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, dec. 1996.

[132] J. Orton and K. Weick. Loosely coupled systems: a reconceptualization. *Academy of Management Review*, 15:203–223, apr. 1990.

[133] K. Palem and A. Lingamneni. What to do about the end of Moore's law, probably! In *Design Automation Conference*, jun. 2012.

[134] B. Panzer-Steindel. Data integrity, internal CERN/IT study. World Wide Web, 2007. `http://indico.cern.ch`.

[135] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *International Symposium on Microarchitecture*, dec. 2009.

[136] Y. Park, J. Park, H. Park, and S. Mahlke. Libra: tailoring SIMD execution using heterogeneous hardware and dynamic configurability. In *International Symposium on Microarchitecture*, dec. 2012.

[137] B. Parviz. Augmented reality in a contact lens. IEEE Spectrum, sept. 2009. `http://spectrum.ieee.org/biomedical/bionics/augmented-reality-in-a-contact-lens/`.

[138] G. Passas, M. Katevenis, and D. Pnevmatikatos. A 128 x 128 x 24gb/s crossbar interconnecting 128 tiles in a single hop and occupying 6% of their area. In *International Symposium on Networks-on-Chip*, may 2010.

[139] A. Patel, C. Madill, M. Saldana, C. Comis, R. Pomes, and P. Chow. A scalable FPGA-based multiprocessor. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, apr. 2006.

[140] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51 –57, sep. 1997.

[141] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. SymPLFIED: symbolic program-level fault injection and error detection framework. In *International Conference on Dependable Systems and Networks*, jun. 2008.

[142] A. Pellegrini and V. Bertacco. Application-aware diagnosis of runtime hardware faults. In *International Conference on Computer-Aided Design*, nov. 2010.

[143] A. Pellegrini and V. Bertacco. Cardio: adaptive CMPs for reliability through dynamic introspective operation. In *International High-Level Design, Validation and Test Workshop*, nov. 2011.

[144] A. Pellegrini and V. Bertacco. Cobra: a comprehensive bundle-based reliable architecture. In *International Conference on Embedded Computer Systems*, jul. 2013.

[145] A. Pellegrini, V. Bertacco, and T. Austin. Fault-based attack of RSA authentication. In *Design Automation and Test in Europe*, mar. 2010.

[146] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin. CrashTest: a fast high-fidelity FPGA-based resiliency analysis framework. In *International Conference on Computer Design*, oct. 2008.

[147] A. Pellegrini, J. Greathouse, and V. Bertacco. Viper: virtual pipelines for enhanced reliability. In *International Symposium on Computer Architecture*, jun. 2012.

[148] A. Pellegrini, R. Smolinski, L. Chen, X. Fu, S. Hari, J. Jiang, S. Adve, T. Austin, and V. Bertacco. CrashTest'ing SWAT: accurate, gate-level evaluation of symptom-based resiliency solutions. In *Design Automation and Test in Europe*, mar. 2012.

[149] M. Powell, A. Biswas, S. Gupta, and S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *International Symposium on Computer Architecture*, jun. 2009.

[150] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: cost-effective architectural support for rollback recovery in shared-mem multiprocessors. In *International Symposium on Computer Architecture*, may 2002.

[151] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi. Systematic software-based self-test for pipelined processors. In *Design Automation Conference*, jul. 2006.

[152] S. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro*, 20(4):47 –57, jul. 2000.

[153] S. Reinhardt and S. Mukherjee. Transient fault detection via simultaneous multithreading. In *International Symposium on Computer Architecture*, jun. 2000.

[154] T. Rintaluoma and O. Silvn. SIMD performance in software based mobile video coding. In *International Conference on Embedded Computer Systems*, jul. 2010.

[155] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, feb. 1978.

[156] E. Rotenberg, S. Bennett, and J. Smith. Trace Cache: a low latency approach to high bandwidth instruction fetching. In *International Symposium on Microarchitecture*, dec. 1996.

[157] G. Saggese, N. Wang, Z. Kalbarczyk, S. Patel, and R. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6), nov.-dec. 2005.

[158] R. Saleh, S. Wilton, S. Mirabbasi, A. Hu, M. Greenstreet, G. Lemieux, P. Pande, C. Grecu, and A. Ivanov. System-on-Chip: reuse and integration. *Proceedings of the IEEE*, 94(6):1050–1069, jun. 2006.

[159] R. Schaller. Moore's law: past, present and future. *Spectrum IEEE*, jun. 1997.

[160] M. Schilling. Towards a general modular systems theory and its application to inter-firm product modularity. *Academy of Management Review*, 25:312–334, apr. 1999.

[161] D. Schroder and J. Babcock. Negative bias temperature instability: road to cross in deep submicron silicon semiconductor manufacturing. *Journal of Applied Physics*, 94(1), jul. 2003.

[162] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *Proceedings of the International Joint Conference on Measurement and Modeling of Computer Systems*, jun. 2009.

[163] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: a high-speed, self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, pages 1318–1335, oct. 1991.

[164] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH*, aug. 2008.

[165] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, oct. 1996.

[166] M. Shah, R. Golla, G. Grohoski, P. Jordan, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja. SPARC T4: a dynamically threaded server-on-a-chip. *IEEE Micro*, 32(2):8–19, apr. 2012.

[167] S. Shamshiri and K.-T. Cheng. Modeling yield, cost, and quality of a spare-enhanced multicore chip. *IEEE Trans. Computers*, 60(9):1246–1259, sep. 2011.

[168] D. Shaw, M. Deneroff, R. Dror, J. Kuskin, R. Larson, J. Salmon, C. Young, B. Batson, K. Bowers, J. Chao, M. Eastwood, J. Gagliardo, J. Grossman, R. Ho, D. Ierardi, I. Kolossváry, J. Klepeis, T. Layman, C. McLeavey, M. Moraes, R. Mueller, E. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. Wang. Anton, a special-purpose machine for molecular dynamics simulation. In *International Symposium on Computer Architecture*, jun. 2007.

[169] S. Shende and A. Malony. The TAU parallel performance system. *Journal of High Performance Computing Applications*, pages 287–311, may 2006.

[170] A. Shimpi. The source of Intel's Cougar Point SATA bug. World Wide Web, 2011. `http://www.anandtech.com/show/4143/the-source-of-intels-cougar-point-sata-bug`.

[171] P. Shivakumar, S. Keckler, C. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *International Conference on Computer Design*, oct. 2003.

[172] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, oct. 2006.

[173] H. Siegel, L. Siegel, F. Kemmerer, P. Mueller, H. Smalley, and S. Smith. PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C-30(12):934 –947, dec. 1981.

[174] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture*, jun. 1995.

[175] D. Sorin. *Fault Tolerant Computer Architecture*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[176] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *International Symposium on Computer Architecture*, may 2002.

[177] J. Sosnowski. Software-based self-testing of microprocessors. *Journal of Systems Architecture: the EUROMICRO Journal*, 52(5):257–271, may 2006.

[178] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *International Symposium on Computer Architecture*, may 2002.

[179] R. Sproull, I. Sutherland, and C. Molnar. The Counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, jul. 1994.

[180] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The impact of technology scaling on lifetime reliability. In *International Conference on Dependable Systems and Networks*, jun. 2004.

[181] Standard Performance Evaluation Corp. SPECweb2005. World Wide Web, 2005. `http://www.spec.org/web2005/`.

[182] J. Stearley. Defining and measuring supercomputer reliability, availability, and serviceability (RAS). In *In Proceedings of the Linux Clusters Institute Conference*, jun. 2005.

[183] A. Stoica and R. Andrei. Adaptive and evolvable hardware - a multifaceted analysis. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, aug. 2007.

[184] A. Strong, E. Wu, R.-P. Vollertsen, J. Sune, G. LaRosa, and T. Sullivan. *Reliability Wearout Mechanisms in Advanced CMOS Technologies*. Wiley-IEEE Press, 2006.

[185] M. Strus, R. Keller, and N. Barbosa. Electrical reliability and breakdown mechanisms in single-walled carbon nanotubes. In *International Conference on Nanotechnology*, aug. 2011.

[186] Sun Microsystems Inc. OpenSPARC T1. World Wide Web, 2005. `http://opensparc-t1.sunsource.net/`.

[187] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *International Symposium on Microarchitecture*, dec. 2003.

[188] S. Swanson and M. Taylor. Greendroid: exploring the next evolution in smartphone application processors. *Communications Magazine, IEEE*, 49(4):112–119, apr. 2011.

[189] D. Tarjan, S. Thoziyoor, and N. Jouppi. Cacti 4.0. Technical report, HP Laboratories Palo Alto, 2006.

[190] M. Taylor. Is dark silicon useful?: harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference*, jun. 2012.

[191] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, mar.-apr. 2002.

[192] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: on-chip interconnect for ILP in partitioned architectures. In *International Symposium on High-Performance Computer Architecture*, feb. 2003.

[193] Textuality Services, Inc. Bonnie file system benchmark. World Wide Web, 1996. `http://www.textuality.com/bonnie/`.

[194] Tom's Hardware. CPU: articles & reviews. World Wide Web, 2012. `http://www.tomshardware.com/reviews/Components,1/CPU,1/`.

[195] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *International Symposium on High-Performance Computer Architecture*, feb. 1998.

[196] K. Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *International Symposium on FPGAs*, feb. 2004.

[197] A. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–396, dec. 1986.

[198] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, mar. 2010.

[199] J. Villasenor, B. Schoner, K.-N. Chia, C. Zapata, H. Kim, C. Jones, S. Lansing, and B. Mangione-Smith. Configurable computing solutions for automatic target recognition. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, apr. 1996.

[200] I. Wagner and V. Bertacco. Caspar: hardware patching for multicore processors. In *Design Automation and Test in Europe*, apr. 2009.

[201] I. Wagner, V. Bertacco, and T. Austin. Shielding against design flaws with field repairable control logic. In *Design Automation Conference*, jul. 2006.

[202] I. Wagner, V. Bertacco, and T. Austin. Using field-repairable control logic to correct design errors in microprocessors. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(2):380–393, feb. 2008.

[203] N. Wang and S. Patel. ReStore: symptom based soft error detection in microprocessors. In *International Conference on Dependable Systems and Networks*, jun. 2005.

[204] P. Wang, J. Collins, C. Weaver, B. Kuttanna, S. Salamian, G. Chinya, E. Schuchman, O. Schilling, T. Doil, S. Steibl, and H. Wang. Intel Atom processor core made FPGA-synthesizable. In *International Symposium on FPGAs*, feb. 2009.

[205] J. Weber. FAA: computer glitch caused airport delays. World Wide Web, 2009. `http://www.washingtontimes.com/news/2009/nov/19/faa-computer-glitch-causing-airport-delays/`.

[206] M. Woh, S. Satpathy, R. Dreslinski, D. Kershaw, D. Sylvester, D. Blaauw, and T. Mudge. Low power interconnects for SIMD computers. In *Design Automation and Test in Europe*, mar. 2011.

[207] E. Wu, E. Nowak, A. Vayshenker, W. Lai, and D. Harmon. CMOS scaling beyond the 100-nm node with silicon-dioxide-based gate dielectrics. In *IBM Journal of Research and Development*, mar.-may, 2002.

[208] E. Wu, J. Sune, W. Lai, E. Nowak, J. McKenna, A. Vayshenker, and D. Harmon. Interplay of voltage and temperature acceleration of oxide breakdown for ultra-thin gate dioxides. In *Solid-state Electronics Journal*, nov. 2002.

[209] L. Wu, C. Weaver, and T. Austin. CryptoManiac: a fast flexible architecture for secure communication. In *International Symposium on Computer Architecture*, jun. 2001.

[210] G. Xenoulis, D. Gizopoulos, N. Kranitis, and A. Paschalis. Low-cost, on-line software-based self-testing of embedded processor cores. In *International On-Line Testing Symposium*, jul. 2003.

[211] Xilinx Corporation. Microblaze processor reference guide. World Wide Web, 2009. `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`.

[212] X. Yang, Z. Wang, J. Xue, and Y. Zhou. The reliability wall for exascale supercomputing. *IEEE Transactions on Computers*, 61(6), jun. 2012.

[213] K. Yeager. The Mips R10000 superscalar microprocessor. *Micro, IEEE*, 16(2):28–41, apr. 1996.

[214] S. Zafar, B. Lee, J. Stathis, A. Callegari, and T. Ning. A model for negative bias temperature instability (NBTI) in oxide and high-K pFETs. In *Symposia on VLSI Technology and Circuits*, jun. 2004.

[215] J. Zhang, A. Lin, N. Patil, H. Wei, L. Wei, H.-S. Wong, and S. Mitra. Carbon nanotube robust digital VLSI. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(4):453 –471, apr. 2012.

[216] W. Zhang, M. Kandemir, N. Vijaykrishnan, M. Irwin, and V. De. Compiler support for reducing leakage energy consumption. In *Design Automation and Test in Europe*, mar. 2003.

[217] J. Ziegler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, pages 19–39, jan. 1996.