# Adaptive Architectures for Robust and Efficient Computing

by

Shantanu Gupta

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
2011

Doctoral Committee:

Associate Professor Scott A. Mahlke, Chair
Professor Todd M. Austin
Associate Professor Valeria M. Bertacco
Assistant Professor Zhengya Zhang
Michael D. Powell, Intel Corporation

To my parents.

# ACKNOWLEDGEMENTS

First, I would like to express my sincerest gratitude to my advisor, Professor Scott Mahlke, for his mentorship and unwavering support throughout my doctoral research. Scott's enthusiasm for research and technical acumen have been an inspiration to everyone in our group, and I consider myself truly fortunate to have worked under his able guidance.

I would also like to thank my dissertation committee, Professor Todd Austin, Professor Valeria Bertacco, Dr. Michael Powell, and Professor Zhengya Zhang, for their time and effort in reviewing my thesis, holding candid discussions about my research, and encouraging me throughout this process. I have known Michael since my internship at Intel in the summer of 2008, and I must thank him for adding an industrial perspective to my research, and giving me every opportunity to learn details of Intel's processors.

I am also indebted to my close research colleagues Jason Blome, Shuguang Feng and Amin Ansari. It has been a real pleasure working with all of them, and I can not imagine having this thesis in its current form without their support. Jason gave me the original idea of the StageNet architecture, which forms the basis of my thesis. Shuguang and Amin have been with me throughout this process, sitting patiently through our long meetings with Scott, giving excellent research insights, and always being there for any help with infrastructure development and writing papers.

I consider myself truly lucky to have worked with such intelligent and vibrant set of people in our research lab, CCCP. All of you made coming to office great fun. Thank you Nathan Clark, Rajiv Ravindran, Michael Chu, Hongtao Zhong, Kevin Fan, Manjunath Kudlur, and Hyunchul Park for setting up a lively culture within our group. I want to especially thank my colleagues who surrounded me during the later half of my PhD: Ganesh Dasika, Amir Hormati, Mark Woh (pseudo-CCCP), Mojtaba Mehrara, Yongjun Park, Hyoun Kyu Cho, Jeff Hao, Po-Chun Hsu, Mehrzad Samadi, Gaurav Chadha and Ankit Sethia.

My stay in Michigan has been enriched by a great set of friends and roommates that surrounded me. I would like to thank Sudherssen Kalaiselvan, Ravikishore Gandikota, Kaviraj Chopra, Sanjay Pant, Visvesh Sathe, Gauri Sathe, Manavendra Mahato, Vivek Joshi, Prashant Singh, Ashwini Kumar, Anurag Tripathi, Naveen Gupta, Trushal Chokshi and Abhishek Kumar, who have all been my roommates at one point or another. I can not thank you all enough for being great company, making excellent food, and taking care of me in times of need. Anurag, thank you for searching and playing all those excellent movies and television shows, that formed a bulk of our evening entertainment. My gratitude also goes to countless other friends at Michigan, whom I have failed to mention in paragraphs above.

Finally and most importantly, my family deserves a major gratitude. My brothers were a constant source of encouragement, and made all my vacations in India truly memorable. And above all, I really appreciate the unconditional love and support of my parents. They never let me doubt my abilities, and have been a guiding light throughout this process.

# TABLE OF CONTENTS

# LIST OF FIGURES

**Figure**

# LIST OF TABLES

**Table**

# ABSTRACT

Adaptive Architectures for Robust and Efficient Computing

by

Shantanu Gupta

Chair: Scott A. Mahlke

Semiconductor technology scaling has long been a source of dramatic gains in our computing capabilities. However, as we test the physical limits of silicon feature size, serious reliability and computational efficiency challenges confront us. The supply voltage levels have practically stagnated, resulting in increasing power densities and operating temperatures. Given that most semiconductor wearout mechanisms are highly dependent on these parameters, significantly higher failure rates are projected for future technology generations. Further, the rise in power density is also limiting the number of resources that can be kept active on chip simultaneously, motivating the need for energy-efficient computing.

In this landscape of technological challenges, incremental architectural improvements to existing designs are likely insufficient, motivating a need to rethink the architectural fabric from the ground up. Towards this end, this thesis presents adaptive architecture and compiler solutions that can effectively tackle reliability, performance and energy-efficiency

demands expected in future microprocessors.

For the reliability challenge, we present StageNet, a highly reconfigurable multicore architecture that is designed as a network of pipeline stages, rather than isolated cores. The interconnection flexibility in StageNet allows it to adaptively route around defective pieces of a processor, and deliver graceful performance degradation in the face of failures. We further complement the fault isolation ability of StageNet with an adaptive testing framework that significantly reduces the overhead of in-field fault detection.

As a second contribution, we build upon the interconnection flexibility of StageNet to develop a unified performance-reliability solution. This subsequent design, named CoreGenesis, relies on a set of microarchitectural innovations and compiler hints to merge processor cores for a higher single-thread performance. This enables customization of processing ability (narrow or wide-issue pipelines) to the dynamic workload requirements.

In the final work of this thesis, we investigate the sources of computational inefficiency in general purpose processors, and propose a configurable compute engine, named BERET, for substantial energy savings. The insight here is to cut down on the redundant instruction fetch, decode and register file access energy by optimizing the execution of recurring instruction sequences.

# CHAPTER I

# Introduction

Over the last few decades, the semiconductor industry has sustained a staggering growth in silicon integration levels. This aggressive scaling, made possible by numerous technological breakthroughs, has been the driving force behind performance and efficiency milestones in computational systems. However, as the silicon technology approaches its fundamental limits, a variety of challenges confront it [15]. The issues range from designs nearing the power and thermal limitations to extreme process variation and wearout failures in the manufactured parts. As device density grows, each transistor gets smaller and more fragile leading to an overall higher susceptibility of chips to hard faults. Hard faults result in permanent silicon defects, and impact the yield, lifetime performance, and reliability of semiconductor parts [17]. Increasing transistor density also poses difficult thermal problems as heat cannot be efficiently dissipated, leading to a plateau in clock frequencies. All of these issues are detrimental to the semiconductor industry's economic model. Loss of compelling performance gains reduces the incentive to regularly upgrade machines, loss in yield directly translates to loss in sales and in-field defects could necessitate conservative designs to avoid substantial performance degradation.

In this broad spectrum of technological concerns, power density was identified as an immediate roadblock for technology scaling. In an effort to address this concern, the industry has shifted its design philosophy from monolithic superscalar processors to multi-core processors composed of relatively smaller cores. Some of the latest desktop and server chips range from 6-12 individual cores [51, 4]. Further, many manufacturers have begun production of many-core chips with simple in-order cores to target market segments that demand throughput computing, e.g. SUN Niagara [75], Tilera TILE64 [109].

## 1.1 Technology Challenges

While the paradigm shift to multicore architectures has abated power and thermal concerns to a certain extent, going forward, three major challenges still need to be addressed by semiconductor manufacturers. The first challenge is the waning reliability of transistors and their increasing vulnerability to wearout, manufacturing defects and process variation. The second, and a more direct ramification of adopting multicores, is the saturation in single-thread performance. This is especially of concern to applications that are not amenable to parallelization. The third and final challenge is power density, which is a result of stagnating supply voltage levels. This is introducing tight power constraints on the manufactured parts, and limiting the number of resources that can be simultaneously kept active on a chip [114].

### 1.1.1 The Reliability Challenge

The sources of computer system failures are widespread, ranging from transient faults, due to energetic particle strikes [124] and electrical noise, to permanent faults, caused by device wearout, manufacturing defects and extreme process variation. In recent years, industry designers and researchers have invested significant effort in building architectures resistant to transient faults [96, 116]. In transient faults (also referred to as *soft errors*) the damage to a chip is never permanent, and a replay of instructions is typically sufficient for recovery. In contrast, dealing with permanent faults is significantly more involved, and relatively little research has been conducted to efficiently tolerate the same. There are numerous sources of permanent faults, ranging from manufacturing defects, process variation, to in-field wearout phenomenas such as electromigration [23], time dependent dielectric breakdown (TDDB) [118], negative bias temperature instability (NBTI) [123], etc.

The challenge of tolerating permanent faults can be broadly divided into three requisite tasks: fault detection, fault diagnosis, and system recovery/reconfiguration. Fault detection mechanisms [16, 71] are used to identify the presence of a fault, while fault diagnosis techniques [19] are used to determine the source of the fault, i.e. the broken component(s). System reconfiguration needs to leverage some form of a spatial or temporal redundancy to keep the faulty component isolated from the design. The granularity at which spares/redundancy is maintained determines the number of failures a system can tolerate.

As an example, many computer vendors provide the ability to repair faulty memory

and cache cells through the inclusion of spare memory elements. This technique of isolating broken structures or supplementing them with spares has also been extended to logic resources [18, 95, 103], and all the way to disabling entire cores [2, 105]. While these resource isolation and sparing techniques are reasonable solutions, in a high failure rate scenario, such systems will exhibit a rapid throughput degradation and quick become unresponsive.

A major thrust of this thesis is to understand the issues associated with system reconfiguration and to design a fault tolerant architecture that is capable of tolerating a large number of manufacture-time and in-field failures.

### 1.1.2 The Performance Challenge

Recent years have witnessed a migration towards multicore architectures by hardware vendors. However, software developers have adopted this trend more slowly, creating a disparity between application requirements and the underlying hardware. Due to this inertia in the software development cycles, applications today range from heavily sequential legacy workloads to throughput oriented parallel counterparts. In such a diverse landscape of software products, a fixed multicore design cannot provide optimal performance, creating a need for architecture level flexibility.

Generally speaking, multiple cores are effective when threads are plentiful and throughput computing is required, but they provide little or no gains for sequential applications. Furthermore, the performance of sequential applications may suffer as cores get simpler and smaller caches per core are provided. Despite a mass transition to application parallelization, there are two significant reasons that make single-thread performance important.

First, most applications today are single-threaded and have been written with a heavy bias towards a monolithic processing model. Converting them all into efficient parallel programs will be a phenomenal challenge. Second, even if a major transition towards parallel programming occurs in the future, Amdahl's law dictates that the sequential component of an application will present itself as a performance bottleneck. Thus, multicore solutions, while being a natural fit for throughput computing, must also have the flexibility to provide high single-thread performance.

One way to provide this flexibility is to design heterogeneous multi-core architectures [63], with a set of small (for throughput) and big cores (for sequential programs). However, this approach is very rigid, and makes strong assumptions about the set of applications that will be active at a given point in time. A part of this thesis is devoted to the concept of *dynamic multicores*, that can reconfigure themselves as per workload requirements. When plenty of threads are available to work on, system resources can be broken up into individual cores, and in an opposing scenario, resources can be coalesced to form larger but fewer cores.

### 1.1.3 The Energy-Efficiency Challenge

Over the years, transistor densities and performance has continued to increase as per Moore's Law, however, the threshold voltage has not kept up with this trend. As a result, the per-transistor switching power has not witnessed the benefits of scaling, causing a steady rise in power density. Overall, this limits the number of resources that can be kept active on a die simultaneously [114]. An instance of this trend can be already seen in Intel's newest Nehalem generation of processors that boost the performance of one core, at the cost of

slowing down/shutting off the rest of them.

Given these circumstances, an improvement in computational efficiency is essential to keep the power density, and hence the total chip-wide power envelope, under check. Alternatively, a better computational efficiency also translates into reduced energy per instruction, increasing the net amount of work done for a fixed energy budget. This is very valuable in the context of server farms, where energy consumed cuts both ways, i.e. for computation as well as for cooling. Besides desktop and server space, the efficiency requirement is also visible within the growing domain of portable devices [68], that operate under a strict energy budget due to their dependence on batteries.

While there has been a lot of work to improve the efficiency of embedded systems, such as specialized hardware units [77, 80, 93], accelerators [35, 122], and application specific instruction extensions [107], the general purpose processor domain has largely been ignored. This is partly influenced by the fact that general purpose application space is very diverse and constantly evolving making it hard to specialize architectures. In this thesis, we investigate this challenge and propose a configurable substrate to improve the energy-efficiency of general purpose processors.

## 1.2 Adaptive Architectures

Given this landscape of increasing failure rates, diminishing single-thread performance, and tighter power constraints, computer architects have an indispensable role to play. However, incremental improvements to existing architectures are likely insufficient for achieving these objective as legacy hardware imposes many restrictions. This motivates a re-

thinking of the architectural fabric from the ground up, with *dynamic adaptivity* and *configurability* as primary requirements. The overarching objective of this thesis is to design and evaluate such adaptive architectures that can deliver robust, configurable, and efficient performance.

In the context of reliability, dynamic adaptivity refers to the system's ability to actively detect failures, isolate broken components and reconfigure itself. For being effective, the fault isolation has to be at a granularity finer than broken cores. Whereas, for single-thread performance, dynamic adaptivity refers to the system's ability to adequately allocate compute resources to the active threads, i.e., more resources to a high IPC thread and vice versa. Finally, for energy efficiency, dynamic adaptivity refers to hardware substrate's ability to specialize itself for data flow patterns in an application.

This thesis presents four efforts towards designing adaptive architectures: 1) the StageNet architecture for fine-grained fault isolation, 2) the adaptive test framework for efficient fault detection/diagnosis, 3) the CoreGenesis architecture for enabling *dynamic multicore* capability, and 4) the Green BERET architecture for energy-efficient execution of recurring instruction sequences.

### 1.2.1 Adaptivity for Defect Isolation

Permanent fault tolerance requires system reconfiguration in order to isolate broken components. The popular solution for this has been the use of redundancy at a coarse granularity, such as dual/triple modular redundancy. More recently in the form of core disabling within a multicore chip. In this work, we challenge the practice of coarse-granularity redundancy by identifying its inability to scale to high failure rate scenarios and investigating

the advantages of finer-grained configurations. As a solution, we present and evaluates a highly reconfigurable CMP architecture, named StageNet (SN), that is designed with reliability as its first class design criteria.

SN is a multicore architecture designed as a network of pipeline stages, rather than isolated cores in a CMP. The network is formed by replacing the direct connections at each pipeline stage boundary by a crossbar switch interconnection. Within the SN architecture, pipeline stages can be selected from the pool of available stages to act as logical processing cores. A logical core in the StageNet architecture is referred to as a *StageNetSlice* (SNS). A SNS can easily isolate failures by adaptively routing around faulty stages. The interconnection flexibility in the system allows SNSs to salvage healthy stages from adjacent cores and even makes it possible for different SNSs to time-multiplex a scarce pipeline resource. Because of this added flexibility, a SN system possesses inherent redundancy (through borrowing and sharing pipeline stages) and is therefore, all else being equal, capable of maintaining higher throughput over the duration of a system's life compared to a conventional multicore design.

As an extension, we also propose StageWeb (SW) architecture that eliminates three limitations of the SN architecture: 1) scalability to 10/100s of cores, 2) interconnection reliability and 3) ability to address process variation.

### 1.2.2 Adaptivity for Online Testing

Detection and diagnosis of failures is a crucial component of a fault tolerant system. In a scenario where in-field silicon defects (from wearout) become commonplace, processors would need to be equipped with online fault detection mechanisms. Periodic online testing

is a popular technique to detect such failures; however, it tends to impose a heavy testing penalty. In this thesis, we propose an adaptive online testing framework to significantly reduce the testing overhead. The proposed approach is unique in its ability to assess the hardware health and apply suitably detailed tests. Near the start of a chip lifetime, most components are relatively healthy, and do not need a full battery of tests. As time goes on, and health sensors indicate deterioration in various components (cores in this case), test patterns can be made more rigorous. Using this approach, a significant chunk of the testing time can be saved for the healthy components.

We further extend the framework to work with the SN fabric, which provides the flexibility to group together pipeline stages with similar health conditions, thereby reducing the overall testing burden.

### 1.2.3 Adaptivity for Performance

The third design presented in this thesis proposal provides architectural adaptivity for performance, by realizing the concept of dynamic multicores. The solution presented, named CoreGenesis, is an adaptive multiprocessor fabric that can assemble variable-width processors from a network of (potentially broken) pipeline stage-level resources. CoreGenesis relies on interconnection flexibility, coupled with a set of microarchitectural innovations for decentralized instruction flow management, 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 (similar to StageNet). Together, the resulting fabric consists of a pool of pipeline stage-level resources that can be fluidly allocated for accelerating single-thread performance, throughput computing, or tolerating failures.

The concept of dynamic multicore has been studied in the past, with solutions like Core Fusion [52] and Federation [106] that can fuse neighboring cores with a help of shared microarchitectural structures. However, none of those solutions can provide fine-grained fault isolation. The novelty of CoreGenesis arises from its approach to co-design a performance/reliability solution, while overlapping a majority of hardware overheads.

### 1.2.4 Adaptivity for Energy Efficiency

The fourth and final design presented in this thesis is a configurable compute engine for energy-efficient execution. Efficiency solutions in the past have typically targeted embedded systems by developing application specific hardware and accelerators. Unfortunately, these approaches do not extend to general purpose applications due to their irregular and diverse code base. In this work, we propose BERET, a novel energy-efficient co-processor that targets general purpose programs and significantly reduces the computational costs for frequently repeated sequences of instructions.

The BERET architectures relies on two insights to meet its efficiency goals. First, it identifies recurring instruction traces in a given program, and buffers them internally to cut down on redundant instruction fetch and decode energy. Targeting these traces also helps BERET add a level of *temporal regularity* to the otherwise irregular behavior of desktop and server programs. Second, it uses a bundled execution model to reduce register reads and writes for temporary variables. These execution bundles are essentially sub-graphs from the trace data flow graph. We consider this bundled execution model a trade-off design that lets us achieve efficiency gains close to an application specific data flow hardware while maintaining application universality of regular Von Neumann execution model.

## 1.3 Contributions

The contribution of this thesis can be summarized as follows:

- It demonstrates that fault isolation can be conducted at a level finer than processing cores, without maintaining any cold spares.
- An adaptive architecture is proposed, named StageNet, that enables pipeline-stage level fault isolation. The impact on single-thread performance is very nominal at 10%.
- A sensor guided adaptive testing solution is presented for improving the efficiency of periodic online testing. The proposed solution is the first of its kind to assess the system health before assigning test vectors for hard fault detection.
- A unified performance / reliability solution, named CoreGenesis. This builds upon the StageNet architecture, and adds capabilities for dynamically grouping together pipelines for creating wider-issue machines.
- A configurable compute engine, named BERET, for energy-efficient execution. The design leverages recurring instructions sequences and a bulk execution model to significantly reduce instruction fetch, decode and register file access energy.

## 1.4 Organization

The rest of this document is organized as follows:

Chapter II presents the StageNet architecture, a solution for fine grained fault isolation.

Chapter III takes the StageNet design forward, making it scalable, tolerant to interconnection faults, and capable of mitigating process variation.

Chapter IV introduces our solution for making periodic online testing more efficient.

Chapter V presents the CoreGenesis architecture, which adds a performance dimension to the StageNet architecture.

Chapter VI presents the BERET architecture for bridging the efficiency gap of general purpose processors and application specific hardware.

Chapter VII concludes this thesis.

# CHAPTER II

# The StageNet Fabric for Constructing Resilient Chip Multiprocessors

## 2.1 Introduction

Technological trends into the nanometer regime have lead to increasing current and power densities and rising on-chip temperatures, resulting in both increasing transient, as well as permanent, failures rates. Leading technology experts have warned designers that device reliability will begin to deteriorate from the 65nm node onward [15]. Current projections indicate that future microprocessors will be composed of billions of transistors, many of which will be unusable at manufacture time, and many more which will degrade in performance (or even fail) over the expected lifetime of the processor [17]. In an effort to assuage these concerns, industry has initiated a shift towards multicore and GPU inspired designs that employ simpler cores to limit the power and thermal envelope of the chips [59, 62]. However, this paradigm shift also leads towards core designs that have little inherent redundancy and are therefore incapable of performing the self-repair possible in big superscalar cores [95, 103]. Thus, in the near future, architects must directly address

reliability in computer systems through innovative fault-tolerance techniques.

There are two major sources of failures in computer hardware. First are transient faults that can occur as a result of energetic particle strikes [124] and electrical noise. By virtue of being transient, their effect is temporary, and a system can continue normal operation after recovering from them. The second category is permanent faults resulting from wafer defects, manufacturing-time variations, and wearout phenomenon such as electromigration [23] and time dependent dielectric breakdown [118]. In recent years, industry designers and researchers have invested significant effort in building architectures resistant to transient faults [96, 116]. In contrast, much less attention has been paid to the problem of permanent faults, specifically transistor wearout due to the degradation of semiconductor materials over time. Traditional techniques for dealing with transistor wearout have involved extra provisioning in logic circuits, known as guard-banding, to account for the expected performance degradation of transistors over time. However, the increasing degradation rate projected for future technology generations implies that traditional margining techniques will be insufficient.

Permanent fault tolerance can be broadly divided into three steps: fault detection, fault diagnosis, and system recovery/reconfiguration. Fault detection mechanisms [16, 71] are used to identify the presence of a fault, while fault diagnosis techniques [19] are used to determine the source of the fault, i.e. the broken component(s). System recovery needs to leverage some form of a spatial or temporal redundancy to isolate the faulty component(s) and perform the repair. For instance, many computer vendors provide the ability to repair faulty memory and cache cells through the inclusion of spare memory elements. Recently, researchers have begun to extend these techniques to support sparing for additional on-chip

resources [103], such as branch predictors [18] and registers [95]. The granularity at which spares/redundancy is maintained determines the number of failures a system can tolerate. The focus of this work is to understand the issues associated with system recovery and to design a fault tolerant architecture that is capable of tolerating a large number of failures.

Traditionally, system recovery in high-end servers and mission critical systems has been addressed by using mechanisms such as dual and triple-modular redundancy (DMR and TMR) [14]. However, such approaches are too costly and therefore not applicable to desktop and embedded systems. With the recent popularity of multicore systems, these traditional core-level approaches have been able to leverage the inherent redundancy present in large chip multiprocessors (CMPs) [2, 105]. However, both the historical designs and their modern incarnations, because of their emphasis on core-level redundancy, incur high hardware overhead and can only tolerate a small number of defects. With the increasing defect rate in semiconductor technology, it will not be uncommon to see a rapid degradation in throughput for these systems as single device failures cause entire cores to be decommissioned, often times with the majority of the core still intact and functional.

In contrast, this chapter argues the case for reconfiguration and redundancy at a finer granularity. To this end, this work presents the *StageNet* (SN) fabric, a highly reconfigurable and adaptable computing substrate. SN is a multicore architecture designed as a network of pipeline stages, rather than isolated cores in a CMP. The network is formed by replacing the direct connections at each pipeline stage boundary by a crossbar switch interconnection. Within the SN architecture, pipeline stages can be selected from the pool of available stages to act as logical processing cores. A logical core in the StageNet architecture is referred to as a *StageNetSlice* (SNS). A SNS can easily isolate failures by adaptively

routing around faulty stages. The interconnection flexibility in the system allows SNSs to salvage healthy stages from adjacent cores and even makes it possible for different SNSs to time-multiplex a scarce pipeline resource. Because of this added flexibility, a SN system possesses inherent redundancy (through borrowing and sharing pipeline stages) and is therefore, all else being equal, capable of maintaining higher throughput over the duration of a system's life compared to a conventional multicore design. Over time as more and more devices fail, such a system can gracefully degrade its performance capabilities, maximizing its useful lifetime.

The reconfiguration flexibility of the SN architecture has a cost associated with it. The introduction of network switches into the heart of a processor pipeline will inevitably lead to poor performance due to high communication latencies and low communication bandwidth between stages. The key to creating an efficient SN design is re-thinking the organization of a basic processor pipeline to more effectively isolate the operation of individual stages. More specifically, inter-stage communication paths must either be removed, namely by breaking loops in the design, or the volume of data transmitted must be reduced. This chapter starts off with the design of an efficient SNS (a logical StageNet core) that attacks these problems and reduces the performance overhead from network switches to an acceptable level. Further, it presents the SN multicore that stitches together multiple such SNSs to form a highly reconfigurable architecture capable of tolerating a large number of failures. In this work, we take a simple in-order core design as the basis of the SN architecture. This is motivated by the fact that thermal and power considerations are pushing designs towards simpler cores. In fact, simple cores are being adopted by designs targeting massively multicore chips and are suitable for low-latency and high throughput applications [62, 59]. At

the same time, we believe that the proposed design methodology of fine-grained reconfiguration can also be effectively applied to deeper and more aggressive pipeline designs.

The primary contributions of this chapter include: 1) A design space exploration of reconfiguration granularities for resilient systems; 2) design, evaluation and performance optimization of StageNetSlice, a networked pipeline microarchitecture; and 3) design and evaluation of StageNet, a resilient multicore architecture, composed using multiple SNSs.

## 2.2 Reconfiguration Granularity

For tolerating permanent faults, architectures must have the ability to reconfigure, where reconfiguration can refer to a variety of activities ranging from decommissioning non-functioning, non-critical processor structures to swapping in cold spare devices. In a reconfigurable architecture, recovery entails isolating defective module(s) and incorporating spare structures as needed. Support for reconfiguration can be achieved at various granularities, from ultra-fine grain systems that have the ability to replace individual logic gates to coarser designs that focus on isolating entire processor cores. This choice presents a trade-off between complexity of implementation and potential lifetime enhancement. This section shows experiments studying this trade-off and draws upon these results to motivate the design of the SN architecture.

### 2.2.1 Experimental Setup

In order to effectively model the reliability of different designs, a Verilog model of the OpenRISC 1200 (OR1200) core [76] was used in lifetime reliability experiments. The

OR1200 is an open-source core with a conventional 5-stage pipeline design, representative of commercially available embedded processors. The core was synthesized, placed and routed using industry standard CAD tools with a library characterized for a 130nm process. The final floorplan along with several attributes of the design is shown in Figure 2.1.

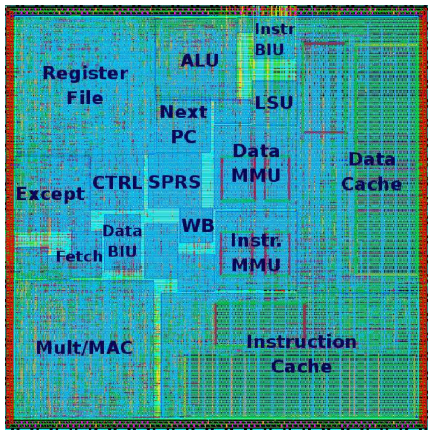
To study the impact of reconfiguration granularity on chip lifetimes, the mean-time-to-failure (MTTF) was calculated for each individual module in the OR1200. MTTF was determined by estimating the effects of a common wearout mechanism, time-dependent dielectric breakdown (TDDB) on a OR1200 core running a representative workload. Employing an empirical model similar to that found in [101], Equation 2.1 presents the formula used to calculate per-module MTTFs. The temperature numbers for the modules were generated using HotSpot [47]. Given the MTTFs for individual modules, stage-level MTTFs in our experiment were defined as the minimum MTTF of any module belonging to the stage. Similarly, core level MTTFs were defined as the minimum MTTF across all the modules.

$$MTTF_{TDDB} \propto (\frac{1}{V})^{(a-bT)} e^{\frac{(X+\frac{Y}{T}+ZT)}{kT}} \tag{2.1}$$

where V = operating voltage, T = temperature, k = Boltzmann's constant, and a,b,X,Y,and Z are all fitting parameters based on [101].

### 2.2.2 Granularity Trade-offs

The granularity of reconfiguration is used to describe the unit of isolation/redundancy for modules within a chip. Various options for reconfiguration, in order of increasing granularity, are discussed below.




(a) Overlay of floorplan

| **OR1200 Core** | |
|---|---|
| Area | 1.0 $mm^2$ |
| Power | 123.9 mW |
| Clock Frequency | 400 MHz |
| Data Cache Size | 8 KB |
| Instruction Cache Size | 8 KB |
| Technology Node | 90nm |

(b) Implementation details

**Figure 2.1: OpenRisc 1200 embedded microprocessor.**

1. *Gate level:* At this level of reconfiguration, a system can replace individual logic gates in the design as they fail. Unfortunately, such designs are typically impractical because they require both precise fault diagnosis and tremendous overhead due to redundant components and wire routing area.

2. *Module level:* In this scenario, a processor core can replace broken microarchitectural structures such as an ALU or branch predictor. Such designs have been active topics of research [29, 95]. The biggest downside of this reconfiguration level is that maintaining redundancy for full coverage is almost impractical. Additionally, for the case of simple cores, even fewer opportunities exist for isolation since almost all modules are unique in the design.

3. *Stage level:* Here, the entire pipeline stages are treated as single monolithic units that can be replaced. Reconfiguration at this level is challenging because: 1) pipeline stages are tightly coupled with each other (reconfiguration can cause performance loss), and 2) cold sparing pipeline stages is expensive (area overhead).
4. *Core level:* This is the coarsest level of reconfiguration where entire processor cores are isolated from the system in the event of a failure. Core level reconfiguration has also been an active area of research [105, 2], and from the perspective of a system designer, it is probably the easiest technique to implement. However, it has the poorest returns in terms of lifetime extension, and therefore might not be able to keep up with increasing defect rates.

While multiple levels of reconfiguration granularity could be utilized, Figure 2.2 demonstrates the effectiveness of each applied in isolation (gate-level reconfiguration was not included in this study). The figure shows the potential for lifetime enhancement (measured as MTTF) as a function of how much area a designer is willing to allocate to cold spares. The MTTF of a n-way redundant structure is taken to be $n$ times its base MTTF. And, the MTTF of the overall system is taken to be the MTTF of the fastest failing module in the design. This is similar to the serial model of failure used in [101]. The figure overlays three separate plots, one for each level of reconfiguration. The redundant spares were allowed to add as much as 300% area overhead.

The data shown in Figure 2.2 demonstrates that going towards finer-grain reconfiguration is categorically beneficial as far as gains in MTTF are concerned. But, it overlooks the design complexity aspect of the problem. Finer-grain reconfiguration tends to exacerbate the hardware challenges for supporting redundancy, e.g. muxing logic, wiring overhead,

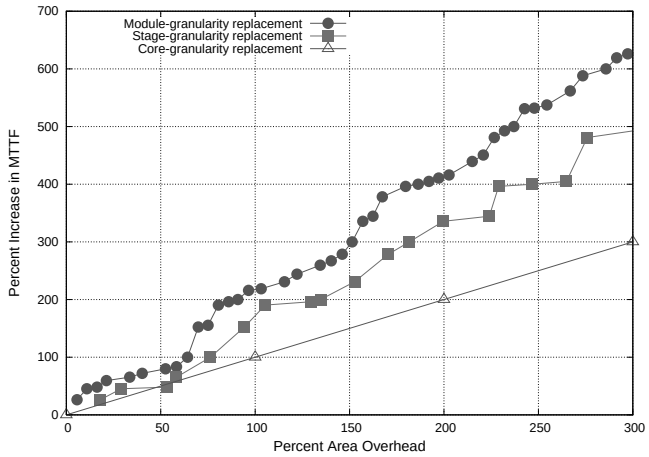



**Figure 2.2: Gain in MTTF from the addition of cold spares at the granularity of micro-architectural modules, pipeline stages, and processor core. The gains shown are cumulative, and spare modules are added (denoted with markers) in the order they are expected to fail.**

circuit timing management, etc. At the same time, very coarse grained reconfiguration is also not an ideal candidate since MTTF scales poorly with the area overhead. Therefore, a compromise solution is desirable, one that has manageable reconfiguration hardware and a better life expectancy.

### 2.2.3 Harnessing Stage-level Reconfiguration

Stage level reconfiguration is positioned as a good candidate for system recovery as it scales well with the increase in area available for redundancy (Figure 2.2). Logically, stages are a convenient boundary because pipeline architectures divide work at the level of stages (e.g., fetch, decode, etc.). Similarly, in terms of circuit implementation, stages are an intuitive boundary because data signals typically get latched at the end of every pipeline stage. Both these factors are helpful when reconfiguration is desired with a minimum impact on the performance. However, there are two major obstacles that must be overcome before stage level reconfiguration is practical:

1. Pipeline stages are tightly coupled with each other and are therefore difficult to isolate/replace.
2. Maintaining spares at the pipeline stage granularity is very area intensive.

One of the ways to allow stage level reconfiguration is to decouple the pipeline stages from each other. In other words, remove all direct point-to-point communication between the stages and replace them by a switch based interconnection network. A conceptual picture of a chip multiprocessor using this philosophy is presented in Figure 2.3. We call this design *StageNet* (**SN**). Processor cores within SN are designed as part of a high speed network-on-a-chip, where each stage in the processor pipeline corresponds to a node in the network. A horizontal slice of this architecture is equivalent to a logical processor core, and we call it a *StageNetSlice* (**SNS**). The use of switches allows complete flexibility for a pipeline stage at depth *N* to communicate with any stage at depth *N+1*, even those from a different SNS. The SN architecture overcomes both of the major obstacles for stage level reconfiguration. Pipeline stages are decoupled from each other, and hence faulty ones can be easily isolated. Furthermore, there is no need to exclusively devote chip area for cold sparing. The SN architecture exploits the inherent redundancy present in a multicore by borrowing/sharing stages from adjacent cores. As nodes (stages) wearout and eventually fail, SN will exhibit a graceful degradation in performance, and a gradual decline in throughput.

Along with its benefits, SN architecture has certain area and performance overheads associated with itself. Area overhead primarily arises from the switch interconnection network between the stages. And depending upon the switch bandwidth, a variable number of cycles will be required to transmit operations between stages, leading to performance

penalties. The next section investigates the performance overheads when using a SNS and also presents our microarchitectural solutions to regain these losses. The remainder of the chapter focuses on the design and evaluation of the SN architecture, and demonstrates its ability to maintain high lifetime throughput in the face of failures.

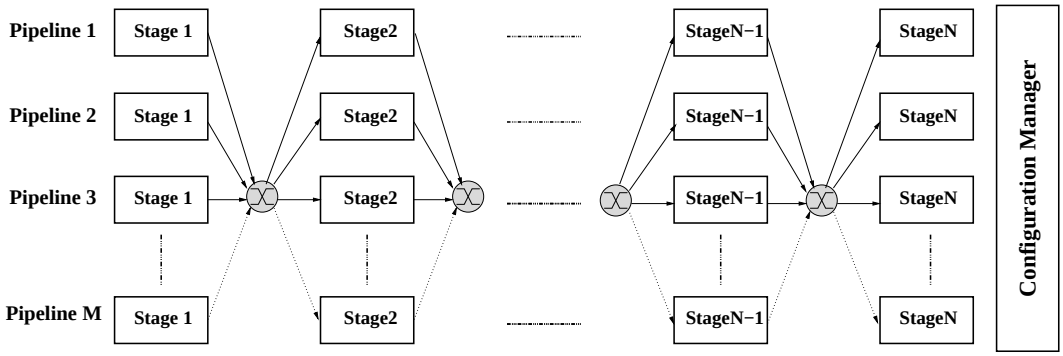



**Figure 2.3: A StageNet assembly: group of slices connected together. Each StageNetSlice (SNS) is equivalent to a logical processing core. This figure shows M, N-stage slices. Broken stages can be easily isolated by routing around them. Crossbar switch spares can also be maintained at the pipeline stage boundaries in order to tolerate rare, albeit possible, switch failures.**

## 2.3 The StageNetSlice Architecture

### 2.3.1 Overview

StageNetSlice (SNS) is a basic building block for the SN architecture. It consists of a decoupled pipeline microarchitecture that allows convenient reconfiguration at the granularity of stages. As a basis for the SNS design, a simple embedded processor core is used, consisting of five stages namely, fetch, decode, issue, execute/memory, and write-back [8, 76]. Although the execute/memory block is sometimes separated into multiple stages, it is treated as a single stage in this work.

Starting with a basic in-order pipeline, we will go through the steps of its transformation

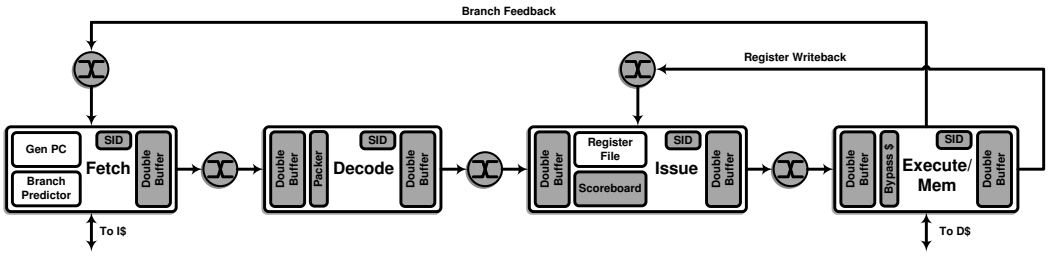



**Figure 2.4: A StageNetSlice (SNS) pipeline. Stages are interconnected using a full crossbar switch. The shaded portions highlight modules that are not present in a regular in-order pipeline.**

into SNS. As the first step, pipeline latches are replaced with a combination of a crossbar switch and buffers. A graphical illustration of the resulting pipeline design is shown in Figure 2.4. The shaded boxes inside the pipeline stages are microarchitectural additions that will be discussed in detail later in this section. To minimize the performance loss from inter-stage communications, we propose the use of full crossbar switches since a) these allow non-blocking access to all of their inputs and b) for a small number of inputs and outputs they are not prohibitively expensive. The full crossbar switches have a fixed channel width and, as a result, transfer of an instruction from one stage to the next can take a variable number of cycles. However, this channel width of the crossbar can be varied to trade-off performance with area. In addition to the forward data path connections, pipeline feedback loops in SNS (branch mispredict, register writeback) also need to go through similar switches. With the aid of these crossbars, different SNSs within a SN multicore can share their stages with each other. For instance, the result from, say, SNS A's execute stage, might need to be directed to SNS B's issue stage for the writeback. Due to the introduction of crossbar switches, SNS has three fundamental challenges to overcome:

1. *Global Communication:* Global pipeline stall/flush signals are fundamental to the functionality of a pipeline. Stall signals are sent to all the stages for cases such

as multi-cycle operations, memory access, and other hazards. Similarly, flush signals are necessary to squash instructions that are fetched along mispredicted control paths. In SNS, all the stages are decoupled from each other, and global broadcast is infeasible.

2. *Forwarding:* Data forwarding is a crucial technique used in a pipeline for avoiding frequent stalls that would otherwise occur because of data dependencies in the instruction stream. The data forwarding logic relies on precisely timed (in an architectural sense) communication between execute and later stages using combinational links. With variable amounts of delay through the switches, and the presence of intermediate buffers, forwarding logic within SNS is not feasible.
3. *Performance:* Lastly, even if the above two problems are solved, communication delay between stages is still expected to result in a hefty performance penalty.

The rest of this section will discuss how the SNS design overcomes these challenges (Section 2.3.2) and propose techniques that can recover the expected loss in performance (Section 2.3.3).

### 2.3.2 Functional Needs

**Stream Identification:** The SNS pipeline lacks global communication signals. Without global stall/flush signals, traditional approaches to flushing instructions upon a branch mispredict are not applicable. The first addition to the basic pipeline, a stream identification register, targets this problem.

The SNS design shown in Figure 2.4 has certain components that are shaded in order to distinguish the ones that are not found in a traditional pipeline. One of these additional

components is a *stream identification* (`sid`) register in all the stages. This is a single bit register and can be arbitrarily (but consistently across stages) initialized to 0 or 1. Over the course of program execution, this value changes whenever a branch mispredict takes place. Every in-flight instruction in SNS carries a stream-id, and this is used by the stages to distinguish the instructions on the correctly predicted path from those on the incorrect path. The former are processed and allowed to proceed, and the latter are squashed. A single bit suffices because the pipeline model is in-order and it can have only one resolved branch mispredict outstanding at any given time. All other instructions following this mispredicted branch can be squashed. In other words, the stream-id works as a cheap and efficient mechanism to replace the global branch mis-predict signal. The details of how and when the `sid` register value is modified are discussed below on a stage-by-stage basis:

- *Fetch:* Every new instruction is stamped with the current value stored in the `sid` register. When a branch mis-predict is detected (using the branch update from execute/memory stage), it toggles the `sid` register and flushes the program counter. From this point onwards, the instructions fetched are stamped with the updated stream-id.
- *Decode:* The `sid` register is updated from the stream-ids of the incoming instructions. If at any cycle, the old stream-id stored in decode does not match the stream-id of an incoming instruction, a branch mispredict is implied and decode flushes its instruction buffer.
- *Issue:* It maintains the `sid` register along with an additional 1-bit `last-sid` register. The `sid` register is updated using the stream-id of the instruction that performs register writeback. And, the `last-sid` value is updated from the stream-id of the

last successfully issued instruction. For an instruction reaching the issue stage, its stream-id is compared with the `sid` register. If the values match, then it is eligible for issue. A mismatch implies that some branch was mispredicted, in the recent past, and further knowledge is required to determine whether this new incoming instruction is on the correct path or the incorrect path. This is where the `last-sid` register becomes important. A mismatch of the new instruction's stream-id with the `last-sid` indicates that the new instruction is on the corrected path of execution and hence it is eligible for issue. A match implies the otherwise and the new instruction is squashed. The complete significance of `last-sid` will be made clear later in this section.

- *Execute/Memory:* compares the stream-id of the incoming instructions to the `sid` register. In the event of a mismatch, the instruction is squashed. A mispredicted branch instruction toggles its own stream-id along with the `sid` register value stored here. This branch resolution information is sent back to the fetch stage, initiating a change in its `sid` register value. The mispredicted branch instruction also updates the `sid` in the issue stage during writeback. Thus, the cycle of updates is completed.

To summarize, under normal operating conditions (i.e. no mispredicts), instructions go through the switched interconnection fabric, get issued, executed and write back computed results. When a mispredict occurs, using the stream-id mechanism, instructions on the incorrect execution path can be systematically squashed in time.

**Scoreboard:** The second component required for proper functionality of SNS is a scoreboard that resides in the issue stage. A scoreboard is essential in this design because a forwarding unit (that normally handles register value dependencies) is not feasible.

More often than not, a scoreboard is already present in a pipeline's issue stage for hazard detection. In such a scenario, only minor modifications are needed to tailor a conventional scoreboard to the needs of a SNS pipeline.

The SNS pipeline needs a scoreboard in order to keep track of the registers that have results outstanding and are therefore invalid in the register file. Instructions for which one or more input registers are invalid can be stalled in the issue stage. The SNS scoreboard table has two columns (see Figure 2.10c), the first to maintain a *valid* bit for each register, and second to store the *id* of the last modifying instruction. In case of a branch mis-predict, the scoreboard needs to be wiped clean since it gets polluted by instructions on the wrong path of execution. To recognize a mis-predict, the issue stage maintains a `last-sid` register that stores the stream-id of the last issued instruction. Whenever the issue stage finds out that the new incoming instruction's stream-id differs from `last-sid`, it knows that a branch mis-predict has taken place. At this point, the scoreboard waits to receive the writeback, if it hasn't received it already, for the branch instruction that was the cause of the mis-predict. This branch instruction can be easily identified because it will bear the same stream-id as the new incoming instruction. Finally, after this waiting period, the scoreboard is cleared and the new instruction is issued.

**Network Flow Issues:** In SNS, the stalls are automatically handled by maintaining network back pressure through the switched interconnection. A crossbar does not forward values to the buffer of a subsequent stage if the stage is stalled. This is similar to the way network queues handle stalls. In our implementation, we guarantee that an instruction is never dropped (thrown away) by a buffer.

The transfer of operations between the stages of a SNS takes place over crossbar switches.

If an instruction along with its operand values is, say, 90 bits in size. Then, a crossbar switch with the channel width of 32-bits, will take 3 cycles to transfer this instruction from stage A to B. While this instruction is being transmitted, stage A cannot work on its next instruction, and similarly stage B also sits idle waiting on the transfer to complete. A straightforward solution to this problem is to use a widely popular design known as double buffer / ping-pong buffer. In this design, as the name implies, two buffers are kept and the producer (consumer) switches back-and-forth between the two buffers for storing (retrieving) the value. We use double buffers at all stage inputs and outputs within a SNS.

### 2.3.3 Performance Enhancement

The additions to SNS discussed in the previous section brings the design to a point where it is functionally correct. In order to compare the performance of this *basic* SNS design to an in-order pipeline, we conducted some experiments using a cycle accurate simulator developed in the Liberty Simulation Environment [113]. *Basic* here implies a SNS pipeline that is configured with the stream identification logic, scoreboard, and double buffering. Interested readers can find the details of our simulation setup and benchmarks in Section 2.5.1. The performance of a basic SNS pipeline (first bar) in comparison to the baseline is shown in Figure 2.5. The results are normalized to the runtime of the baseline in-order processor. On average, a 4X slowdown was observed, which is a significant price to pay in return for the reconfiguration flexibility. However, in this version of the SNS design, much is left on the table in terms of performance. Most of this performance is lost in the stalls due to 1) the absence of forwarding paths and 2) transmission delay through the switches.

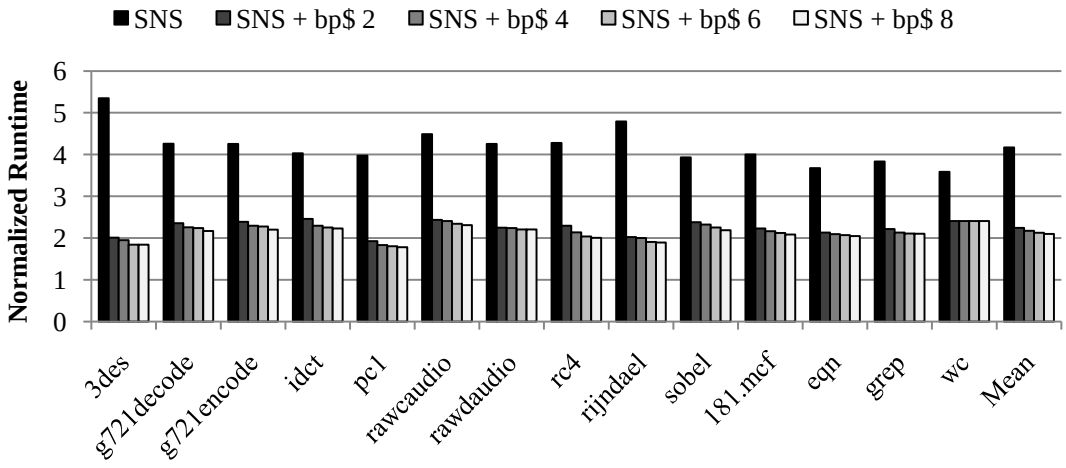



**Figure 2.5: SNS performance normalized to the baseline. Different configurations of SNS are evaluated, both with and without the bypass cache. The slowdown reduces as the bypass cache size is increased (fewer issue-stage stalls).**

**Bypass Cache:** Due to the lack of forwarding logic in SNS, frequent stalls are expected for instructions with register dependencies. To alleviate the performance loss, we add a *bypass cache* in the execute/memory stage (see Figure 2.10d). This cache stores values generated by recently executed instructions within the execute/memory stage. The instructions that follow can use these cached values and need not stall in issue waiting for writeback. In fact, if this cache is large enough, results from every instruction that has been issued, but has not written back, can be retained. This would completely eliminate the stalls arising from register dependencies emulating forwarding logic.

A FIFO replacement policy is used for this cache because older instructions are less likely to have produced a result for an incoming instruction. The scoreboard unit in the issue stage is made aware of the bypass cache size when the system is first configured. Whenever the number of outstanding registers in the scoreboard becomes equal to this cache size, instruction issue is stalled. In all other cases, the instruction can be issued as all of its input dependencies are guaranteed to be present within the bypass cache. Hence,

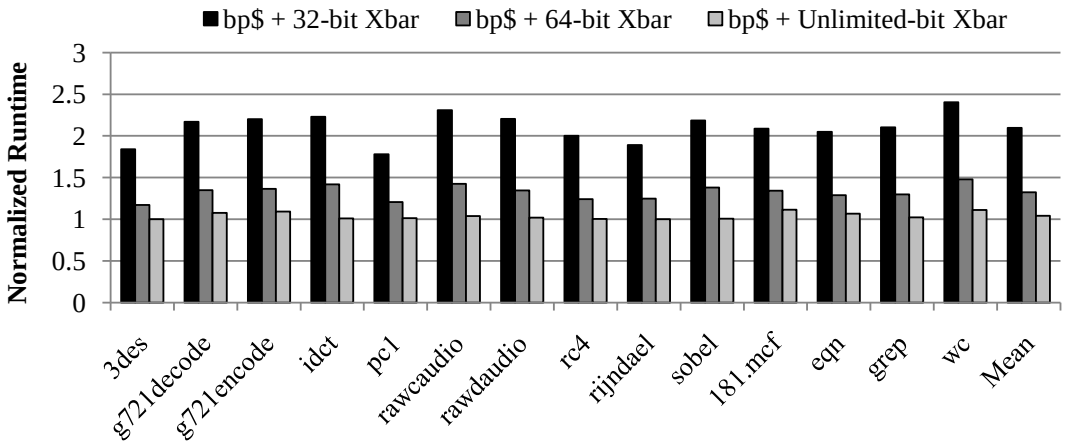



**Figure 2.6: A SNS pipeline, with variation in the transmission bandwidth. The performance improves with the increasing transmission bandwidth, and almost matches the base pipeline at unlimited bandwidth.**

the scoreboard can accurately predict whether or not the bypass cache will have a vacancy to store the output from the current instruction. Furthermore, the issue stage can perform selective register operand fetch for only those values that are not going to be available in the bypass cache. By doing this, the issue stage can reduce the number of bits that it needs to transfer to the execute/memory stage.

As evident from the experimental results (Figure 2.5), the addition of the bypass cache results in dramatic improvements in the overall performance of SNS. The biggest improvement comes between the SNS configuration without any bypass cache (first bar) to the one with a bypass cache of size 2 (second bar). This improvement diminishes after a while, and saturates beyond 8 entries. The average slowdown hovers around 2.1X with the addition of the bypass cache.

**Crossbar Width:** The crossbar channel width is the number of bits that can be transferred to/from the crossbar in a single cycle. In the context of SNS, it determines the number of cycles it will take to transfer an instruction between the stages. The results

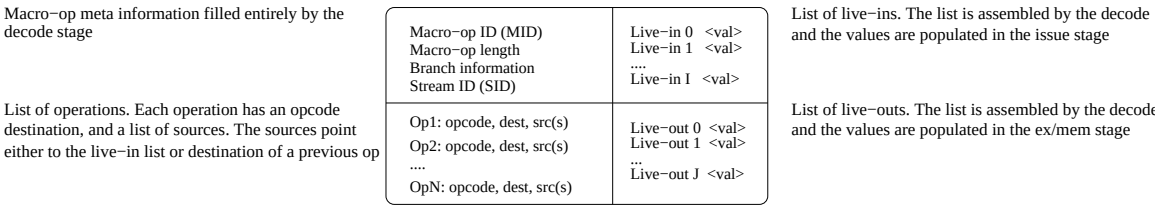



**Figure 2.7: Structure of a macro-op (MOP).**

presented so far in this section have been with a crossbar channel width of 32-bits. Figure 2.6 illustrates the impact of varying this width on performance. Three data points are presented for every benchmark: a 32-bit channel width, a 64-bit channel width, and infinite channel width. A large performance gain is seen when going from 32-bit width to 64-bit width. Infinite bandwidth essentially means eliminating all transfer latency between the stages, resulting in performance comparable to the baseline (however, at a tremendous area cost). With a 64-bit crossbar switch, SNS has an average slowdown of about 1.35X. The crossbar-width discussion is revisited after the next performance enhancement.

**Macro Operations:** The performance of the SNS design suffers significantly from the overhead of transferring instructions between stages, since every instruction has to go through a switched network with a variable amount of delay. Here, a natural optimization would be to increase the granularity of communication to a bundle of multiple operations, that we call a macro-op (*MOP*). There are two advantages of doing this:

1. More work (multiple instructions) is available for the stages to work on while the next MOP is being transmitted.
2. MOPs can eliminate the temporary intermediate values generated within small sequences of instructions, and therefore give an illusion of data compression to the underlying interconnection fabric.

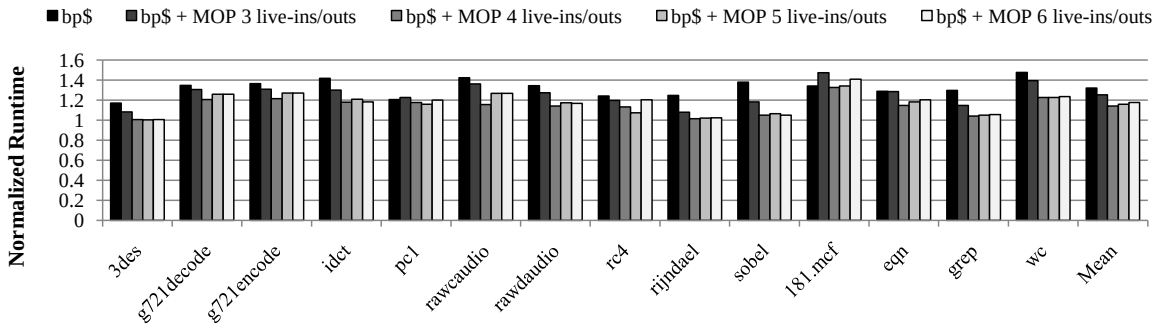



**Figure 2.8: SNS with a bypass cache and the capability to handle MOPs, compared to the baseline in-order pipeline. The first bars are for MOP sizes fixed at 1, while the other bars have constraint on the number of live-ins and live-outs.**

These collections of operations can be identified both statically (at compile time) or dynamically (in the hardware). To keep the overall hardware overhead low, we form these statically in the compiler. Our approach involves selecting a subset of instructions belonging to a basic block, while bounding two parameters: 1) the number of live-ins and live-outs and 2) the number of instructions. We use a simple greedy policy, similar to [26], that maximizes the number of instructions, while minimizing the number of live-ins and live-outs. When forming MOPs, as long as the computation time in the stages can be brought closer to the transfer time over the interconnection, it is a win.

The complete structure of a macro-op is shown in the Figure 2.7. The compiler embeds the MOP boundaries, internal data flow, and live-in/live-out information in the program binary. During runtime, the decode stage's *Packer* structure is responsible for identifying and assembling MOPs. Leveraging hints for the boundaries that are embedded in the program binary, the Packer assigns a unique MOP id (MID) to every MOP flowing through the pipeline. All other stages in the SNS are also slightly modified in order to work with these MOPs instead of simple instructions. This is particularly true of the execute/memory stage where a controller cycles across the individual instructions that comprise a MOP, executing

them in sequence. However, the bandwidth of the stages is not modified, and they continue to process one instruction per cycle. This implies that register file ports, execution units, memory ports etc. are not increased in their number or capability.

The performance results shown in Figure 2.8 are for a SNS pipeline with the bypass cache, 64-bit switch channel width and MOPs. The various bars in the plot are for different configurations of the MOP selection algorithm. The results show that beyond a certain limit, relaxing the MOP selection constraints (live-ins and live-outs) does not result in performance improvement. Prior to reaching this limit, relaxing constraints helps in forming longer MOPs, thereby balancing transfer time with computation time. Beyond this limit, relaxing constraints does not result in longer MOPs. Instead it produces wider MOPs that have more live-ins/outs, which increases transfer time without actually increasing the number of distinct computations that are encoded. On average, the best performance was observed for live-ins/outs constraint of 4. This yielded 1.14X slowdown for a SNS pipeline over the baseline. The worst performers were the benchmarks that had very poor branch prediction rates. In fact, the performance on SNS was found to be strongly correlated with the number of mispredicts per thousand instructions. This is expected because the use of MOPs, and the additional cycles spent for data transfer between stages, causes the SNS pipeline to behave like a very deep pipeline.

**Crossbar width optimization:** The bandwidth requirement at each SNS switch interface is not the same. For instance, macro-ops that are transmitted from decode to issue stage do not have any operand values. But, the ones that go from issue to execute/memory stage hold the operand values read from the register file, making them larger. This observation can be leveraged to optimize the crossbar widths between every pair of stages,

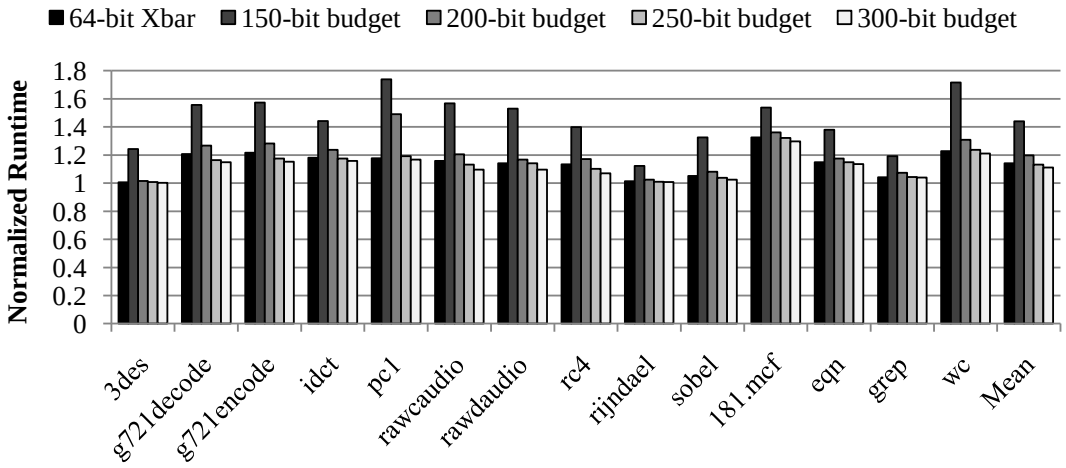



**Figure 2.9: Performance comparison with different budgets for crossbar widths. A budget of 150-bit implies that all interfaces can have a combined width of 150. The first bar is for static assignment of 64-bit crossbars at all interfaces, which is equivalent to a 320-bit ($64 \times 5$) budget. Optimized assignment of 300-bits is able to deliver better performance than 320-bit static assignment.**

resulting in an overall area saving.

A series of experiments were conducted to track the number of bits transmitted over each crossbar interface (fetch-decode, decode-issue, issue-execute, execute-issue and execute-fetch) for every MOP. The average number of bits transmitted varied from 32 to 87. Given a fixed budget of total crossbar-width (across all interfaces), a good strategy is to allocate width to each interface in proportion to the number of bits it transfers. The result of applying this optimization to the SNS pipeline is shown in Figure 2.9. For nearly the same crossbar area (budget of 300-bits), the optimized assignment of crossbar-widths is able to deliver 3% performance improvement over uniform usage of 64-bit crossbars (equivalent to 320-bits in total). With this final performance enhancement, the SNS pipeline slowdown stands at about 1.11X of the baseline.

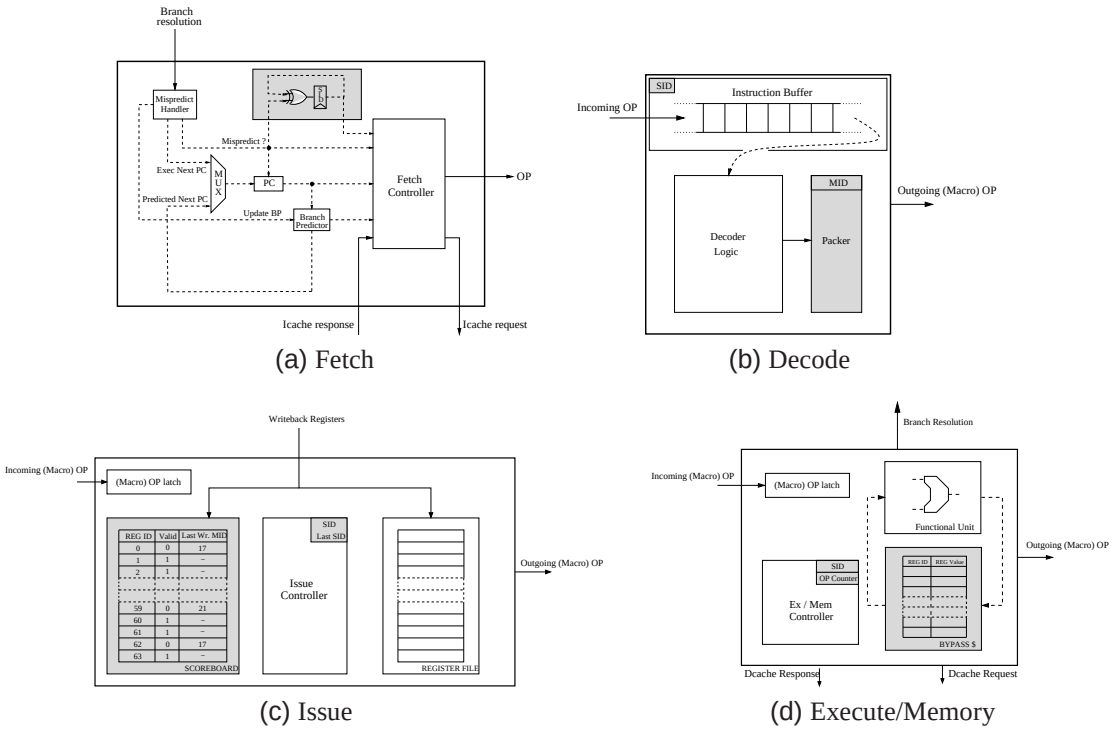



**Figure 2.10: Pipeline stages of SNS. Gray blocks highlight the modules added for transforming a traditional pipeline into SNS.**

### 2.3.4 Stage Modifications

This section goes over the pipeline stages in SNS, and summarizes the modules added to each of them.

- *Fetch:* The modifications made here are restricted to the addition of `sid` register and a small amount of logic to toggle it upon branch mis-predicts (Figure 2.10a).
- *Decode:* The decode stage (Figure 2.10b) collects the fetched instructions in a buffer. An instruction buffer is a common structure found in most pipeline designs, and to that we add our `sid` register. For an incoming instruction with a different stream-id, this register is toggled and the instruction buffer is flushed. The decode stage is also augmented with the Packer. The Packer logic reads instructions from the buffer, identifies the MOP boundaries, assigns them a MID, and fills out the MOP structure

attributes such as length, number of operations and live-in/out register names.

- *Issue:* The issue stage (Figure 2.10c) is modified to include a Scoreboard that tracks register dependencies. For a MOP that is ready for issue, the register file is read to populate the live-ins. The issue stage also maintains two 1-bit registers: `sid` and `last-sid`, in order to identify branch mis-predicts and flush the Scoreboard at appropriate times.
- *Execute/Memory:* The execute/memory stage (Figure 2.10d) houses the bypass cache that emulates the job of forwarding logic. This stage is also the first to update its `sid` register upon a branch mis-predict. In order to handle MOP execution, the execute/memory controller is modified to walk the MOP instructions one at a time (one execution per cycle). At the same time, the computed results are saved into the bypass cache for later use.

## 2.4 The StageNet Multicore

The SNS presented in the last section is in itself a complete microarchitectural solution to allow pipeline stage level reconfiguration. By maintaining cold spares for stages that are most likely to fail, a SNS-based design can achieve the lifetime enhancement targets projected in Figure 2.2. However, these gains can be greatly amplified, without the cold sparing cost, by using multiple SNSs as building blocks to form a StageNet (SN) multicore.

The high level abstraction of SN (Figure 2.3), in combination with the SNS design, forms the basis of the SN multicore (Figure 2.11). The resources within this are not bound to any particular slice and can be connected in any arbitrary fashion to form log-

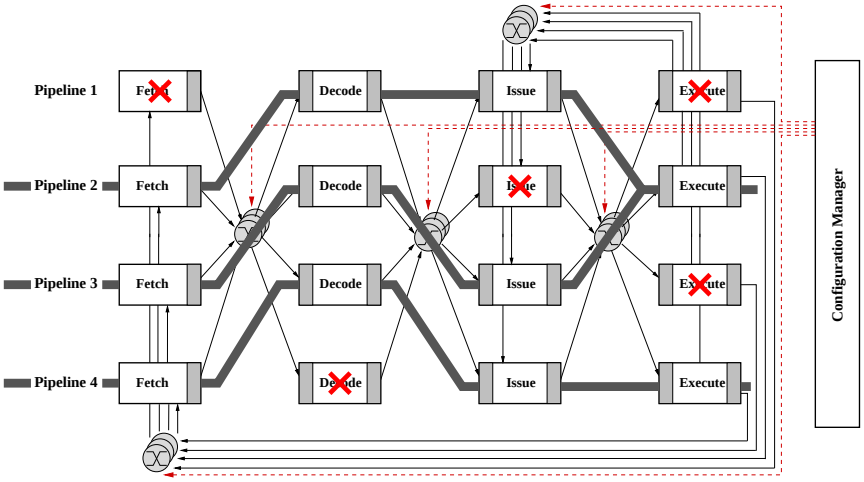



**Figure 2.11: A SN multicore formed using four SNSs. As an example, a scenario with five broken stages is shown (crosses indicate broken stages). Faced with a similar situation, a regular CMP will lose all its cores. However, SN is able to salvage three operational SNSs, as highlighted by the bold lines (note that these bold lines are not actual connections). The configuration manager is shown for illustrative purposes, and is not an actual hardware block.**

ical pipelines. The SN multicore has two prominent additions to glue SNSs together:

1. *Interconnection Switch:* The role of the crossbar switch is to direct the incoming MOP to the correct destination stage. For this task, it maintains a static routing table that is addressed using the thread-id of the MOP. The thread-id uniquely determines the destination stage for each thread. To circumvent the danger of having them as single points of failure, multiple crossbars can be maintained by the SN multicore.
2. *Configuration Manager:* Given a pool of stage resources, the configuration manager divides them into logical SNSs. The configuration manager logic is better suited for a software implementation since: 1) it is accessed very infrequently (only when new faults occur), and 2) more flexibility is available in software to experiment with resource allocation policies. The configuration manager can be designed as a firmware/kernel module. When failures occur, a trap can be sent to the virtualization/OS interface, which can then initiate updates for the switch routing tables.

In the event of any stage failure, the SN architecture can initiate recovery by combining live stages from different slices, i.e. salvaging healthy modules to form logical SNSs. We refer to this as the *stage borrowing* (Section 2.4.1). In addition to this, if the underlying stage design permits, stages can be time-multiplexed by two distinct SNSs. For instance, a pair of SNSs, even if one of them loses its *execute* stage, can still run separate threads while sharing the single live *execute* stage. We refer to this as *stage sharing* (Section 2.4.2).

### 2.4.1 Stage Borrowing

A pipeline stage failure in the system calls upon the configuration manager to determine the maximum number of *full* logical SNSs that can be formed using the pool of live stages. *Full* SNS here implies a SNS with exclusive access to exactly one stage of each type. The number of such SNSs that can be formed by the configuration manager is determined by the stage with the fewest live instances. For example, in Figure 2.11, the bottom two SNSs have a minimum of one stage alive of each type, and, thus, one logical SNS is formed. The logical slices are highlighted using the shaded path indicating the flow of the instruction streams.

It is noteworthy that all four slices in Figure 2.11 have at least one failed stage, and therefore, a multicore system in a similar situation would have lost all working resources. Hence, SN's ability to efficiently borrow stages from different slices, gives it the competitive edge over a traditional multicore.

### 2.4.2 Stage Sharing

Stage borrowing is good, but it is not enough in certain failure situations. For example, the first stage failure in the SN fabric reduces the number of logical SNSs by one. However, if the stages can be time-multiplexed by multiple SNSs, then the same number of logical SNSs can be maintained. Figure 2.11 has the top two logical SNSs sharing an *execute* stage. The number of logical SNSs that can share a single stage can be tuned in our implementation.

The sharing is beneficial only when the threads involved present opportunities to interleave their execution. Therefore, threads with very high IPC (instructions per cycle) are expected to derive lesser benefit compared to low IPC threads. Furthermore, as the degree of stage sharing is increased, the benefits are expected to shrink since more and more threads will contend for the available stage. In order for the stages to be shareable, certain hardware modifications are also required:

- *Fetch:* It needs to maintain a separate program counter for each thread and has to time-multiplex the memory accesses. The instruction cache, in turn, will also be shared implicitly by the executing threads
- *Decode:* The instruction buffer has to be partitioned between different threads.
- *Issue:* The scoreboard and the register file are populated with state values specific to a thread, and it is not trivial to share them. There are two ways to handle the sharing for these structures: 1) compile the thread with fewer registers or 2) use a hardware structure for register caching [82]. In our evaluation, we implement the register caching in hardware and share it across multiple threads.

- *Execute/Memory:* The bypass cache is statically partitioned between the threads. Similarly, the data cache gets shared by the threads.

### 2.4.3 Fault Tolerance and Reconfiguration

SN relies on a fault detection mechanism to identify broken stages and trigger reconfiguration. There are two possible solutions for detection of permanent failures: 1) continuous monitoring using sensors [16, 56] or 2) periodic testing for faults. The discussion of exact mechanism for detection is beyond the scope of this chapter. The configuration manager is invoked whenever any stage or crossbar switch is identified to be defective. Depending upon the availability of working resources, configuration manager determines the number of logical SNSs that can be formed. It also configures the stages that need to be shared and partitions their resources accordingly between threads. While working with higher degrees of sharing, the configuration manager employs a fairness policy for resource allocation, so that the work (threads) gets evenly divided among the stages. For example, if there are five threads that need to share three live stages of same type, the fairness policy prefers a 2-2-1 configuration (two threads each to stages 1 and 2 and remaining one to stage 3) over a 3-1-1 configuration (three threads to stage 1, one each to stages 2 and 3).

## 2.5 Results and Discussion

### 2.5.1 Simulation Setup

The evaluation infrastructure for the SN architecture consisted of three major components: 1) a compilation framework, 2) an architectural simulator, and 3) a Monte Carlo

simulator for lifetime throughput estimations. A total of 14 benchmarks were selected from the embedded and desktop application domains. For these evaluations, the emphasis was on the embedded benchmarks because the SNS is based on an in-order embedded core. A variety of these were used including several encryption (3des, pc1, rc4, rijndael), audio processing (g721encode, g721decode, rawcaudio and rawdaudio), and image/video processing (idct, sobel) benchmarks. In addition, four desktop benchmarks (181.mcf, eqn, grep, wc) were also included in order to exhibit the potential of this architecture for other domains.

We use the Trimaran compilation system [111] as our first component. The MOP selection algorithm is implemented as a compiler pass on the intermediate code representation. During this pass, the code is augmented with the MOP boundaries and other miscellaneous attributes. The final code generated by the compiler uses the HPL-PD ISA [58].

The architectural simulator for the SN evaluation was developed using the Liberty Simulation Environment (LSE) [113]. A functional emulator was also developed for the HPL-PD ISA within the LSE system. Two flavors of the microarchitectural simulator were implemented in sufficient detail to provide cycle accurate results. The first simulator modeled a simple five stage pipeline, which is also the baseline for our experiments. The second simulator implemented the SN architecture with all the proposed enhancements. Table 5.5 lists the common attributes for our simulations.

The third component of our simulation setup is the Monte Carlo engine that we employ for lifetime throughput study. Each iteration of the Monte Carlo process simulates the lifetime of the SN architecture. The configuration of the SN architecture is specified in Table 5.5. The MTTF for the various stages and switches in the system was calculated

using Equation 2.1[1]. The crossbar switch peak temperature was taken from [94] that performs interconnection modeling for the RAW multicore chip [65]. The stage temperatures were extracted from HotSpot simulations of the OR1200 core with the ambient temperature normalized to the one used in [94]. The calculated MTTFs are used as the mean of the Weibull distributions for generating a time to failure (TTF) for every module (stage/switch) in the system. For each iteration of the Monte Carlo, the system gets reconfigured over its lifetime whenever a failure is introduced. The instantaneous throughput of the system is computed for each new configuration using the architectural simulator on multiple random benchmark subsets. From this, we obtain the system throughput over the lifetime. 1000 such iterations are run for conducting the Monte Carlo study.

**Table 2.1: Architectural attributes.**

| | |
|---|---|
| Base core | 5-stage in-order pipeline |
| SNS | 4-stage in-order, with double buffering and all other performance enhancements |
| Branch pred. | global, 16-bit history, gshare predictor BTB size of 2KB |
| L1 I$, D$ | 4-way, 16 KB, 1 cycle hit latency |
| L2 $ unified | 8-way, 64 KB, 5 cycle hit latency |
| Memory | 40 cycle hit latency |

### 2.5.2 Simulation Results

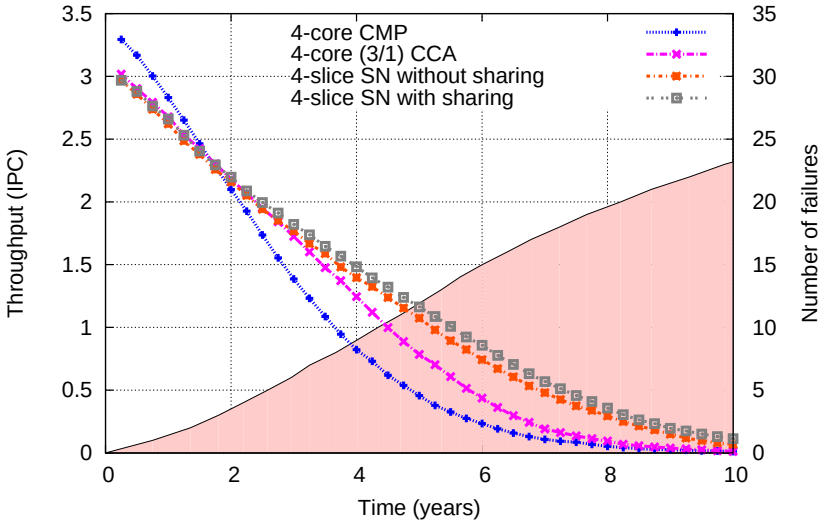
**Lifetime performance:** Figure 2.12a shows the lifetime throughput results for a 4-core CMP compared against two equally provisioned configurations of the SN architecture. The CMP system starts with a performance advantage over the SN architecture. However, as failures accumulate over time, the throughput of SN overtakes the baseline performance and thereafter remains dominant. For instance, at year 6, the throughput of SN is nearly

[1]The fetch stage was qualified to have a MTTF of 10 years. This is a conservative estimate and no actual module level MTTF values are available from any public source.

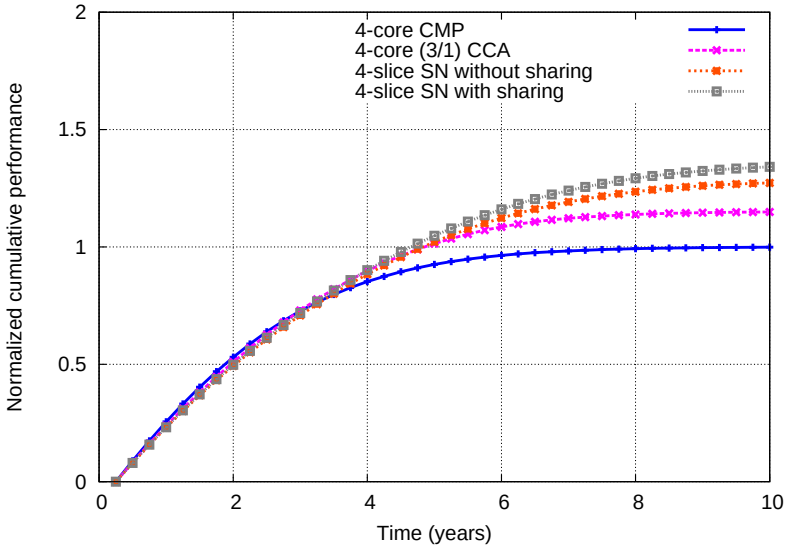
3X the baseline CMP. The shaded portion in this figure depicts the cumulative distribution function (CDF) of the combined MTTF Weibull distributions. For instance, this plot shows that after 8 years, on average, there are 20 failed structures in the system. The difference between SN configuration with and without sharing was found to be almost negligible. Remaining results in the chapter are for SN configuration without sharing.

Figure 2.12b shows the cumulative performance (total work done) for SN configurations compared against the baseline. By the end of the lifetime, we achieve as much as 37% improvement in the work done for the SN fabric. About 30% of this is achieved by *stage borrowing* only, and the additional 7% benefit is a result of *stage sharing*. The sharing was not found to be very effective as the opportunities to time-multiplex stages were very few and far between.

**Area overhead:** The area overhead in the SN arises from the additional microarchitectural structures that were added and the interconnection fabric composed of the crossbar switches. Area overhead is shown using an OR1200 core as the baseline (see Section 2.2.1). The area numbers for the bypass cache and register cache are estimated by taking similar structures from the OR1200 core and resizing them appropriately. More specifically, bypass cache and register cache areas are based on the TLB area, which is also an associative look-up structure. And finally, the area of double buffers is based on the maximum macro-op size they have to store. The sizing of all these structure was done in accordance with the SNS configuration that achieved the best performance. The crossbar switch area is based on the Verilog model from [81]. The total area overhead for the SN design (no sharing) is ~15% (Table 2.2). This was computed assuming six slices share a crossbars, and that each crossbar maintain two cold spares. Note that the scoreboard area is ignored in this




(a) Throughput over time.




(b) Cumulative work done. Cumulative work represents the integral of throughput over time.

**Figure 2.12: Throughput and cumulative performance results for 4-core CMP, 4-slice SN and 4-slice SN with sharing. Plot (a) also shows (shaded portion) the expected number of failed modules (stages/switch) until that point in the lifetime.**

discussion, as the introduction of sufficiently sized bypass cache eliminates the need for them.

All the design blocks were synthesized, placed and routed using industry standard CAD

tools with a library characterized for a 130nm process. The area overhead for separate modules, crossbar switches, and SN configurations is shown in Table 2.2.

**Timing overhead:** Although, we have not investigated the impact of our microarchitectural changes to the circuit critical paths, a measurable influence on the cycle time is not expected in SNS, because: 1) our changes primarily impact the pipeline depth (due to the additional buffers), and 2) all logic changes are local to the stages, and do not introduce any direct (wire) communication between them.

**Table 2.2: Area overhead of SN architecture.**

| **Design Blocks** | | |
|---|---|---|
| **Block name** | **Area** ($mm^2$) | **Percent overhead** |
| Bypass cache | 0.044 | 3.4% |
| Register cache | 0.028 | 2.2% |
| Double buffers | 0.067 | 5.3% |
| Miscellaneous logic | 0.012 | 0.9% |
| 64-bit crossbar switch | 0.028 | 2.1% |
| **SN Configurations** | | |
| **Configuration** | | **Percent overhead** |
| SN without sharing | | 15.1% |
| SN with sharing | | 17.3% |

## 2.6 Related Work

Concern over reliability issues in future technology generations has spawned a new wave of research in reliability-aware microarchitectures. Recent work has addressed the entire spectrum of reliability topics, from fault detection and diagnosis to system repair and recovery. This section focuses on the most relevant subset of work, those that propose architectures that tolerate and/or adapt to the presence of faults.

High-end server systems designed with reliability as a first-order design constraint have been around for decades but have typically relied on coarse grain replication to provide

a high degree of reliability [14, 100], such as Tandem NonStop [12], Teramac [30, 5], Stratus [117], and the IBM zSeries [12]. However, dual and triple modular redundant systems incur significant overheads in terms of area and power. Furthermore, these systems still remain susceptible to wearout-induced failures since they cannot tolerate a high failure rate.

Configurable Isolation [2] is another high level solution that works by disabling broken cores as soon as they develop a fault. ElastIC [105] and Maestro [38] are more resource conscious architectural vision for multiprocessor fault tolerance. Exploiting low-level circuit sensors for monitoring the health of individual cores, these papers propose dynamic reliability management that can throttle and eventually turn off cores as they age over time. Although effective in a limited failure rate scenario, all of these proposals need a large number of redundant cores, without which they face the prospect of rapidly declining processing throughput as faults lead to core disabling.

An alternative to core disabling is fine-grained redundancy for defect tolerance. There are numerous proposals for fine-grained redundancy maintenance such as Bulletproof [29], sparing in array structures [18], branch predictors [18], register files [95], functional units [19, 103], and other such microarchitectural structures. These schemes typically rely on inherent redundancy of superscalar cores or add cold spares for vulnerable microarchitectural modules.

In a multicore chip, the concept of architectural core salvaging [83] can partially mimic the benefits of maintaining spares. Architectural core salvaging leverages natural cross-core redundancy and migrates threads to a healthy core whenever a broken core encounters an instruction it cannot execute. However, given that a critical fraction of the core logic

is non-redundant [83] (~80% in Intel Core2 like architectures), schemes that depend on salvaging microarchitectural and architectural can provide only a limited fault coverage.

Much work has also been done in building reliable systems from FPGA components. The Teramac Configurable Computer [5] is one instance of this paradigm. The Teramac Custom Computer is designed to tolerate defective resources using specialized algorithms which identify legal mappings of user workloads that also avoid faulty components. Other research has focused on building reliable substrates out of future nanotechnologies that are expected to be inherently fault-prone. The NanoBox Processor Grid [61] was designed as a recursive system of black box devices, each employing their own unique fault tolerance mechanisms. While this project does boast a significant amount of defect tolerance, it comes at a 9X overhead in terms of redundant structures.

SN differs dramatically from solutions previously proposed in that our goal is to minimize the amount of hardware used solely for redundancy. More specifically, we enable reconfiguration at the granularity of a pipeline stage, and allow pipelines to share their stages, making it possible for a single core to tolerate multiple failures at a much lower cost. In parallel to our efforts, Romanescu et al. [88] have proposed a multicore architecture, Core Cannibalization Architecture (CCA), that also exploits stage level reconfigurability. CCA allows only a subset of pipelines to lend their stages to other broken pipelines, thereby avoiding full crossbar interconnection. Unlike SN, CCA pipelines maintain all feedback links and avoid any major changes to the microarchitecture. Although these design choices reduce the overall complexity, fewer opportunities of reconfiguration exist for CCA as compared to SN.

## 2.7 Summary

As CMOS technology continues to evolve, so too must the techniques that are employed to counter the effects of ever more demanding reliability challenges. Efforts in fault detection, diagnosis, and recovery/reconfiguration must all be leveraged together to form a comprehensive solution to the problem of unreliable silicon. This work contributes to the area of recovery and reconfiguration by proposing a radical architectural shift in processor design. Motivated by the need for finer-grain reconfiguration, networked pipeline stages were identified as the effective trade-off between cost and reliability enhancement. Although performance suffered at first as a result of the changes to the basic pipeline, a few well-placed microarchitectural enhancements were able to reclaim much of what was lost. Ultimately, the SN fabric exchanged a modest amount of area overhead (15%) in return for a highly resilient CMP fabric that yielded about 40% more work during its lifetime than a traditional CMP.

# CHAPTER III

# A Scalable Architecture for Wearout and Process Variation Tolerance

## 3.1 Introduction

In an effort to combat the silicon reliability threat expected in future technology generations, the previous chapter introduces a multicore wearout tolerance solution name StageNet (SN). The basic idea of SN is to organize a multicore as a dynamically configurable network of pipeline stages. Logical cores are created at run-time by connecting together one instance of every pipeline stage. The underlying pipeline microarchitecture is designed to be completely decoupled at stage boundaries, providing full flexibility to construct logical cores. In the event of stage failures, the SN architecture initiate recovery by salvaging healthy stages to form logical cores. This ability of SN to isolate failures at a finer granularity (stages rather than cores) forms the basis of its reliability benefits.

Despite all the benefits SN architecture offers, it faces three principal limitations. Firstly, the original proposal was designed for a CMP with 4-8 cores, and does not scale well to a large number of cores (say, 100). The crossbar, that was used as the SN interconnec-

tion, is notorious for steep increases in area and delay overheads as the number ports is increased [81], and therefore limits the SN scaling. Secondly, the SN proposal focuses primarily on the stage failures and does not investigate methods for interconnection fault tolerance. Thus, a scaled up SN system will waste all of its working stages if the shared crossbar between them develops a failure. And finally, the SN design was evaluated for wearout related failures only. A more immediate concern for the industry today is the impact of process variation on the production yield and design efficiency.

Process variation [17, 91] is caused by the inability to precisely control the fabrication process at small-feature technologies. This can lead to significant deviation of circuit parameters (channel length, threshold voltage, wire spacing) from the design specification. These parametric deviations can create a wide distribution of operating characteristics for components within/across chip(s), resulting in slow parts that work at a low frequency to those that are very fast but leaky (high static power). Either extreme is bad for efficient computing.

This chapter introduces StageWeb (SW), a scalable CMP fabric for wearout and process variation tolerance, that eliminates all the aforementioned limitations of SN. The SW system is optimized to determine the best degree of connectivity between pipelines (that can share their resources together), while incurring a modest amount of overhead. A range of interconnection alternatives, and corresponding configuration algorithms, are explored to enable scalable fault-tolerance using SW. The reliability of the interconnection network is also tackled in the SW design through the use of spare crossbars, robust crossbar designs and intelligent connectivity to give an illusion of redundancy. The underlying interconnection flexibility of SW is further leveraged to mitigate process variation. Using SW, the faster

components (pipeline stages) in the fabric can be selectively picked, to form pipelines that can operate at a higher frequency. This ability of SW limits the harmful effects of process variation that intersperse slower components with faster ones throughout a chip.

Contributions of this chapter are as follows:

1. SW, a comprehensive solution for the upcoming reliability challenges - permanent faults and process variation.
2. Exploration of robust and scalable interconnection alternatives for building SW chips.
3. Configuration algorithms to a) maximize the SW system throughput in the face of failures, and b) improve the distribution of core frequencies in the presence of process variation.
4. Comparisons of SW and traditional CMPs on the ground of 1) cumulative work a chip can perform before being decommissioned, 2) throughput guarantees over the lifetime, 3) distribution of core frequencies, 4) energy efficiency, and 5) yield.

## 3.2 Background

The StageWeb architecture proposed in this chapter builds upon StageNet(SN) [43], which is a solution for permanent fault tolerance in multicores. SN creates a network of pipeline stages using full crossbars, thereby allowing mutual exchange of working resources between pipelines. Figure 3.1 illustrates a SN multicore created out of four SNSs that share a common crossbar network. The inherent symmetry of the SN allows arbitrary formation of a logical SNS by grouping together at least one pipeline stage of each type. For instance, fetch, issue and execute stage from slice 0 are linked with the decode from

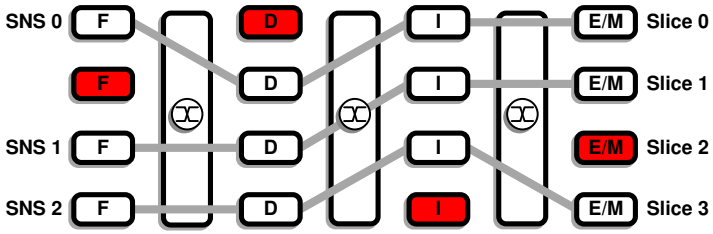



**Figure 3.1: The SN architecture with four slices interconnected to each other. Despite four failed stages (marked by shading), SN is able to salvage three working pipelines, maintaining healthy system throughput. Given a similar fault map, a core-disabling approach for reliability would lose all working resources.**

slice 1, to construct a working pipeline.

SN relies on a fault detection mechanism to identify broken stages and trigger reconfiguration. The manufacture time failures can be easily identified at the test time and SN can be configured accordingly. However, an active mechanism is required to catch failures in the field. There are two possible solutions for detection of permanent failures: 1) continuous monitoring using sensors [16, 56] or 2) periodic testing for faults. SN can employ either of these or use a hybrid approach. In the presence of failures, SN can easily isolate broken stages by adaptively routing around them. Given a pool of stage resources, a software based configuration manager can divide them into a globally optimal set of logical SNSs. In this way, SN's interconnection flexibility allows it to salvage healthy stages from interconnected cores.

### 3.2.1 Limitations of SN

The SN design and analysis, as presented in chapter II, is an acceptable wearout tolerance solution for a small scale multicore system. However, SN is limited in three distinct ways that prevent it from meeting the many-core reliability challenge:

- First, SN was designed for a CMP with 4-8 cores, and does not scale well to a large

number of cores. The crossbar, that was used as the SN interconnection, is notorious for steep growth in area and delay overheads as the number of ports is increased [81], and therefore limits the SN scaling.

- Second, the SN proposal focuses primarily on stage failures and does not investigate methods for interconnection fault tolerance. SN's robustness hinges on the link and crossbar reliability. For instance, a SN chip will waste all of its working stages if the shared crossbar between them develops a failure.

- And finally, the SN design targets only wearout related failures, which constitutes only a part of the reliability challenge. A more immediate concern for the industry today is the accelerating rate of process variation and manufacturing defects, and its impact on the performance-efficiency of semiconductor products.

### 3.2.2 Impact of Process Variation and Defects

Process variation is encountered at manufacturing time, and influences almost every chip manufactured from day one. The variations can be systematic (e.g., lithographic lens aberrations) or random (e.g. dopant density fluctuations), and can manifest at different levels – wafer-to-wafer (W2W), die-to-die (D2D) and within-die (WID). Traditionally, D2D has been the most visible form of variation, and was tackled by introducing the notion of speed-binning (chips are partitioned based on their frequency and sold accordingly). However, the increasing levels of WID variations [91, 66, 108] has created newer challenges for today's multi-core designs. As a result of WID variations, operational frequencies of CMP cores can exhibit a wide distribution (see Figure 3.2). To deal with this disparity, designers

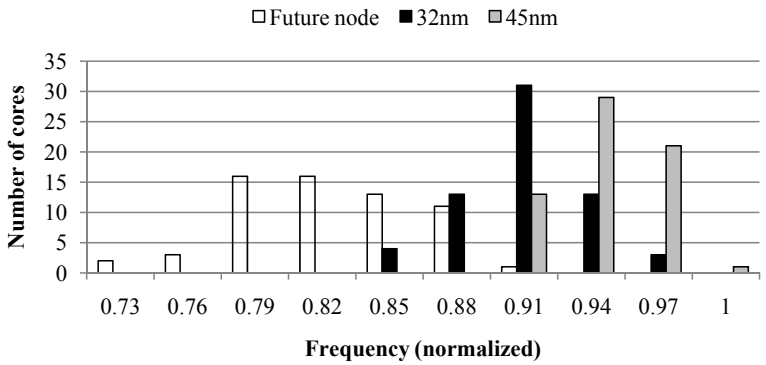



**Figure 3.2: Impact of process variation on a 64-core CMP. The plot shows the distribution of core frequencies at current technology nodes (45nm and 32nm) and the (next-to-arrive) future node. As the technology is scaled, the distribution shifts towards the left (more slower cores) and widens out (more disparity in core frequencies). This is a consequence of large number of cores ending up with slower components, bringing down their operational frequencies.**

in upcoming technology generations may create overly conservative designs or introduce large frequency guard-bands. Both of which are undesirable alternatives for computing efficiently.

An extreme case of process variation is manufacture-time defects. Defect-tolerance for individual cores is a challenging problem. At one extreme is the option to disable cores as soon as they develop a fault [2], we refer to this as *core isolation*. However, with an increase in defect rate, systems with core isolation can exhibit rapid throughput degradation, and quickly become useless. This challenge will manifest itself as a process yield problem (fewer chips with an acceptable throughput). Thus, achieving an economically viable yield will get harder with the rise in the manufacturing defect density. Figure 3.3 shows the yield for a 100 core CMP at a range of defect densities. As evident from this chart, any rise in the defect densities seen today can have a catastrophic impact on yield.

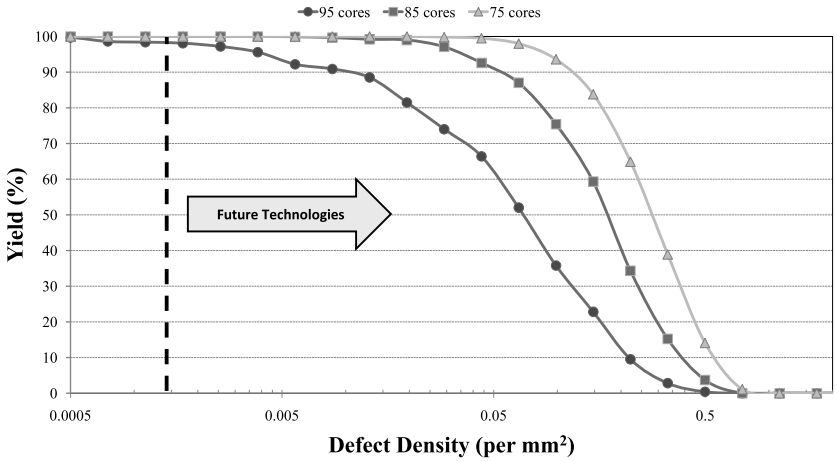



**Figure 3.3: This plots shows the yield for a 100 core CMP at a range of defect densities. The yield is computed as the fraction of *working* chips for a 1000 chip Monte-Carlo simulation (at each defect density point). A *working* chip is one that has greater than 75/85/95 cores functional. The black dotted line shows the currently observed defect density according to the latest ITRS report [53].**

## 3.3 The StageWeb Architecture

SW is a scalable architecture for constructing dependable CMPs. SW *interweaves* pipeline stages and interconnection into an adaptive fabric that is capable of withstanding wearout failures as well as mitigating process variation. The interconnection is designed to be flexible such that the system can react to local failures, reconfiguring around them, to maximize the computational potential of the system at all times. Figure 3.4 shows a graphical abstraction of a large scale CMP employing the SW architecture. The processing area of the chip in this figure consists of a grid of pipeline stages, interconnected using a scalable network of crossbars switches. The pipeline microarchitecture of SW is same as that of a SNS. Any complete set of pipeline stages (reachable by common interconnection) can be assembled together to form a logical pipeline.

The fault-tolerance within SW can be divided into two sub-problems. The first half is to utilize as many working pipeline stages on a chip as possible. And the second half is

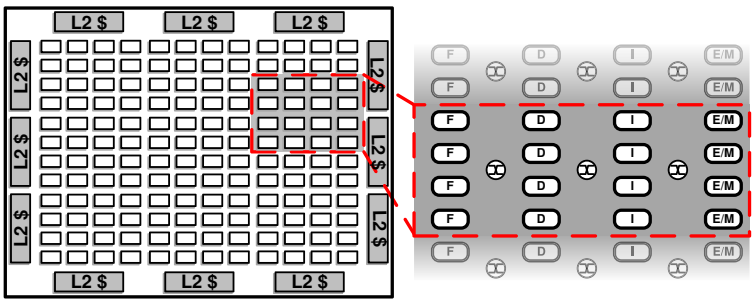



**Figure 3.4: The StageWeb (SW) architecture. The pipeline stages are arranged in form of a grid, surrounded by conventional memory hierarchy. The inset shows a part of the SW fabric. Note that the figure here is an abstract representation and does not specify the actual number of resources.**

to ensure interconnection reliability. A naive solution for the first problem is to provide a connection between all stages. However, as we will show later in this section, full connectivity is not necessary between all stages on a chip to achieve the bulk of reliability benefits. As a combined solution to both these problems, we explore alternatives for the interconnection network, interconnection reliability and present configuration algorithms for SW. The underlying interconnection infrastructure is also leveraged by SW to mitigate process variation. This is accomplished by introducing a few minor changes to the configuration policy.

### 3.3.1 Interweaving Range

The reliability advantages of SW stem from the ability of neighboring slices (or pipelines) to share their resources with one another. Thus, a direct approach for scaling the original SN proposal would be to allow full connectivity, i.e. a logical SNS can be formed by combining stages from anywhere on the chip. However, such flexibility is unnecessary, since the bulk of the reliability benefits are garnered by sharing amongst small groups of stages. To verify this claim, we conducted an experiment with a fixed number of pipeline

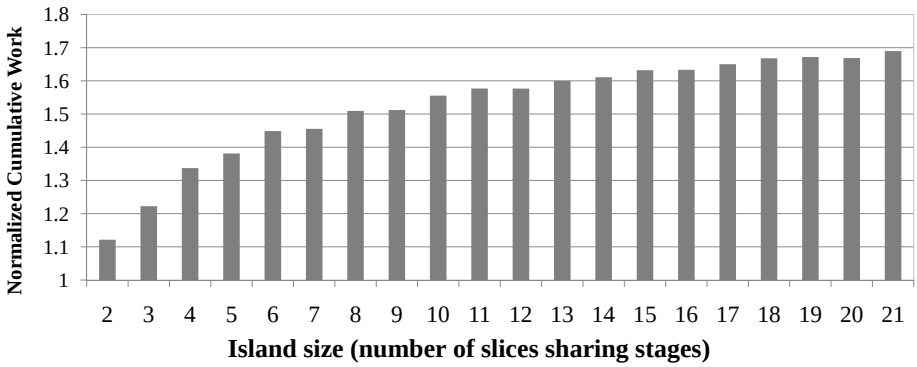



**Figure 3.5: Cumulative work performed by a fixed size SW system with increasing SW island width. The results are normalized to an equally provisioned regular CMP. These results are a theoretical upper bound, as we do not model interconnection failures for the experiments here.**

resources interwoven (grouped together) at a range of values. Each fully connected group of pipelines is referred to as a *SW island*. For instance, 16 pipelines can be interwoven at granularity of 2 pipelines (leading to 8 SW islands), 4 pipelines (4 islands), 8 pipelines (2 islands) or all 16 pipelines can be interwoven together. Figure 3.5 shows the cumulative work done by a large number of slices interwoven at a range of SW island sizes. All SW configurations in this figure have same the amount of pipeline resources. Cumulative work metric, as defined in Section 3.4.3, measures the amount of useful work done by a system in its entire lifetime. Note that the interconnection fabric here is kept fault free for the sake of estimating the upper bound on the throughput offered by SW.

As evident from Figure 3.5, a significant amount of defect tolerance is accomplished with just a few slices sharing their resources. The reliability returns diminish with the increasing number of pipelines, and beyond 10-12 pipelines, interweaving has a marginal impact. This is so because, as a SW island spans more and more slices, the variation in time to failure of its components gets smaller and smaller. This factors into the amount of gains that the flexibility of interconnection can garner in combining working stages, re-

sulting in a diminishing return with an increase in island width. Thus, a two-tier design can be employed for SW by dividing the chip into SW islands, where a full interconnect is provided between stages within the island and no (or perhaps limited) interconnect exists between islands. In this manner, the wiring overhead can be explicitly managed by examining more intelligent system organizations, while garnering near-optimal benefits of the SW architecture.

### 3.3.2 Interweaving Candidates

Interweaving a set of 10-12 pipelines together, as seen in Figure 3.5, achieves a majority of the SW reliability benefits (assuming failure immune interconnection). However, using a single crossbar switch might not be a practical choice for connecting all the 10-12 pipelines together because: 1) area overhead of crossbars scales quadratically with the number of input/output ports, 2) stage to crossbar wire delay increases with the number of pipelines connected together, at some point this can become the critical path for the design, 3) and lastly, failure of the single crossbar shared by all the pipelines can compromise usability of all of them.

In light of the above reasons, there is a need to explore more intelligent interconnections that can reach a wider set of pipelines while keeping the overheads under check. Reaching a wider set of pipelines enables salvaging of more stage resources from farther ends of the chip. The rest of this section introduces four interconnection alternatives for improving the range of pipelines that can share their stages together. In addition, some sort of fault tolerance also needs to be incorporated into the interconnection network, which is addressed later in this section.

**Single Crossbars** The most basic interconnection option is to use the one employed in the original SN proposal, a single full crossbar switch (bufferless). Figure 3.6 shows a full crossbar connecting fetch and decode stages of *n* slices. The figure also shows an abstraction for a crossbar (a bold vertical line) that will be used hereafter in this chapter. By virtue of being bufferless, this crossbar's delay has to fit within a single CPU cycle, thus, the value of *n* (slices connected) is bounded by this delay.

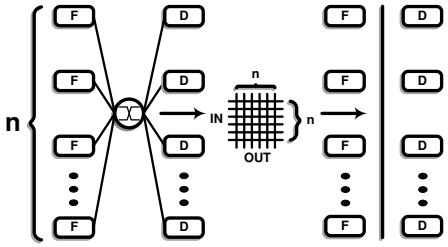



**Figure 3.6: A single crossbar interconnect connecting *n* slices. The diagram on the right also shows the abstraction that we use henceforth for representing a single crossbar.**

**Overlapping Crossbars** The overlapping crossbar interconnection builds upon the single crossbar design, while enabling a wider number of pipelines to share their resources together. As the name implies, adjacent crossbars overlap half of their territories in this interweaving setup. Figure 3.7 illustrates the deployment of overlapping crossbars over $\frac{3}{2}n$ slices. Unlike the single crossbar interconnection, overlapping crossbars have a fuzzy boundary for the SW islands. The shaded stages in the figure highlight a repetitive interconnection pattern here. Note that these $\frac{n}{2}$ stages can connect to the stages above them using crossbars Xbar 1,4,7, and to the stages below them using crossbar Xbar 2,5,8. Thus, overall these stages have a reach of $\frac{3}{2}n$ slices. The overlapping crossbars configuration has two distinct advantages over the single crossbars: 1) it can allow up to 50% more slices to share their resources together, and 2) it introduces an alternative crossbar link at every

stage interface, improving the interconnection robustness (stages can choose an alternative crossbar in case one of them fails) without additional cold sparing.

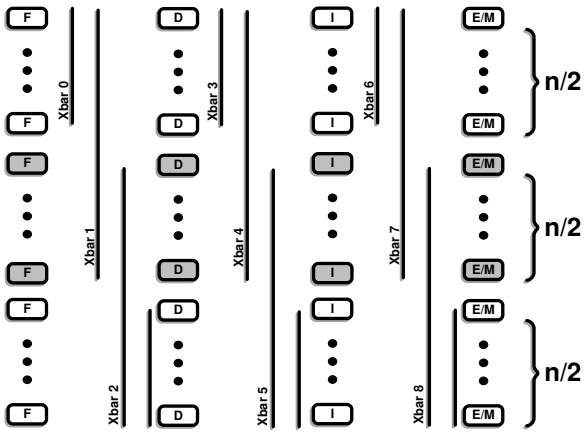



**Figure 3.7: Overlapping crossbar connections. The overlap allows a wider set of pipelines to share their resources. In this figure, the shaded stages in the middle have a reach of $\frac{3}{2}n$ pipelines.**

**Single and Front-Back Crossbars** The primary limitation of single crossbars is the interweaving range they can deliver. This value is bounded by the extent of connectivity a single-cycle crossbar can provide. However, if this constraint is relaxed by introducing two-cycle crossbars, virtually twice the number of slices can communicate with one another. Unfortunately, the two cycle latency between every pair of stages can introduce a significant slowdown on the single thread performance of the logical SNSs (~25%). A compromise solution would be to apply two cycle crossbar at a coarser granularity than pipeline stages. One way to accomplish this is by classifying fetch-decode pair as one block (front-end), and issue-exmem pair as the other (back-end). The single thread performance loss when using this is about 7%. Connecting up these two blocks would need one front-end to back-end crossbar, and the other in the reverse direction. We call such two-cycle interconnections front-back crossbars. Figure 3.8 shows $2n$ slices divided into

front-end and back-end blocks, which are connected by a front-back crossbar (FB-Xbar 0).

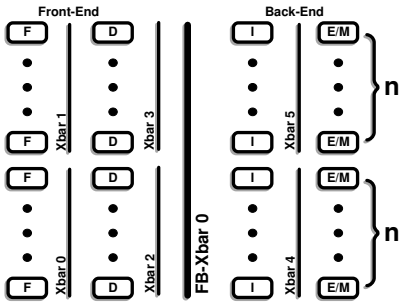



**Figure 3.8: Combined application of single crossbars in conjunction with front-back crossbars. The reverse connections, execute/memory to issue and execute/memory to fetch, are not shown here for the sake of figure readability.**

**Overlapping and Front-Back Crossbars** The single and front-back crossbar combination benefits from the interweaving range it obtains from the front-back crossbar, but, at the expense of single thread performance loss. An alternative is to combine the overlapping crossbars with the front-back crossbars. Figure 3.9 shows this style of interconnection applied over $2n$ slices. In this scenario, $\frac{3}{2}n$ slices can be reached without losing any performance, and the remaining $n/2$ bordering slices can be reached using the front-back crossbars.

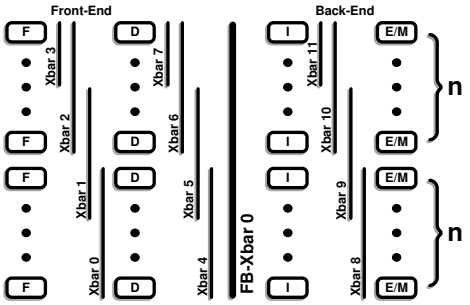



**Figure 3.9: Combined application of overlapping crossbars in conjunction front-back crossbars. The reverse connections are not shown here for the sake of figure readability.**

### 3.3.3 Configuration Algorithms

The faults in a SW chip can manifest as broken stages, crossbar ports or interconnection links. Each of these scenarios demand a reconfiguration of the system such that the defective components are isolated. A good configuration algorithm would guarantee formation of a maximum number of logical pipelines (or SNSs), thus achieving the highest possible system throughput. This section presents three configuration algorithms for handling each type of crossbar deployment, namely, single crossbars, overlapping crossbars and front-back configurations. All four interweaving alternatives discussed in the Section 3.3.2 can be successfully configured by using a combination of these three algorithms. For the sake of avoiding complicated interactions of different types of failures when forming logical SNSs, the algorithm we propose here abstracts all failures as stage failures. For instance, a crossbar port failure can be accounted for by declaring the stage connecting to it as dead. The same abstraction can be applied to interconnection link failures.

**Single Crossbar Configuration** The input to this algorithm is the fault map of the entire SW chip, and it is explained here using a simple example. Figure 3.10 shows a four-wide SW system. The SW islands are formed using the top two and bottom two slices. There are eight defects in this system, four stage failures and four crossbar port/interconnection link failures. The dead stages are marked using a solid shade (F2, D4, I3, E4) and the interconnection as crosses. The stages connected to a dead interconnection are also declared dead, and are lightly shaded (D1, D2, I2). This is to distinguish them from the physically defective stages. For illustration purposes, the backward connections are not shown here and are assumed fault-free.

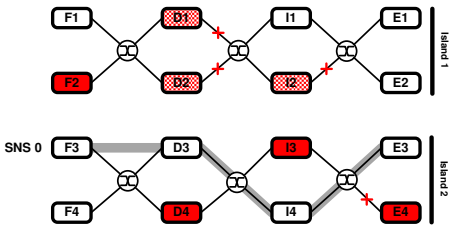



**Figure 3.10: Configuration of SW with single crossbars. The marked stages and interconnections are dead. Island 1 is not able to form any logical SNS, whereas island 2 forms only one logical SNS (SNS 0).**

Given the updated fault-map (with interconnection failures modeled as stage failures), the single crossbar configuration is conducted for one SW island at a time. The first step is to create a list of working stages of each type. For the example in Figure 3.10, this results in - fetch {F1}, decode {}, issue {I1} and execute/memory {E1, E2} - for SW island 1, and - fetch {F3, F4}, decode {D3}, issue {I4} and execute/memory {E3} - for SW island 2. The second step groups unique working stages within an island, and sets them aside as a logical SNS. In our example, this results in having only *one* working SNS: F3, D3, I4, E3, and the configuration is complete.

**Overlapping Crossbar Configuration** The overlapping crossbars provide additional connectivity for resources from two neighboring SW islands. For the explanation of this algorithm, we will use the same SW example from before. Figure 3.11 is almost the same as before, with the exception of a new overlapping crossbar layer in the middle. This layer makes the tally for the number of logical SN islands three. Also, note the change in shading used for stages D2 and I2. The top half of these stages are lightly shaded, and the bottom half is clear. This is to denote that these stages are dead for use in island 1, but are available for use in island 2.

The core of the overlapping crossbar configuration algorithm is same as the one used

for the single crossbar configuration. Given the fault-map, and the proper abstraction of interconnection faults as stage faults, the single crossbar configuration algorithm is used to form logical SNSs for one island at a time. This process is started at one end of the SW fabric, and is swept across the entire SW. When this process is started at the top of the fabric, working stages from the top of the pile within each island are given preference to form logical SNSs. This heuristic helps in keeping more resources free when the succeeding islands are configured. Figure 3.11 illustrates this logical progression from island 1 to island 3 in our example. The steps for each island configuration are detailed below, and result in a total of *two* logical SNSs.

**Island 1:**

1. Free working stages: fetch {F1}, decode {}, issue {I1}, execute/memory {E1,E2}.
2. Logical SNSs: *none.*

**Island 2:**

1. Free working stages: fetch {F3}, decode {D2, D3}, issue {I2}, execute/memory {E2, E3}.
2. Logical SNSs: F3, D2, I2, E2.

**Island 3:**

1. Free working stages: fetch {F4}, decode {D3}, issue {I4}, execute/memory {E3}.
2. Logical SNSs: F4, D3, I4, E3.

**Front-Back Crossbar Configuration** The front-back crossbars are only used to connect the front-end (fetch-decode pair) with the back-end (issue-execute/memory pair). This requires their use to be in conjunction with some other crossbar configuration (see Sec-

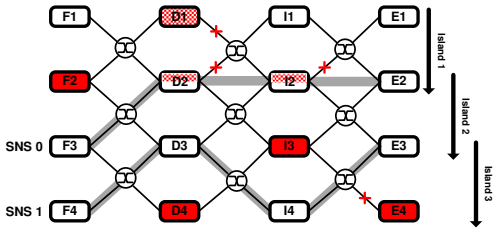



**Figure 3.11: Configuration of SW with overlapping crossbars. The red marked stages and interconnections are dead. The partially marked stages are dead for one island, but are available for use in the other. Island 1 is not able to form any logical SNS, island 2 forms one logical SNS (SNS 0) and island 3 also forms one logical SNS (SNS 1).**

tion 3.3.2). Henceforth, we will refer to this *other crossbar configuration* as the first-level interconnection. Nevertheless, the configuration algorithm for front-back crossbars is independent of the choice made for the first-level interconnection. The running example from the previous algorithms will again be employed in this section (see Figure 3.12). In our example (Figure 3.12) front-back crossbars are assumed to be fault-free. The front-back algorithm can be divided into three phases:

1. **First-level Interconnection:** Prior to configuring front-back crossbars, the maximum potential of the first-level interconnection should be exploited. In our example, we employ overlapping crossbars as the first-level interconnection. This results in forming two logical SNSs: F3, D2, I2, E2 and F4, D3, I4, E3.
2. **Front-back Bundling:** In this step, the resources remaining in the SW fabric are individually bundled up in the front-end and the back-end. Figure 3.12 forms one front-end bundle (F1, D1) and one back-end bundles (I1, E1).
3. **Front-back Integration:** The last phase in the configuration is to combine pairs of front-end and back-end bundles and form logical SNSs. Figure 3.12 forms one logical SNS using the front-back crossbars: F1, D1, I1, E1.

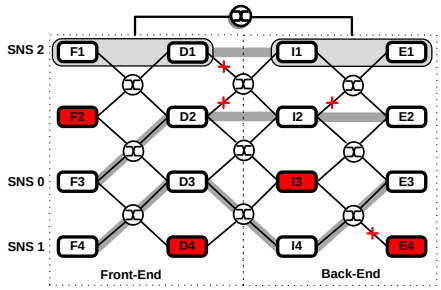



**Figure 3.12: Configuration of SW with overlapping and front-back crossbars. The front-back crossbars adds one more logical SNS (SNS 2) over the configuration result of overlapping crossbars.**

In summary, front-back crossbar configuration, along with overlapping crossbar as the first-level interconnection, is able to form *three* logical SNSs. The configuration algorithms discussed in this section can cover all possible interweaving candidates discussed in Section 3.3.2. It is noteworthy that the algorithms presented here are not optimal (in specific, the latter two), and are based on heuristics. This was done in order to keep their run-times linear and, thus, minimize the overhead of in-field reconfigurations.

### 3.3.4 Interconnection Reliability

Interconnection reliability can be divided into link reliability and crossbar reliability. The link reliability is accounted for, to a certain extent, by the interconnection alternatives which introduce redundancy. Further, they are not as vulnerable to wearout and variation as logic. For crossbar reliability, SW can use three alternatives:

1. *Simple Crossbar:* This is the simplest scenario with a single crossbar switch used at each interconnection spot. No redundancy is maintained in this case.
2. *Simple Crossbar with spare(s):* In this set-up, one spare is maintained for every crossbar in the system. This doubles the area required by the crossbar switches, but significantly improves the interconnection reliability. The cold spare corresponding

to a crossbar switch is only brought into use when the latter develops a certain number of port failures.

3. *Fault-Tolerant Crossbar (no spares):* The third and final option is to deploy one-sided fault-tolerant (FT) crossbars [115] that nearly eliminate the chances of crossbar failures. Note that in a FT crossbar, multiple paths exist from a given input port to the output port. This is unlike a regular crossbar that have a unique path for every input-output pair. The biggest downside of these crossbars is that they tend to have a 2-3X area overhead compared to regular crossbars.

### 3.3.5 Variation Tolerance

Process variation introduces slower circuit components throughout a chip. This presence of slower components results in a wide distribution of operational frequencies for different structures on the die. For instance, in a conventional CMP, the slowest structure within each core would determine the best frequency achievable by that core. Similarly, in the case of SW, this impact can be observed at the granularity of pipeline stages, a few of which will be much slower than others. However, unlike a conventional CMP, SW can selectively salvage faster pipeline stages from the grid of resources and construct logical pipelines that can operate at a higher frequency. This will result in an improved distribution of core frequencies as compared to a traditional CMP with isolated cores.

The configuration methodology of SW in the presence of process variation builds upon the algorithms discussed earlier. The key observation is that for a given frequency target (and fixed supply voltage), pipeline stages can be marked functional or non-functional. Once this level of abstraction is reached, the non-functional stages can treated in the same

manner as broken stages were earlier in this section. Given a SW chip with a wide variation in pipeline stage frequencies, the algorithm proceeds as follows. It start with the highest possible frequency, and marks the working stages in the grid. Standard configuration algorithm is used to form logical pipelines. The frequency is now reduced by a unit step, and the process is repeated. This is continued until the configuration is defined for the lowest operational frequency of the system. At this point, the number of cores functional at each frequency point can be tabulated.

Apart from enhancing the performance, the improvement in core frequencies using SW can also be translated into energy savings relative to a conventional CMP. The insight here is that given a system utilization level (fraction of cores occupied) of less than one, SW can form the fastest cores from its pool of stages and meet the frequency target at a lower operational voltage than a CMP. Since the CMP lacks the flexibility to combine faster stages across its cores, it will be forced to run at a higher voltage to meet the same frequency target. This difference in voltage translates to (quadratic) dynamic power savings and (cubic) static power savings [34]. As both systems operate at the same frequency, these power savings map directly to energy savings.

### 3.3.6 System Level Issues

**Wearout detection:** In-field detection of wearout failures is crucial in order to maintain fault-free working of a SW chip. As in the case of SN (see section 4.3.2.1), SW can also employ a continuous [16, 56] or periodic [28, 41] fault detection mechanism. For the sake of our evaluations in this chapter, we assume the presence of a continuous fault monitoring system.

**Manufacture time testing:** The presence of process variation makes every SW chip unique by introducing a fair bit of non-determinism across the die. SW's variation tolerance technique requires measurement of this non-determinism in order to mitigate it. In specific, a standard test flow is needed to determine the frequency of every pipeline stage on the chip. The present day manufacturing tests are already equipped to measure processor core frequencies, and can be augmented to provide frequency data one level deeper. Further, it is likely that testing methodologies will also adapt in future designs as process variability increases [66].

**Configuration manager:** SW requires a software level configuration manager for supervising the system-wide reliability and performance configuration. The inputs to this manager are a list of working resources, the best working frequency for each resource and the system utilization. Upon receiving this input, the configuration manager assembles the desired number of pipelines with the fastest available stages. The application of fastest stages saves the maximum amount of energy (as voltage can be scaled down). The configuration manager is re-invoked every time a failure occurs or the workload set changes. As the run-time of configuration algorithms is linear, its impact on system performance is very limited.

## 3.4 Evaluation

### 3.4.1 Methodology

The evaluation methodology for SW encompasses four different components: 1) microarchitectural simulator for pipeline performance, 2) wearout and process variation mod-

eling, 3) overhead computation, and finally, 4) CMP throughput and lifetime Monte-Carlo simulations.

#### 3.4.1.1 Microarchitectural Simulation

The microarchitectural simulator for the SW evaluation was developed using the Liberty Simulation Environment (LSE) [113]. Two flavors of the microarchitectural simulator were implemented in sufficient detail to provide cycle accurate results for single thread performance. The first simulator models a five stage pipeline, which is used as the baseline. The second simulator models the decoupled SNS pipeline microarchitecture with all its enhancements (see Section 4.3.2.1). Table 5.5 lists the parameters for the core and the memory hierarchy used for the simulations. These parameters and the baseline microarchitecture pipeline stages are modeled after the OR1200 processor [76], an open source RISC microprocessor.

**Table 3.1: Architectural parameters.**

| | |
|---|---|
| Pipeline | 4-stage in-order OR1200 RISC [76] |
| Frequency | 400 MHz |
| Area | $1mm^2$ |
| Branch predictor | Global, 16-bit history, gshare predictor<br>BTB size - 2KB |
| L1 I$, D$ | 4-way, 16 KB, 1 cycle hit latency |
| L2 $ | 8-way, 64 KB (per core), 5 cycle hit latency |
| Memory | 40 cycle hit latency |

#### 3.4.1.2 Wearout and Process Variation Modeling

The evaluation of SW involves both lifetime wearout experiments and process variation modeling. For the wearout failures, the mean-time-to-failure (MTTF) was calculated for

the various stages and crossbars in the system using the empirical models found in [103]. The entire core was qualified to have a MTTF of 10 years. This is a conservative estimate for future technologies as this value is expected to get much lower. These wearout models heavily depend on the module (stages and crossbar) temperatures that were generated using HotSpot [47]. A customized floorplan was created for StageWeb to account for the lateral heat transfer on the die. Finally, the calculated MTTFs are used as the mean of the Weibull distributions for generating times to failure (TTF) for each module (stage/crossbar) in the system. The stages are considered dead as a whole when a fault occurs, whereas, the crossbar failures are modeled at the crossbar-port granularity.

Process variation was modeled using VARIUS [91]. Given a chip's floorplan, and $\sigma/\mu$ for a technology process, VARIUS can be used to obtain the spread of operational frequencies for all structures on the die. In our experiments, we use $\sigma/\mu$ of $0.25$, as a representative value for technologies beyond $32nm$.

#### 3.4.1.3 Area, Power and Timing

Industry standard CAD tools with a library characterized for a 90nm process[1] are used for estimating the area, power and timing for all design blocks. A Verilog description for the OR1200 microprocessor was obtained from [76]. All other design blocks, SNS enhancements, and crossbar configurations were hand-coded in Verilog. The procedure adopted for each of the ELSI overheads is summarized below:

1. **Area:** All blocks were synthesized using Synopsys Design Compiler. Placement and routing was conducted using Cadence First Encounter. The area for the interconnec-

[1]The use of an older process technology is a limitation of our academic research setup. However, it is not catastrophic as all overhead comparisons are relative.

tion links between stages and crossbars was estimated using the same methodology as in [64] with intermediate wiring-pitch at 90nm taken from the ITRS road map [53].

2. **Power:** The power consumption for all structures was computed using Synopsys Power Compiler. For the power saving experiments, we assume that dynamic power scales quadratically with supply voltage, and linearly with frequency [86].
3. **Timing:** The synthesis tool chain (used for area) was also employed to find the target frequency for the design. The interconnection link delay between stages and crossbars was estimated using the intermediate wiring-delay from the ITRS road map [53].

#### 3.4.1.4 CMP Simulations

A thorough simulation infrastructure was developed to simulate a variable-size regular CMP system and SW system. This infrastructure integrates all components of our evaluation methodology and SW design: single thread performance, wearout modeling, interweaving alternatives, configuration algorithms and crossbar models. To obtain statistically significant results, 1000 Monte-Carlo runs were conducted for every lifetime reliability experiment.

For lifetime reliability experiments, the stages/crossbars fail as they reach their respective time-to-failures (TTFs). The system gets reconfigured over its lifetime whenever a failure is introduced. The instantaneous throughput of the system is computed for each new configuration using the number of logical SNSs. This way, we can obtain the chip's throughput over its lifetime.

### 3.4.2 StageWeb Design Space

For the latest generation Intel Core 2 processors, about $60\%$ die area is occupied by the processing cores. With that estimate, in order to accommodate 64 OR1200 RISC cores (our baseline in-order core) we assume a $100mm^2$ die (a typical size for current multicore parts). We use this die area as the basis for constructing various SW chip configurations. There are a total of twelve SW configurations that we evaluate, distinguished by their choice of interweaving candidates (single, single with front-back, overlap, overlap with front-back) and the crossbars (no spare, with spare, fault-tolerant). Table 3.2 shows the twelve configurations that form the SW design space. The cap on the processing area guarantees an area-neutral comparison in our results. In the base CMP case, the entire processing area can be devoted to the cores, giving it a full 64 cores. However, depending upon the interconnection complexity, SW configurations can have a varying number of cores.

The interconnection (crossbar + link) delay acts as a limiting factor while connecting a single crossbar to a group of slices. As per our timing analysis, the maximum number of slices that can be connected using a single crossbar is 6. This is for the 90nm technology node and a single-cycle crossbar. A two-cycle crossbar (that is used as the Front-Back crossbar) can connect up to 12 slices together. The overlapping crossbar also uses a single-

**Table 3.2: Design space for SW. The rows span the different interconnection types (F/B denotes front-back), and the columns span the crossbar type: crossbar w/o (without) sp (spares), crossbar w/ sp and fault-tolerant (FT) crossbar. Each cell in the table mentions the number of pipeline slices, in each SW configuration, given the overall chip area budget ($100mm^2$).**

| **Interweaving** | Xbar (w/o sp) | Xbar (w/ sp) | FT Xbar |
|---|---|---|---|
| Single Xbar | 56 | 55 | 54 |
| Single + F/B Xbar | 55 | 53 | 52 |
| Overlap Xbar | 55 | 53 | 52 |
| Overlap + F/B Xbar | 54 | 51 | 50 |

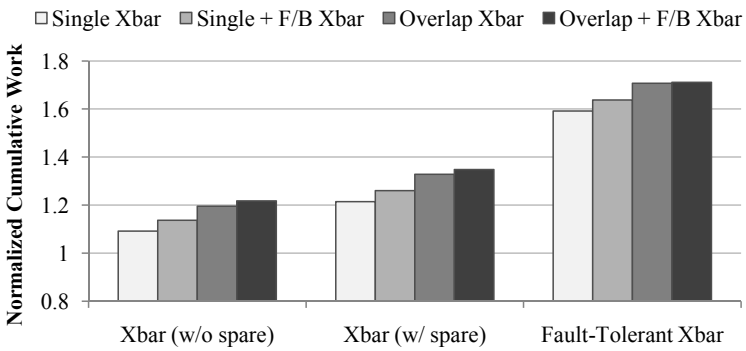



**Figure 3.13: Cumulative work performed by the twelve SW configuration normalized to a CMP system. The cumulative work improves with the richer choices for interweaving, as well as with the more resilient crossbars. In the best case, a SW system can achieve 70% more cumulative work relative to the CMP system.**

cycle crossbar, so it can give an illusion of connecting $\frac{3}{2}n$ slices, which is 9 in this case.

### 3.4.3 Cumulative Work

The lifetime reliability experiments, as discussed in the evaluation methodology, track the system throughput over its lifetime. The cumulative work, used in this section, is defined as the total work a system can accomplish during its entire lifetime, while operating at its peak throughput. In simpler terms, one can think of this as the total number of instructions committed by a CMP during its lifetime. This metric is same as the one used in [43]. All results shown in this section are for 1000 iteration Monte-Carlo simulations.

Figure 3.13 shows the normalized cumulative work results for all twelve SW configurations. The cumulative work for all configurations is normalized to what is achievable using a 64 core traditional CMP. The results categorically improve with increasing interweaving richness, and better crossbar reliability. The biggest gains are achieved when transitioning from the regular crossbar to the fault-tolerant crossbar. This is due to the ability of the fault-tolerant crossbar to effectively use its internal fine-grained cross-point redundancy [115],

while maintaining fault-free performance.

Between the four interweaving candidates, the richer interconnection options perform consistently better. This is independent from the choice made for the crossbars. The overlapping crossbar configuration tends to do almost as well as the overlapping with front-back crossbars. When using the fault-tolerant crossbars, SW system can deliver up to 70% more cumulative work over a regular CMP.

The same set of experiments (as above) were repeated in an area-neutral fashion for the twelve SW configurations (using the data from Table 3.2). Figure 3.14 shows the cumulative work results for the same. The trend of improving benefits while transitioning to a more reliable crossbar remains true here as well. However, the choice of the best interweaving candidate is not as obvious as before. Since the area of each interconnection alternative is factored-in, the choice to use a richer interconnect has to be made at the cost of losing computational resources (pipelines). For instance, the (fault-tolerant) overlapping crossbar configuration (column 11) fares better than the (fault-tolerant) overlapping with front-back crossbar configuration (column 12). The best result in this plot (fault-tolerant overlapping crossbar) achieves 40% more cumulative work than the baseline CMP.

### 3.4.4 Throughput Behavior

The cumulative work done by the system is a useful metric, but is insufficient in showing the quality of system's behavior during its lifetime. For this purpose, we conducted an experiment to track the system throughput over its lifetime (Figure 3.15), as wearout failures occur. Three systems configurations are compared head-to-head: SW's best configuration *fault-tolerant overlapping crossbars*, area-neutral version of *fault-tolerant overlapping*

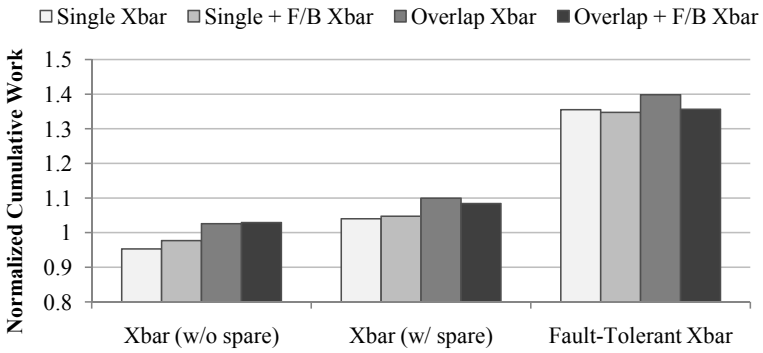



**Figure 3.14: Cumulative work performed by the twelve SW configuration normalized to a CMP system (*area-neutral study*). The cumulative work improves with more resilient crossbar choice. However, richer interweaving does not map directly to better results. For instance, front-back crossbars add a lot of area overhead without delivering proportional amount of reliability. In the best case, a SW system achieves 40% more cumulative work relative to the CMP system.**

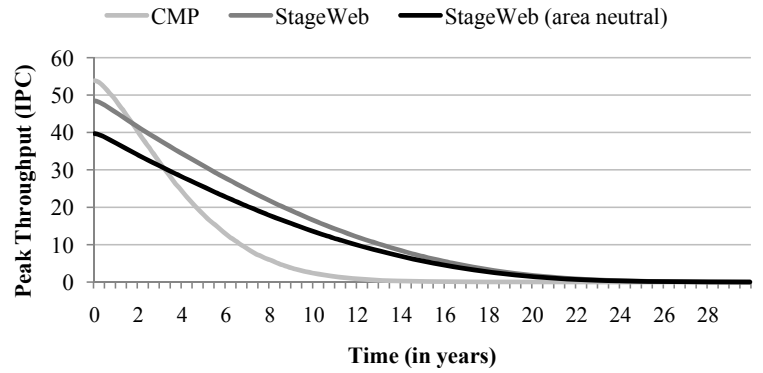



**Figure 3.15: This chart shows the throughput over the lifetime for the best SW configurations and the baseline CMP. The throughput for the SW system degrades much more gradually than an equally provisioned CMP system. In the best case (around the 8 year mark), SW delivers 4X throughput of the CMP.**

*crossbars*, and the baseline CMP. As evident from Figure 3.15, the throughput for the SW system exhibits a very graceful degradation with the progression of time. At the beginning of life, the CMP system has an edge over the SW system. This is due to the higher number of pipeline resources a CMP system initially possesses. However, the SW catches up soon enough into the lifetime, and maintains its advantage for the remaining lifetime. The lifetime range, shown here as 24 years, is expected to shrink in future technology generations, making the case for SW-like systems even stronger.

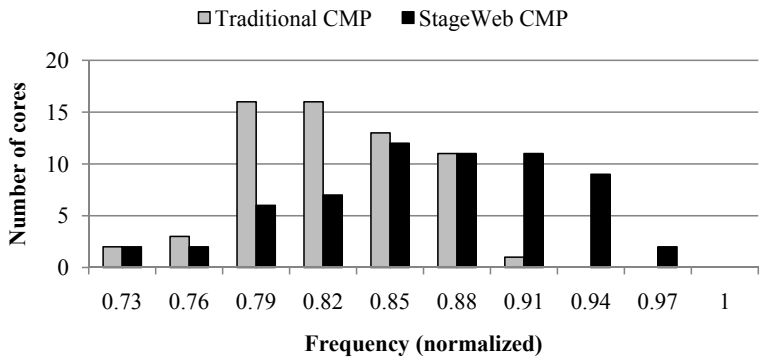



**Figure 3.16: The distribution of core frequencies in 64-core CMP and StageWeb chips. Facing the same level of process variation, SW enables a noticeable improvement in the frequency distribution.**

### 3.4.5 Variation Mitigation

In addition to wearout tolerance, the interconnection flexibility of SW can also be leveraged to mitigate process variation. As discussed in Section 3.4.5, the basic idea is to group together faster pipeline stages to form pipelines that can run at higher frequencies. This way, the slower resources are isolated, reducing their overall performance impact. Figure 3.16 shows the distribution of core frequencies for a regular CMP system and a SW CMP with overlapping configuration. In this experiment, both systems contain 64 cores each, and process variation is injected with $\sigma/\mu = 0.25$. The results confirm that the distribution of core frequencies in a SW CMP are considerably better than that of a conventional CMP. The mean increase in the core frequencies is $7\%$. It is noteworthy that the slowest cores in both systems operate at the same frequency ($0.73$). This is true by construction, since even in a SW CMP, some logical pipeline has to absorb the slowest stage and operate at that frequency.

### 3.4.6 Power Saving

The better distribution of frequencies, as discussed in Section 3.4.5, can also translate into power/energy savings. For a given system utilization, SW can scale down the supply voltage (reducing power quadratically) and still provide the same level of performance as a baseline CMP. Note that a single global supply voltage is assumed in all our experiments. This is a commonly accepted practice as multiple supply sources introduce significant noise. Figure 3.17 shows the power savings obtained at different levels of system utilization (fraction of cores occupied) when using SW. Each bar is normalized to the CMP power at that utilization level. The results range from 16% power saving at 12.5% utilization to a small loss in power at 100% utilization. When the utilization is low, more opportunity exists for SW to gather faster stages, and switch off the slowest ones. But, at full utilization, everything (including the slowest stage) has to be switched on, requiring the global supply voltage to be scaled back to its original level. Most commercial servers have time-varying utilization [6] (segments of high and low utilization), and can be expected to create many opportunities where SW saves power. Since this power is saved without any accompanying loss in performance (frequency), it translates directly to energy savings.

### 3.4.7 Yield Analysis

Manufacturing yield is a indispensable metric when evaluating a defect-tolerant system. A principal objective of the SW architecture was to develop a system that can meet the challenging goal of high defect density scenarios. To verify this claim, we evaluated the yield for all twelve SW configurations. This study was also kept area-neutral. One

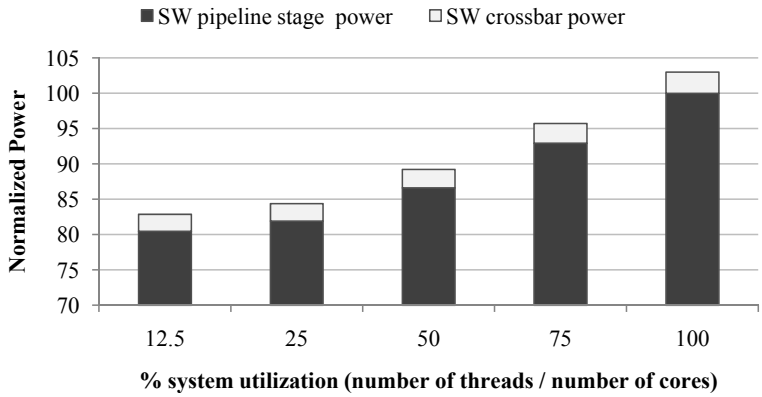



**Figure 3.17: Power saving using SW relative to a CMP at different system utilization levels. This saving is made possible due to SW's ability to deliver same performance as a CMP at a lower voltage, in the presence of process variation. The plot also shows the break up between pipeline stage power and crossbar power.**

thousand chips were generated, and chips with at least 100 working cores were rated as good. Figure 3.18 shows the results for three different defect densities. For defect density values lower than 0.2 defects/$mm^2$, all configuration yielded 100%. It is noteworthy that unlike lifetime wearout experiments, the crossbar type does not have a major influence on the results. Fault-tolerant crossbar area is significantly larger than the regular crossbars, increasing the number of defects that will manifest in them (since more area translates to more defects). This negates the benefits it can provide over the regular crossbars that are much smaller and face fewer defects. In contrast, wearout failures occur based on the usage of structures, and not their area. Thus, the fault-tolerant crossbar fares better in lifetime experiments. As far as the interweaving alternatives are concerned, a richer interconnection goes noticeably farther in delivering better yields, with overlapping and front-back crossbar together giving the best result.

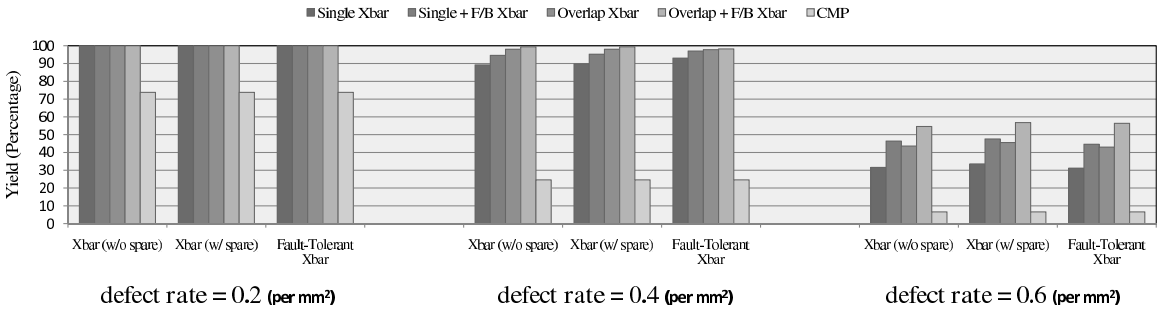



**Figure 3.18: Yield obtained for all the twelve SW configurations and the CMP at three defect densities. The advantage of the SW becomes more prominent as the defect density rises.**

## 3.5 Related Work

StageWeb, the architecture proposed in this chapter, leverages the concept of stage-level reconfiguration introduced by StageNet (Chapter II), and extends it to build many-core systems resilient to wearout failures as well as process variation. To the best of our knowledge, SW is the first work to study concepts such as interconnection scalability and crossbar reliability at the sub-core granularity.

The prior research efforts on tolerating process variation have mostly relied on using fine-grained VLSI techniques such as adaptive body biasing / adaptive supply voltage [110], voltage interpolation [67], and frequency tuning. Although effective, all such solutions can have high overheads, and their feasibility has not been established in mass productions. SW stays clear of any such dependence on circuit techniques, and mitigates process variation with a fixed global supply voltage and frequency.

Please refer to Section 2.6 for a summary of past work in reliable architecture design.

## 3.6 Summary

With the looming reliability challenge in the future technology generations, mitigating process variation and tolerating in-field silicon defects will become necessities in future computing systems. In this chapter we proposed a scalable alternative to the tiled CMP design, named StageWeb (SW). SW fades out the inter-core boundaries and applies a scalable interconnection between all the pipeline stages of the CMP. This allows it to salvage healthy stages from different parts of the chip to create working pipelines. In our proposal, the flexibility of SW is further enhanced by exploring a range of interconnection alternatives and the corresponding configuration algorithms. In addition to tolerating failures, the flexibility of SW is also used to create more power-efficient pipelines, by assembling faster stages and scaling down the supply voltage. The best interconnection configuration for the SW architecture was shown to achieve 70% more cumulative work over a regular CMP containing equal number of cores. Even in an area-neutral study, SW system delivered 40% more cumulative work than a regular CMP. And lastly, in low system utilization phases, its variation mitigation capabilities enable SW to achieve up to 16% energy savings.

In summary, SW provides the basis for constructing dependable and efficient CMPs by adding new dimensions of adaptability and configurability.

# CHAPTER IV

# Adaptive Online Testing for Efficient Hard Fault Detection

## 4.1 Introduction

The challenge of tolerating such permanent hardware faults (i.e., silicon defects) encountered in-field can be divided into three tasks 1) defect detection and diagnosis, 2) recovery to a correct system state after a failure and 3) reconfiguration/repair mechanism to prepare the system for future computation. The focus of this chapter is on improving the efficiency of the first task: defect detection and diagnosis. Recovery techniques (second task) typically employ a checkpointing mechanism to rollback the system after a failure. These checkpoints are created periodically so that in the event of a failure, not much useful work is lost. SafetyNet [99] and ReVive [84] are two good examples of CMP checkpointing solutions. Finally, the solutions for the repair (third task) typically leverage hardware redundancy to replace broken component(s) or in some cases, merely isolate them. The Replacement/isolation techniques exist for a range of granularities: cores [2], pipeline stages [43] (more discussion in Chapters II and III) and modules within a processor [95].

Defect detection and diagnosis mechanisms can be broadly divided into two broad categories: 1) *continuous*: those that constantly monitor the logic blocks for errors and 2) *periodic*: those that periodically check the processor's logic. A few examples of the *continuous* detection mechanisms are dual modular redundancy (DMR) and DIVA [10]. The common idea between all these solutions is to have some sort of redundant computation (in time or in space) to validate the execution. However, all of them impose significant overheads for area, latency, power and energy. Another means for continuous detection is through sensors that can estimate the amount of device level wearout. Although a variety of low level sensors have been proposed [1, 56, 16], they are limited in their capability to accurately predict/detect a failure.

In contrast, *periodic* detection does not require redundant execution and can give sound guarantees on the fault coverage. These techniques periodically test the system for defects and in case of a failure, they rely on checkpointing and recovery mechanisms. Figure 4.1 shows snapshot of a system where the tests are conducted at the end of every checkpoint interval. Some of the recent proposals of periodic detection mechanisms are ACE analysis [28] and VAST [48]. Unfortunately, in these proposals, the periodic testing time constitutes as much as 5%-30% of the total system time [28]. This sort of overhead is unacceptable for a high end server that typically apply (virtual machine) consolidation to maintain 100% utilization levels. Even in the case of embedded systems, a great deal of time and energy can be saved by reducing the overhead of periodic testing.

In this work, we propose an adaptive testing framework (ATF) that significantly reduces the overhead of periodic testing in a CMP system. The key insight in ATF is to adapt the testing process to the state of the underlying hardware. For instance, a healthy processor

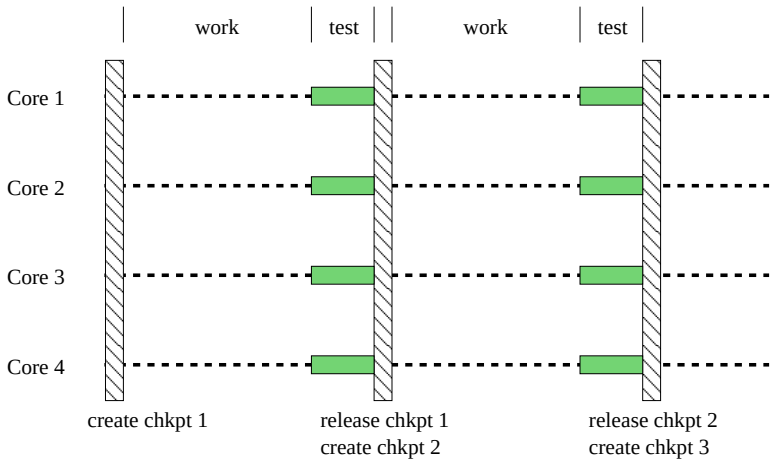



**Figure 4.1: Periodic testing for fault detection. The vertical stripes represent the checkpoint start/release and the horizontal lines show the progression of threads. At the end of every checkpoint interval, testing is conducted for all processing cores, this is shown as solid horizontal bars.**

within a CMP can be lightly tested, whereas a weaker counterpart needs thorough testing. In specific, this adaptivity is applicable in three different scenarios:

1. The health of a system varies over its lifetime due to device wearout. Thus, all processors are relatively healthy in the beginning and then deteriorate over time.
2. Manufacture time process variation can form components with differing health levels.
3. Different amounts of stress are experienced by the processors depending up on the workloads assigned.

In all the aforementioned cases, the proposed ATF can deliver significant savings on the periodic testing effort while providing the same level of fault coverage. Essentially, our system assesses the health of different processors in a CMP, and appropriately conducts tests. To enable the assessment of processor health, we employ a population of low level sensors [56, 57]. These sensors can predict the mean time to failure (MTTF) with about 25% error for less than a 3% area overhead. We further extend the ATF for application to the

StageNet (SN) CMP fabric [43], a highly flexible computing substrate. SN allows arbitrary grouping of stage-level resources from different pipelines to form logical pipelines. We exploit this feature of SN to group together weaker resources from different pipelines and conduct concurrent testing.

The main contributions of this work can be summarized as follows:

1. The proposed ATF introduces the use of low level sensors to guide the online testing process.
2. The ATF achieves a significant reduction in the overhead of periodic testing by adaptively matching the testing process to the underlying hardware's health.
3. An extension of the ATF to StageNet, a flexible CMP fabric, for achieving larger benefits.
4. Lifetime reliability experiments to measure the fraction of time devoted to periodic testing. This setup models process variation, sensor error, device wearout, and testing overhead.

## 4.2 Background

Here we provide a brief overview of the latest techniques for assessing system health and conducting online tests. Both of these form integral part of the adaptive testing framework proposed later in Section 4.3.

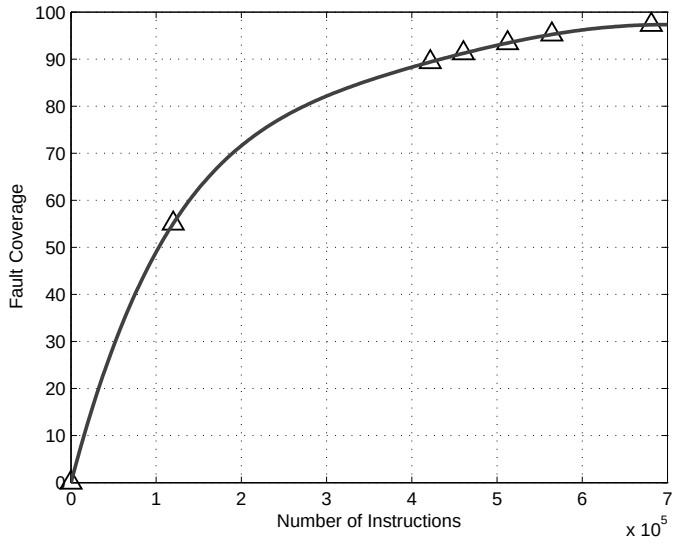



**Figure 4.2: Fault coverage achieved (in percentage) for varying number of software based self test instructions.**

### 4.2.1 Wearout Sensors

Wearout monitoring for on-chip devices is a challenging problem and has been an active area of research. Circuit-level designs have been proposed for in-situ sensors that detect the progress of various wearout mechanisms with a reasonable accuracy [56, 72]. A trade-off exists between their accuracy and the area overhead from using them. These sensors are usually designed with area efficiency as a primary design criteria, allowing a large number of them to be deployed throughout the chip for monitoring overall system health. A different approach to sensor design has been to examine the health of on-chip resources at a coarser granularity. Research has involved simple temperature sensors, two dozen on the POWER6 [39], to more complex designs such as the wearout detection unit [16]. These sensors can effectively approximate the useful life remaining in a microarchitectural module.

### 4.2.2 Online Testing

The goal of online testing is to detect fault effects, or errors, while the system is in-field. A number of test methodologies exist for online testing, the three important categories being: 1) built-in self test (BIST) based, 2) functional test, and 3) software based self-test (SBST). While BIST addresses the testing problem comprehensively by providing a high fault coverage, it introduces significant hardware overheads [32]. For low-cost embedded systems, such an overhead can not be justified. On the other hand, functional tests use a software program to conduct the testing. The challenge there is the generation of high fault coverage program instructions and automating the process for the same. Most functional testing solutions achieve low fault coverage because they do not consider the RTL structure and are not based on a gate-level fault model (like s-a-fault) [13].

SBST links the instruction-level tests with low-level fault models to achieve good fault coverage while introducing no hardware overhead. SBST starts off by generating module specific deterministic tests patterns and then uses processor instructions as a vehicle for delivering the patterns to module inputs and collecting their responses. The processor simply executes the test program at-speed from the on-chip memory. The test program length is chiefly determined by the module/structure that needs the maximum number of tests. The advantages of SBST are its low cost, ease of application and extensibility. A variety of proposals have been made for SBST [22, 78, 69] with a considerable success. The latest being [69] that reports up to 97.3% fault coverage. The test generation algorithms (for SBST and functional testing) can comfortably trade-off the test size with the amount of fault coverage. Figure 4.2 illustrates this trade-off between the amount of fault coverage

and the number of software test instructions executed for a ARM9-v4 compatible RISC processor using data from [69]. As seen in the figure, the last few percentages of the coverage require the maximum testing effort (number of test instructions).

## 4.3 Adaptive Online Testing

Periodic test based fault detection approaches suffer from the constant overhead of the full test application for all available processing components. In view of the increasing process variation, and the differing amounts of component wearout over the lifetime, an effective optimization is to match the testing thoroughness with the health of a component. We propose an adaptive online testing methodology that builds upon this key insight. Our technique leverages low level sensors to assess the probability of failure in various system components, and appropriately decides the quality of tests applied. The primary benefits from this strategy are the savings in the test time and energy. In addition to the traditional CMP, we extend this adaptive testing philosophy to StageNet(SN) [43], a highly flexible CMP fabric. The advantages of the proposed technique are further magnified while using the SN architecture. The rest of this section provides the details of the adaptive testing framework and discusses its application to a traditional CMP and the SN architecture.

### 4.3.1 Adaptive Test Framework

A conceptual illustration of the adaptive test framework (ATF) is shown in the Figure 4.3. The baseline CMP system is enhanced with the capabilities to assess component health, apply suitable tests, recover from faults (if any) using a checkpointing mecha-

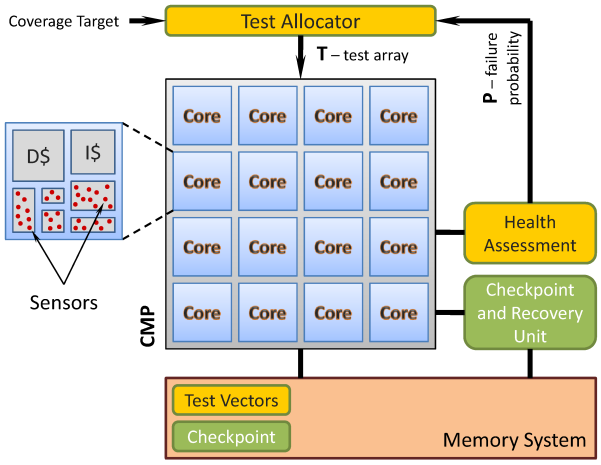
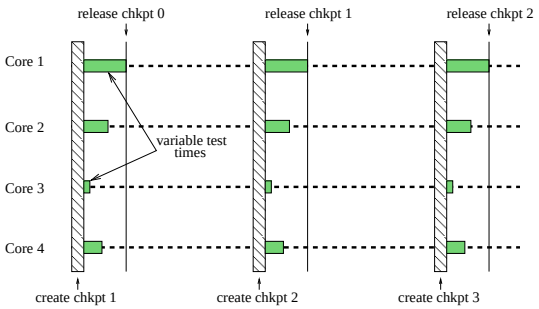



**Figure 4.3: Adaptive testing framework. A generic CMP system is shown along with the enhancements needed to enable adaptive testing. Health assessment is responsible for gathering sensor readings and producing a fault probability array (P). This array is taken up by the test allocator, along with the target coverage, to generate appropriate tests (T) for different processing cores.**

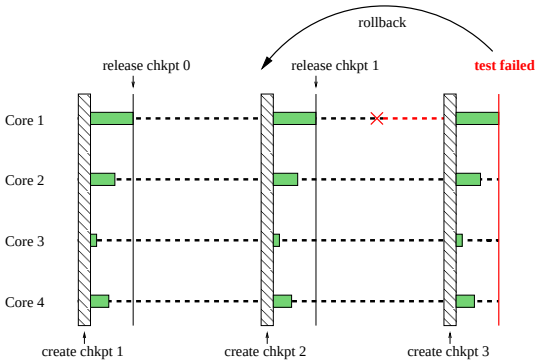
nism [84], and, finally, isolate the faulty core (if and when found). At the end of every periodic checkpoint interval, a health assessment is conducted for all the components in the system, and the corresponding probabilities of failure (**P**) are determined. This array is in turn used by the test allocator to generate suitable tests (**T**) for all components. In the early lifetime, when most of the components are healthy (have low probability of failure), a fewer number of tests are required to make sure the system operates correctly. As the components grow older, their failure probabilities are expected to rise, resulting in a need for more thorough tests. Later in this section, we use this intuitive argument to derive a fault coverage metric ($C$), that measures the probability of the system to be in a *safe state*. Given a system-wide fault coverage target $C$, the ATF decides the optimal number of tests required at a per component level (core in this case). This way, testing effort is reduced for the healthy components in the system. The functioning of the important blocks in the Figure 4.3 is detailed below:

**Health Assessment:** The lack of knowledge of the underlying component health is the primary reason for applying full tests throughout the component's lifetime. We alleviate this problem by deploying low level sensors that can measure degradation at the transistor level. The primary requirement for such a sensor is to accurately measure the device level characteristics taking into account the process variation and the wearout accumulated over its lifetime. Furthermore, a single sensor won't be enough to provide statistically significant results for an entire core health, and therefore 10-100s would need to be deployed. In such a scenario, low area overhead becomes a favorable feature for such sensors. In this work, for the purpose of illustration, we use the oxide breakdown sensors proposed by E. Karl et al. [56, 57]. These are close to ideal sensors in behavior and have an extremely small area footprint. The results in [57] demonstrate that 500 such sensors are enough to estimate the MTTF (mean time to failure) for an entire chip with less than $10\%$ error. Note that the proposed methodology is not tied to any one sensor type, and a variety of other sensor designs [1, 16] are equally applicable to the methodology proposed here. The ongoing research on NBTI sensors and IDDQ based wearout sensors can also be easily integrated within the ATF. Nevertheless, our choice here was directed by the available data on the accuracy of the oxide breakdown sensor [57].

All cores in our system are enhanced with these sensors. The data from these sensors is gathered and processed in software to generate the MTTF [57, 55] with an error based on the number of sensors. Using the current mean sensor reading, the projected MTTF, and the error in MTTF, we calculate the probability of failure for a core. Note that a higher error in the MTTF estimation translates into a more conservative value of the probability of failure. The discussion of this derivation has been left out in the interest of space. This




(a) No fault scenario




(b) Fault occurrence, test failure and rollback scenario

**Figure 4.4: Checkpointing and adaptive testing for efficient fault detection. Notice that 1) the tests are applied after a new checkpoint is started, and 2) old checkpoint is released once the tests finish successfully.**

process is repeated for all the cores in the system to generate the probability of failure array (**P**).

**Test Allocator:** The task of the test allocator is to prepare suitable test programs for all the cores in the system. At every checkpoint interval, the test allocator is provided with two inputs: 1) a coverage target $C$ (ranges from 0 to 1), and 2) a probability of failure array **P**. Using these two values, it determines the test *fault coverage* ($FC$) needed by each individual core, such that the coverage target $C$ is always met for that core. Here, the term *fault coverage* implies the fraction of hardware faults covered by test patterns.

For a given core $i$, and a checkpoint interval $t$, if the:

$$\begin{aligned} probability\ of\ failure &= P_i(t),\ and \\ fault\ coverage &= FC_i(t),\ then \\ 1 - C &= P_i(t)[1 - FC_i(t)] \end{aligned}$$

In other words, the probability of the periodic test not catching a fault $1 - C$ in core $i$ is the product of fault occurring $P_i(t)$ and not getting covered $1 - FC_i(t)$. From this, we can solve for the required test fault coverage:

$$\begin{aligned} FC_i(t) &= 1 - \frac{1 - C}{P_i(t)},\ placing\ bounds\ on\ coverage: \\ FC_i(t) &= Min\left\{best\_coverage, Max\left\{0, 1 - \frac{1 - C}{P_i(t)}\right\}\right\} \end{aligned}$$

Thus, given a coverage target $C$, a higher probability of failure $P_i(t)$ necessitates an increase in the fault coverage and vice versa. The final equation above also adds bounds to the possible values of the fault coverage, $0$ being the minimum and $best\_coverage$ being the best possible coverage using the test generation technique employed. In this work, we propose the use of software based self test (SBST) to conduct the online testing [69]. The advantage of the software based testing is two fold: 1) no hardware overhead, and 2) the fault coverage level is flexible. The proposed methodology in [69] allows generation of test programs to meet different levels of fault coverage. The number of software test instructions are thus tuned on a per core basis to match the fault coverage desired for the

same. Figure 4.2 shows the (single stuck at) fault coverage achievable for an ARM9-v4 compatible RISC core for a range of number of software test instructions. A full set of test instructions is stored in the main memory. The test allocator uses this set of instructions to prepare an array of test programs (**T**) for all the cores. As in the case of sensors, our proposed methodology is not tied to any specific online testing technique.

**Checkpoint and Recovery:** In the event of a failure, a recovery system is needed to get the system back into an operational state and isolate the broken component(s). This can be achieved by deploying a CMP checkpoint solution. In this work, we use the ReVive checkpoint system [84]. Revive has a very minimal hardware overhead and maintains the checkpoint in the main memory. The checkpoint interval length can be tuned based on the availability of storage in the targeted system.

Figure 4.4 shows two scenarios of the ATF in action. The first scenario, illustrated in the Figure 4.4(a), is for a case with no failures. The horizontal lines show the progression of thread execution, interspersed by the regular checkpoint creations (shaded vertical stripes). The testing phases are shown by solid horizontal bars following each checkpoint creation. The test times vary from core to core, depicting the adaptive nature of the online tests. In this example core 1 runs the longest test (worst health), and core 3 the shortest (best health). The previous checkpoint is released once the tests for all the cores complete successfully. Notice that unlike the traditional practice of testing and then forming a checkpoint (Figure 4.1), we do the reverse. This design choice is a result of variable test times of the cores in our system. In order to run variable lengths tests on all the cores before a checkpoint, they have to be started at different times. This adds to the complexity of health assessment, test allocation and test scheduling. Thus, in ATF, all tests start concurrently af-

ter a new checkpoint is created. Over time, as the cores finish their tests, they are released by the ATF and are made available for job scheduling. However, by creating an additional checkpoint just before running the tests, ATF necessitates two outstanding checkpoints to co-exist while the tests run on the cores. Fortunately, most checkpoint systems, including ReVive, maintain checkpoints as a log of system-wide updates. As the testing phase is very short in length, the additional updates saved in the log due to the second checkpoint are very few, leading to a negligible memory burden. The second scenario, illustrated in the Figure 4.4(b), shows a case with a failure in core 1 while running a job. The failure is detected during the tests following the creation of the third checkpoint, and system is rolled back to an operational state using the second checkpoint.

**System Coverage (SC) Metric:** For a system that is periodically tested for faults, there are three distinct categories of events:

1. No failure occurs in the last completed interval
2. Failure occurs and is detected by the test program
3. Failure occurs and is *not* covered by the test program

The first two events maintain the system in the *safe state*, and represent the scenarios where no fault escapes the test. However, the third event is an unwelcome scenario where a fault occurs without being caught. Let us say we have a multi-core chip with $n$ cores. As discussed above, probability of a core $i$ missing a fault in a checkpoint interval $t$ is $P_i(t)[1 - FC_i(t)]$. In other words, the fault occurs and the test is not able to expose it. Continuing along the same lines, the average probability of missing a fault in the entire $n$

core system, within a given checkpoint interval $t$:

$$Probability\ of\ missing\ fault \quad = \quad \frac{1}{n}\sum_{i=1}^{n} P_i(t)[1 - FC_i(t)]$$

If we sum this over the entire system lifetime, the average probability for the system to miss a fault can be written as:

$$\frac{1}{nT}\sum_{t=1}^{T}\sum_{i=1}^{n} P_i(t)[1 - FC_i(t)]$$

Therefore, the average probability (over the lifetime) of the system *not* missing any faults, i.e. the probability of system being in a *safe state* is:

$$SC = 1 - \frac{1}{nT}\sum_{t=1}^{T}\sum_{i=1}^{n} P_i(t)[1 - FC_i(t)]$$

We refer to $SC$ as the probability of the system being in a safe state. This can also be understood as the effective fault coverage of the system, since it represents the average probability of not missing a fault. We use $SC$ as the metric to specify the target fault coverage in our evaluations.

**ATF Summary:** The ATF primarily benefits in terms of the test application efficiency. In the early lifetime, when the processing cores are healthy, a lot fewer tests suffice for achieving a given fault coverage target $SC$. With time, and device wearout, this testing overhead gradually rises. Overall, the application of fewer tests has multiple advantages:

1) more time available for actual job execution, 2) power/energy saving, and 3) low fault detection cost visible to the end user. The intended application of the ATF is to detect permanent faults. Another possible application is its use in systems that have variable reliability modes. For instance, a server can tune the coverage target $SC$ of the system based on the job it is running (higher $SC$ for a financial transaction, and lower $SC$ for a regular web page request).

The discussion of the ATF so far has been in the context of a traditional CMP. The key observation that helps the adaptive online testing is the variation in the health of CMP cores (spatially and temporally). The following subsection applies an extension of this to the StageNet CMP fabric, a highly flexible computing substrate.

### 4.3.2 Adaptive Testing for StageNet

This section introduces the StageNet (SN) fabric [43], an architectural concept that decouples stages of a pipeline for the purpose of fault tolerance. The real strength of SN fabric is in its ability to isolate broken stages within pipelines. Nevertheless, its flexibility can also assist in forming cores with an even greater variation in their health, thereby magnifying the benefits of the adaptive online testing. The rest of this subsection is broken into two parts, 1) introduction to the SN fabric and 2) application of adaptive testing to the SN.

#### 4.3.2.1 StageNet CMP Fabric

The SN design is a highly reconfigurable and adaptable multi-core computing substrate. It is designed as a network of pipeline stages, rather than isolated cores (Figure 4.5). A logical core in the SN architecture is referred to as a StageNetSlice (SNS). It is formed

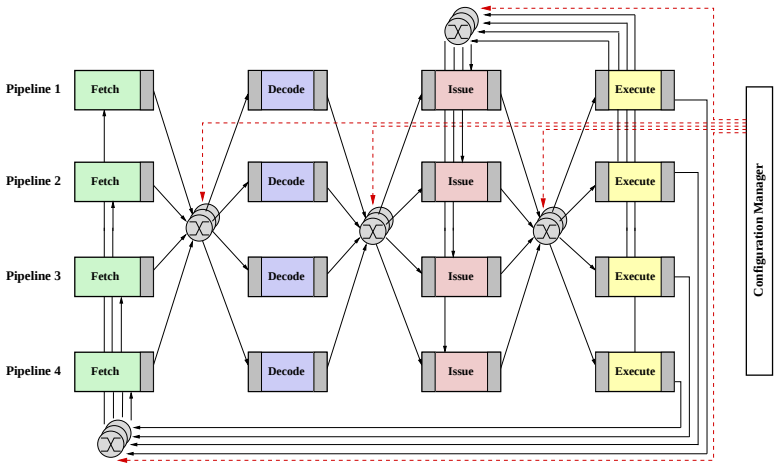



**Figure 4.5: StageNet fabric with four in-order pipelines woven together using 64-bit full crossbar interconnects. The interconnection configuration is managed by the configuration manager. Within StageNet, logical pipelines, can be constructed by joining any set of unique pipeline stages.**

by grouping together at lease one pipeline stage of each type. A SNS can easily isolate failures by adaptively routing around faulty stages. In the event of any stage failure, the SN architecture can initiate recovery by combining live stages from different slices, i.e. salvaging healthy modules to form logical SNSs. We refer to this as *stage borrowing*. In addition to this, if the underlying stage design permits, stages can be time-multiplexed by two distinct SNSs. For instance, a pair of SNSs, even if one of them loses its *issue* stage, can still run separate threads while sharing the remaining *issue* stage. We refer to this as *stage sharing*. Thus, a SN system possesses natural redundancy (through borrowing and sharing pipeline stages) and is, all else being equal, capable of maintaining higher throughput over the duration of a system's life compared to a conventional multi-core design.

The SN architecture consists of three prominent components:

a) *StageNetSlice (SNS):* The SNS is a basic building block for the SN architecture. It consists of a decoupled pipeline microarchitecture that allows convenient reconfiguration at the granularity of stages. The decoupling of stages makes the data forwarding and control

handling infeasible. Furthermore, the introduction of switches into the heart of a processor pipeline leads to significantly worse performance (4X slowdown over the baseline) due to high communication latencies between the stages. Fortunately, each of these problems can be solved with a few well placed microarchitectural additions (see [43]). With the application of the following optimizations, the performance of the SNS is within 11% of the baseline in-order pipeline.

- *Stream Identification:* Eliminates control hazard.
- *Scoreboard:* Tracks data hazards.
- *Bypass Cache:* Emulates data forwarding.
- *Macro Operations:* Amortizes transfer time of the interconnection network.

b) *Interconnection Switch:* The role of the switch is to direct the incoming instruction bundle to the correct destination stage using a routing table. The crossbar switches allows complete flexibility for a pipeline stage at depth *N* to communicate with any stage at depth *N+1*.

c) *Configuration Manager:* Given a pool of stage resources, the configuration manager divides them into a globally optimal set of logical SNSs.

The lifetime reliability results for SN demonstrated nearly 50% improvement in the cumulative work compared to a traditional CMP [43]. Furthermore, the high resiliency of the SN fabric can be leveraged to combat process variation and manufacture time defects, in addition to the wearout failures.

#### 4.3.2.2 Adaptive Testing

At any point in the lifetime, because of the manufacture time process variation and the device wearout, different pipeline stages within the SN fabric would exhibit different amounts of degradation. A snapshot of the SN fabric in Figure 4.6 shows the varying degrees of degradation between pipeline stages of the system. For the sake of illustration, four health levels are shown from lightest shade (best health) to the darkest shade (worst health). Let us say that the health assessments (that provide probability of failure) map to levels 1-4 of test thoroughness (test fault coverage). In the case of a traditional CMP, the ATF decides the test thoroughness on the basis of weakest component in a core/pipeline. For instance, even if only one stage within a pipeline is badly worn out, ATF for a traditional CMP assigns a thorough test program to that pipeline. Going by this principle, pipeline 1 would apply level 2 test, pipeline 2 - level 3 test, and pipelines 3,4 - level 4 tests. In contrast, the SN can make the testing more efficient by grouping together stronger components separately from weaker components. The bold lines in the figure show the pipeline stages that are combined to formed logical SNSs. First logical SNS (P1) would need to apply level 1 test, P2 - level 2, P3 - level 3 and P4 - level 4. Thus, SN achieves a reasonable amount of test reduction over a traditional CMP.

In order to separate out the stronger pipeline resources from the weaker ones, we sort stages of each type on the basis of their health. For instance, in Figure 4.6, *fetch* stages are already sorted based on their health (from the top to the bottom pipeline). The stages with equal health ranks are connected to form logical pipelines. These health rankings of the stages can vary over the lifetime depending up on the stress experienced by different

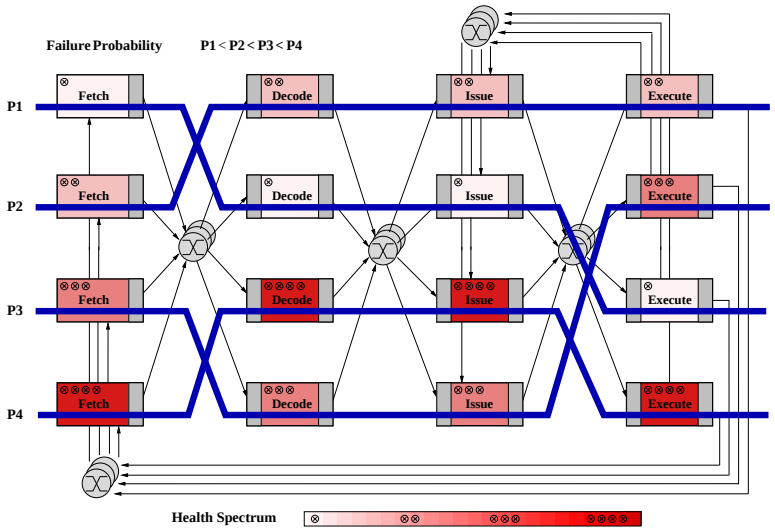



**Figure 4.6: The shading intensity of stages represents their deterioration. Thus, a darker stage has a higher failure probability and vice-versa. SN flexibility allows connecting stages with similar health, forming logical pipelines.**

stages in the system. Fortunately, the flexibility in the SN system allows it to dynamically segregate stronger and weaker components at will (after every checkpoint interval).

## 4.4 Evaluation

### 4.4.1 Methodology

For evaluating the potential of the proposed approach in reducing the testing overhead, we conduct lifetime reliability experiments. This is required in order to measure the cumulative reduction in the amount of test instructions over the system's lifetime. A CMP is modeled consisting of 16 ARM9-v4 compatible RISC processors. The SN CMP is configured as four group of 4-pipeline wide SN blocks. The operating frequency was set to 1GHz at 130nm IBM process. The systematic and random process variations were modeled using VARIUS [91]. Oxide breakdown (OBD) was used as the representative wearout mechanism with degradation equations from [102, 55]. This choice was motivated by the

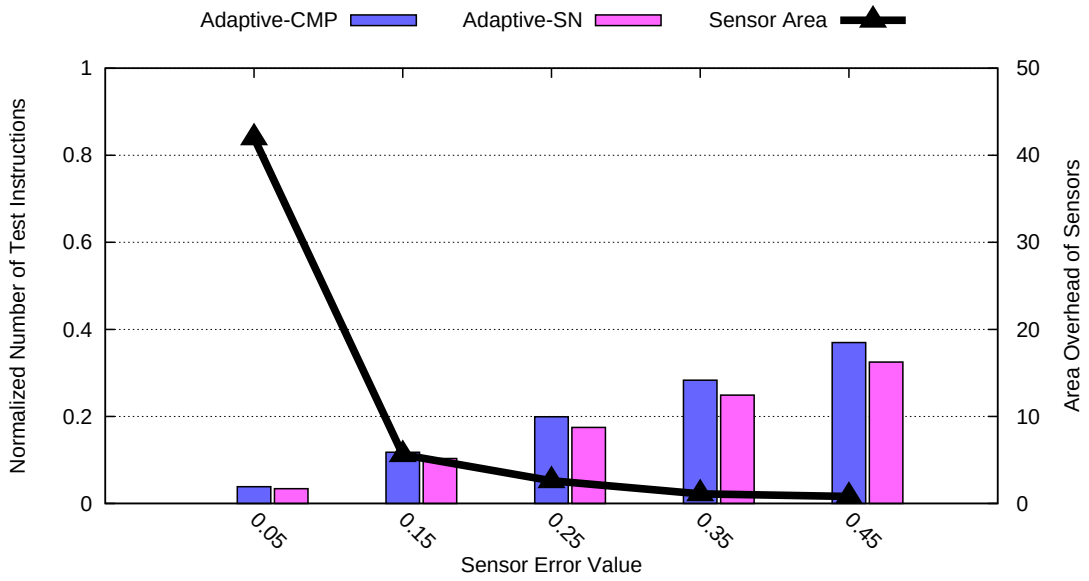



**Figure 4.7: Number of test instructions for the adaptive online testing in CMP and SN with varying amount of sensor error. The number of test instructions are normalized to a regular CMP with fixed periodic testing. The plot also shows the sensor area overhead used by the proposed approach for health assessment. The coverage target ($SC$) is fixed at 97.3%.**

presence of accuracy data for the low level OBD sensors [56]. A variable number of these OBD sensors were deployed within the cores for the health assessment.

The lifetime experiments are conducted as a series of interval simulations. Each interval simulation updates the sensor readings, and allocates the appropriate size of tests to the cores based on their probabilities of failure ($P_i$). For a given fault coverage ($FC_i$) (as determined by the test allocator), the number of test instructions executed is extracted from the data plotted in Figure 4.2. The maximum achievable fault coverage for testing is 97.3% and is bounded by the SBST scheme that we employ [69]. The presented results use the system fault coverage metric $SC$ as derived in the Section 4.3 wherever we refer to coverage target.

### 4.4.2 Results

Figure 4.7 shows the number of instructions used (over the CMP's lifetime) by the ATF, normalized to a baseline CMP system which applies a constant amount of test (given a coverage target). The target system coverage is set to 97.3% (best achievable by the chosen SBST scheme [69]), and the test instructions reported are accumulated over the lifetime. For a 5% sensor error in the health assessment, about 96% of the test instructions are saved while using the proposed ATF. As the number of sensors is reduced (thereby making the reading less accurate), only a more conservative estimate of failure probability is possible, forcing the adaptive system into assigning bigger tests to all system processors. However, even with the higher levels of sensor error, the benefits erode gradually, and the proposed scheme can deliver up to 82% test time saving with 25% sensor error. We believe this point offers a good trade-off between the sensor area overhead (2.6%) and the saving in the test instruction count (82%). Thus, our scheme does not depend on high sensor accuracy levels to achieve test reduction.

Figure 4.8 uses the similar terms as the one before, and presents the test instruction savings for a range of system coverage targets. The sensor error is fixed at 25% for these results. Depending upon the reliability requirements of a system, the coverage target can be dynamically tuned. For instance, a move lower to 88% coverage target can result in an over 90% test instructions saving. The increasing divergence (when going towards higher coverage) between the saving obtained using adaptive CMP and adaptive SN is also noteworthy. We expect the adaptive SN to well surpass the benefits of adaptive CMP in high coverage target scenarios. For future technology nodes, with higher levels of process vari-

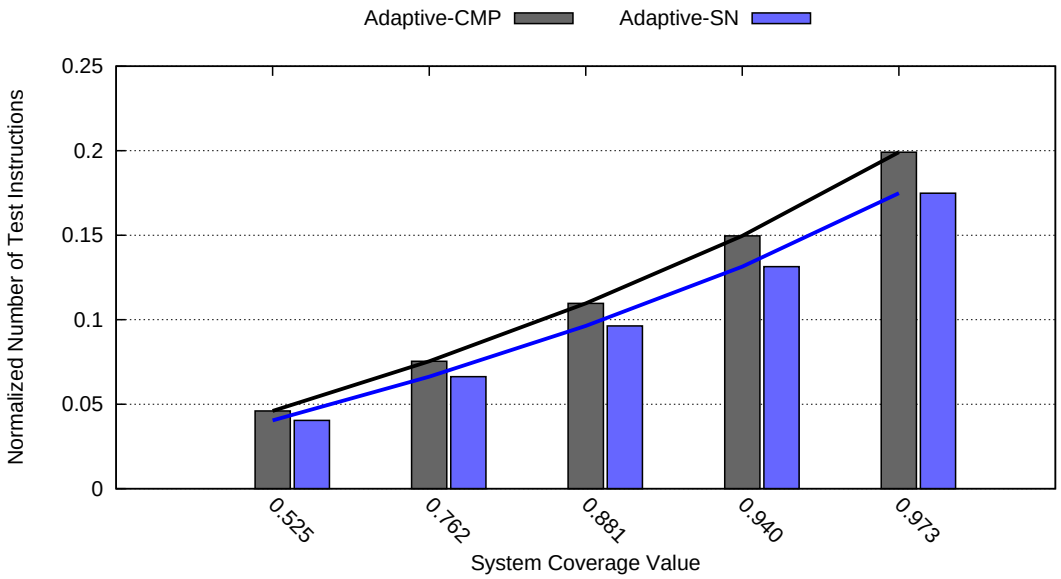



**Figure 4.8: Number of test instructions for the adaptive online testing in CMP and SN with varying system coverage target ($SC$). The number of test instructions are normalized to that needed by a CMP with non-adaptive testing.**

ation, a SN based system would be capable of extracting even bigger gains by segregating stronger resources from the weaker ones. That way, much fewer pipelines would need a thorough testing.

The result plots so far have presented a cumulative value for the number of test instructions over the entire lifetime, in this next result, we present the data of test thoroughness over time. Figure 4.9 plots a three dimensional plot with average number of test instruction executed in consecutive simulation intervals for a range of coverage target values. This plot is for the SN system with $25\%$ sensor error. Here, the trend of the number of test instructions over time reveals an interesting behavior of the proposed scheme. For extremely low coverage targets, say 0.5 (or 50%), hardly any test instructions are applied. However, for higher values of coverage target, there is a rhythmic pattern of the test instruction count over the lifetime. The number of test instructions rise to a peak, and then fall-off. This peak formation is representative of a core nearing its time to failure, and then failing sub-

sequently. As a core reaches close to its failure time, the adaptive system ramps up the number of test instructions to guarantee the coverage target. Once the core fails, the system returns to a nominal state since most of the other cores are healthy. There are 16 such peaks in this plot, each representing dying time of a core. Overall average for the number of test instructions is higher later in the lifetime due to the poorer health of many cores in the system. This plot is a clear demonstration of the proposed adaptive testing framework in tuning the testing time with the probability of failure. In contrast, a traditional periodic testing approach will exhibit a flat surface with constant testing intensity.

All the savings that we have reported for the test instructions, can translate into a range of benefits in a target system: 1) *performance* gain from spending less time for test; and 2) *power* and *energy* saving from running fewer instructions. For the system that we simulate (16 core CMP) with a checkpoint interval of 10ms, the performance overheads are 7%, 1.85% and 1.6% for CMP testing, CMP adaptive testing and SN adaptive testing, respectively. According to Revive [84], a 10ms checkpoint interval would require 20MB storage on an average and up to 100MB peak storage requirement. A smaller allocation of storage to the checkpoint mechanism can force the checkpoint intervals to be even shorter, making the testing time even more significant.

## 4.5 Summary

With the looming reliability challenges in future technology generations, in-field tolerance to silicon defects will be a necessity in future computing systems. Periodic online testing, although a good fit to this problem, imposes heavy test time overheads. The pro-

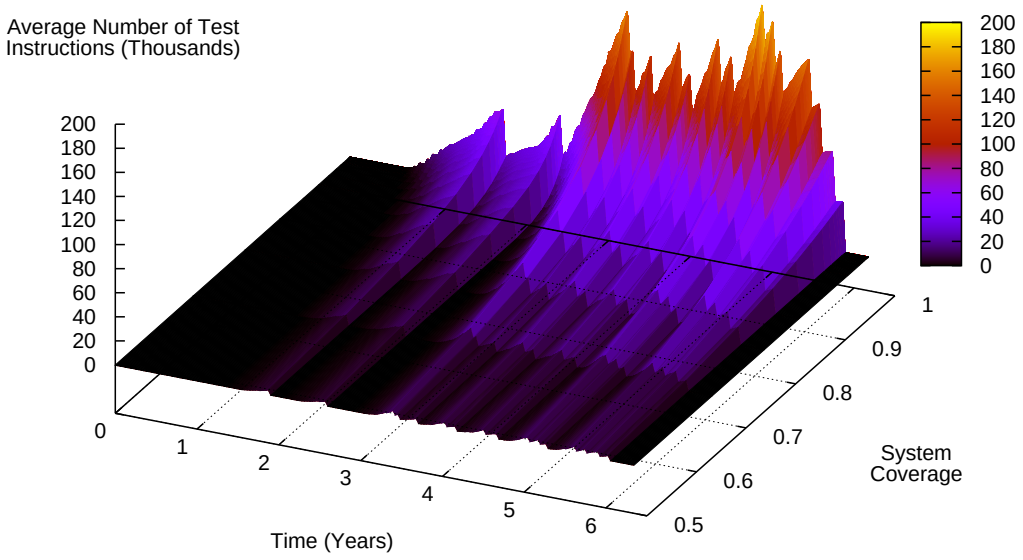



**Figure 4.9: This plot shows the variation in the average number of test instructions executed in the CMP system over its lifetime for a range of system coverage targets.**

posed adaptive test framework significantly reduces this testing overhead. The key insight is to leverage low level sensors to assess failure probability of various system resources, and suitably apply the tests. This way, a healthy system uses a fraction of resources for testing compared to another one nearing its time to failure. Over the lifetime, testing detail is adaptively managed by the proposed solution. The lifetime simulation for a system with 2.6% area devoted to health assessment sensors, resulted in an 80% reduction in the software test instructions while delivering the same fault coverage. We further extend this reduction by 12% when applying the adaptive testing to the StageNet architecture. This test time reduction can translate to varying levels of benefits in power, performance and energy depending up on the attributes of the targeted system. Overall, we believe, that the adaptive online testing offers an economical solution to the challenge of online fault detection.

# CHAPTER V

# Erasing Core Boundaries for Robust and Configurable Performance

## 5.1 Introduction

The introduction of this thesis lists the three major challenges that need addressing by the semiconductor manufacturers: reliability, performance, and energy-efficiency. In this landscape of multicore challenges, prior research efforts have focused on addressing these issues in isolation. For example, to tackle single-thread performance, a recent article by Hill and Marty [46] introduces the concept of *dynamic multicores* (Figure 5.1(a)) that can allow multiple cores on a chip to work in unison while executing sequential codes. This notion of *configurable performance* allows chips to efficiently address scenarios requiring throughput computing, high sequential performance, and anything in between. Core Fusion [52], Composable Lightweight Processors [60] and Federation [106] are representative works with this objective. However, the scope of present day *dynamic multicore* solutions is limited as they cannot provide customized processing, as in [63, 74], or better throughput sustainability, as achieved by techniques in [88, 43]. The customized process-

ing in [63] (Figure 5.1(b)) is typically accommodated by introducing heterogeneity of types and number of functional units, execution models (in-order, OoO), etc., into different cores. Whereas, better throughput sustainability can be provided by fine-grained reliability solutions like CCA [88] and StageNet [43] (Chapter II), that disable broken pipeline stages, instead of entire cores (Figure 5.1(c)), within a multicore.

Unfortunately, by virtue of being independent efforts, combining existing performance, power and reliability solutions for multicores is neither cost-effective nor straightforward. The overheads quickly become prohibitive as the changes required for each solution are introduced, with very little that can be amortized across multiple techniques. Configurable performance requires dedicated centralized structures (adding drawbacks such as access contention/latency, global wiring), customization requires a variety of static core designs, and fine-grained reliability requires either large amounts area for cold spares or the flexibility to share resources across cores. Apart from excessive overheads, a direct attempt to combine these solution also faces engineering hurdles. For instance, when combining CoreFusion [52] (a configurable performance solution) and StageNet (Chapter II) (a fine-grained reliability solution), two prominent issues arise: 1) CoreFusion requires centralized structures for co-ordinating fetch, steering, commit across fused pipelines. These structures become single points of failure and limit reliability benefits of StageNet. 2) StageNet requires a decoupled microarchitecture for its sub-core defect tolerance. This is not compatible with CoreFusion, as resources within a single CoreFusion core are tightly coupled together.

Instead of targeting one challenge at a time, the goal of this chapter is to devise a design philosophy that can naturally be extended to handle a multitude of multicore chal-

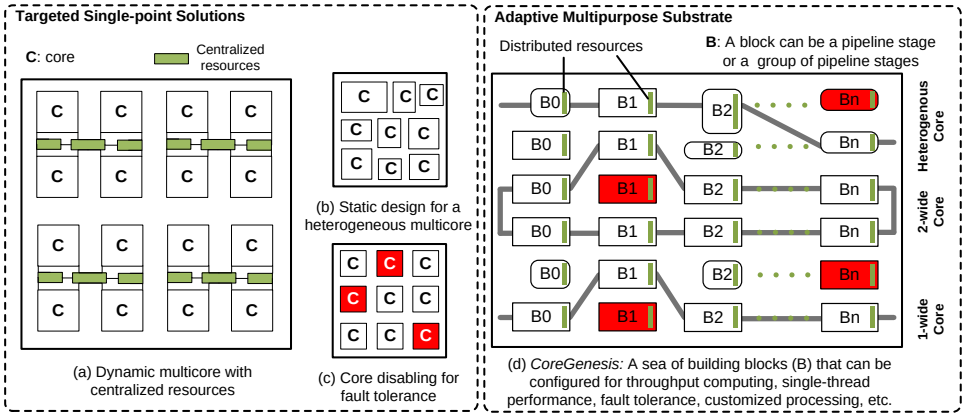



**Figure 5.1: Contemporary solutions for multicore challenges (a,b,c) and vision of this work (d). In (a), centralized resources are used to assist in fusing neighboring cores. In (b) and (d), different shapes/sizes denote heterogeneity. In (c) and (d), dark shading marks broken components.**

lenges seamlessly, while overlapping costs, maintaining efficiency and avoiding centralized structures. Towards this end, this chapter proposes the CoreGenesis (CG) architecture (see Figure 5.1(d)), an adaptive computing substrate that is inherently flexible, and can best align itself to the immediate system needs. CG eliminates the traditional core boundaries and organizes the chip multiprocessor as a dynamically configurable network of building blocks. This sea of building blocks can be symmetric or heterogeneous in nature, while varying in granularity from individual pipeline stages to groups of stages. Further, the CG pipeline microarchitecture is decoupled at block boundaries, providing full flexibility to construct logical processors from any complete set of building blocks. Another key feature of the CG proposal is the use of distributed resources to coordinate instruction execution across decoupled blocks, without any significant changes to the ISA or the execution model. This is a major advancement over prior configurable performance works, and addresses the shortcomings of centralized resources.

Resources from CG's sea of blocks can be fluidly allocated for a number of performance, power and reliability requirements. Throughput computing can be optimized by

forming many single-issue pipelines, whereas sequential performance can be accelerated by forming wider-issue pipelines. Power and performance characteristics can be further improved by introducing heterogeneous building blocks in the fabric, and appropriately configuring them (dynamically or statically) for active program phases or entire workloads. This enables a dynamic approach to customized processing. Finally, fault tolerance in CG can be administered at the block granularity, by disabling the broken components over time.

Guided by this architectural vision, in this chapter, we present a CG instance that targets configurable performance and fine-grained reliability. For the fabric, an in-order pipeline model is used with single pipeline stages as its building blocks. As a first step, we define mechanisms for decoupling pipeline stages from one another (inspired by the StageNet architecture [43]). This enables salvaging of working stages from different rows of the fabric to form logical processors, thereby tackling the throughput sustainability challenge. To address configurable performance, we generalize the notion of logical processors to form processors of varying issue widths.

The engineering of distributed resources to support the assembly of decoupled pipeline stages into a wide-issue processor is especially hard due to the heavy co-ordination and communication requirements of an in-order superscalar. Our solution adopts a best effort strategy here, speculating on control and data dependencies across pipeline ways, and falling back to a light-weight replay in case of a violation. To register these violations, hardware schemes were formulated for distributed control, register and memory data flow management. The frequency of data flow violations from instructions executing on two different pipeline ways was found to be a leading cause of performance loss. We address this by incorporating compiler hints for instruction steering in the program binary. This

**Table 5.1: Comparison to Prior Work**

| | Configurable Performance | Fine-grained Reliability | No centralized structures | Supports in-order model | Supports heterogeneity |
|---|---|---|---|---|---|
| CG (this chapter) | ✓ | ✓ | ✓ | ✓ | ✓ |
| CLP [60] | ✓ | | ✓ | ✓ | |
| Core Fusion [52], Federation [106] Multiscalar [98] | ✓ | | | | |
| StageNet [43], CCA [88] | | ✓ | ✓ | ✓ | |
| Heterogeneous CMPs [63] | | | ✓ | ✓ | ✓ |

circumvents the hurdles in fusing in-order cores, as presented in [89], while also achieving a near-optimal pipeline way assignment. Overall, the manifestation of CG presented in this chapter relies on interconnection flexibility, microarchitectural innovations, and compiler directed instruction steering, to provide a unified performance-reliability solution.

## 5.2 Related Work

Within the framework of multicore chips, efficient solutions that can deliver configurable performance and throughput sustainability are desirable. This section gives an overview of prior works targeting these issues. Table 6.1 summarizes the key aspects of CG in comparison to the relevant prior proposals. CG stands out by simultaneously offering configurable performance and fine-grained reliability while eliminating centralized structures. This section also presents a study that motivates a need for unified performance-reliability solutions for the sake of efficiency.

### 5.2.1 Single-Thread Performance Techniques

**Dynamic multicores.** Dynamic multicore processors consists of a collection of homogeneous cores that can work independently to provide throughput computing, or a subset of them can be fused together to provide better single-thread performance. Core Fusion [52]

is a dynamic multicore design that enables the fusion of adjacent OoO cores to form wider-issue OoO processors. Federation [106], on the other hand, combines neighboring in-order cores to form an OoO superscalar. Both these approaches employ centralized structures (for fetch management, register renaming, instruction steering, etc.) to assist in aggregation of pipeline resources. In contrast, Composable Lightweight Processors (CLP) [60] leverages the EDGE ISA and compiler support to eliminate centralized structures, enabling it to scale up to 64-cores. CG also eliminates centralized structures, but its compiler support is limited to generating hints for instruction steering, and ISA is modified to include this hint carrying instruction. Multiscalar [98] is a seminal work that can compose a large logical processor from many smaller processing elements. It uses an instruction sequencer to distribute task sub-graphs among the processing elements, and relies on hardware to satisfy dependencies. However, in all these prior schemes, resources within individual cores are tightly coupled together, dismissing the opportunity for fine-grained reliability.

Another distinction of CG is that it fuses in-order pipelines to form wider-issue in-order processors. While out-of-order fusion provides opportunities for hiding latency (large instruction window sizes), in-order fusion is made harder due to the negligible room for inefficiency. In fact, Salverda et al. [89] argue that in-order pipeline fusion is impractical because of the associated hardware overheads for interleaving active data flow chains (instruction steering). CG circumvents these challenges by using compiler hints to guide instruction steering, and employing simple mechanisms to detect and recover from data flow violations.

**Heterogeneous CMPs.** Heterogeneous designs exhibit good power and performance char-

acteristics for their targeted class of applications. However, being a static design, its effectiveness is limited outside this set or when flexibility is desired. For instance, in a scenario where all applications prefer throughput computing, a heterogeneous CMP will operate sub-optimally.

In addition to static scheduling of jobs on heterogeneous CMP cores, there have also been dynamic scheduling approaches to match program phase behaviors to cores. Core contesting [74] is one such example, but it runs the same program redundantly on different cores to allow a faster transfer of state between them. In CG, inclusion of heterogeneous blocks can allow static, dynamic as well as fine-grained dynamic exploitation of program phase to architecture mapping. This is possible due to CG's inherent flexibility to swap resources between pipelines.

**Clustered Architectures.** The early research in clustered architectures was to enable wider issue capabilities, without adding sophisticated hardware support. The Multicluster [37] architecture is a good example of this, and it uses static instruction scheduling from compile time. CG, on the other hand, uses a compiler clustering algorithm [33] to generate hints that are used for dynamic instruction steering. This is also in contrast to past works that solely use hardware support [11] to implement heuristics for distributing instructions among clusters in a superscalar.

### 5.2.2 Multicore Reliability Solutions

**Coarse-Grained Reconfiguration.** All reliability solutions that administer reconfiguration at the granularity larger than or equal to a processor core fall into this category. Some of

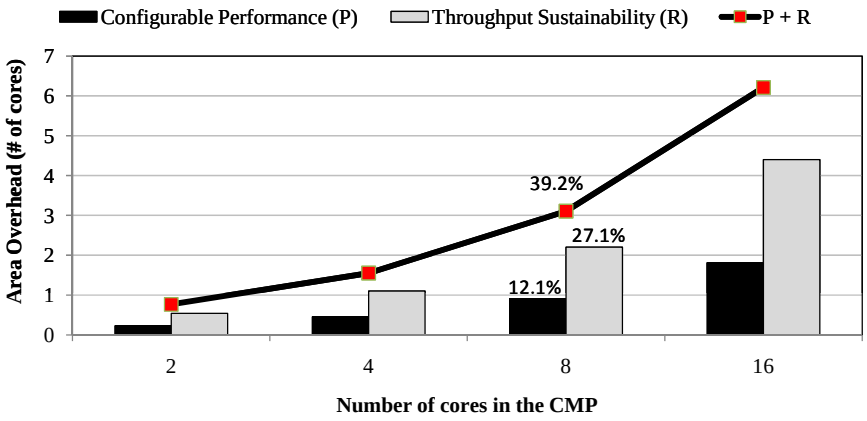



**Figure 5.2: Area overhead projections (measured as number of cores) for supporting configurable performance (P) and throughput sustainability (R) in different sized CMP systems. P+R curve shows the cumulative overhead. For this plot, throughput sustainability is defined as the ability to maintain 50% of original chip's throughput after three years of usage in the field.**

the prominent works being [12, 105, 2]. Details and discussions in Section 2.6.

**Fine-Grained Reconfiguration.** A newer category of techniques use stage-level reconfiguration (isolates broken stages, not cores) for reliability. The StageNet design in Chapter II is a leading example of fine-grained reconfiguration. It groups together a small set of pipelines stages with a simple crossbar interconnect. By enabling reconfiguration at the granularity of a pipeline stage, StageNet can tolerate a considerable number of failures. In CG, fine-grained reconfiguration is supported in the same way as StageNet. More examples of fine-grained reconfiguration appear in Section 2.6.

### 5.2.3 Combining Performance and Reliability

All prior works target the multicore challenges separately, either configurable performance or throughput sustainability (reliability). The central problem here is that solutions for each of these require new hardware to be incorporated into existing CMPs. This turns out to be an expensive proposition, as the hardware costs are additive. We conducted a small

study to assess this cost. Figure 5.2 shows the results from this study using Core Fusion [60] as the configurable performance solution, and standard core disabling for throughput sustainability. The line plot shows the cumulative overhead of performance (Core Fusion) and reliability (core disabling) solutions (P+R). Resulting overhead is almost 40% additional area. There are two factors at play here: 1) costs are additive, as the two solutions share nothing in common, 2) reliability is administered at core level (instead of being fine-grained). On top of this, the design, test, verification and validation efforts need to be duplicated for performance and reliability separately. The next section presents CG, our unified performance-reliability solution, that overcomes these issues to a large extent.

## 5.3 The CoreGenesis Architecture

### 5.3.1 Overview

The manifestation of CoreGenesis (CG) architecture presented here is a unified performance-reliability solution that allows fusion of standalone cores for accelerating single-thread performance as well as isolation of defective pipeline stages for sustainable throughput. The CG fabric consists of a large group of pipeline stages connected using non-blocking crossbar switches, yielding a highly configurable multiprocessor fabric. These switches replace all direct wire links that exist between the pipeline stages including the bypass network, branch mis-prediction signals and stall signals. The pipeline microarchitecture within CG is completely *decoupled*, and all pipeline stages are standalone entities. The symmetric crossbar interconnection allows any set of unique stages to assemble as a logical pipeline.

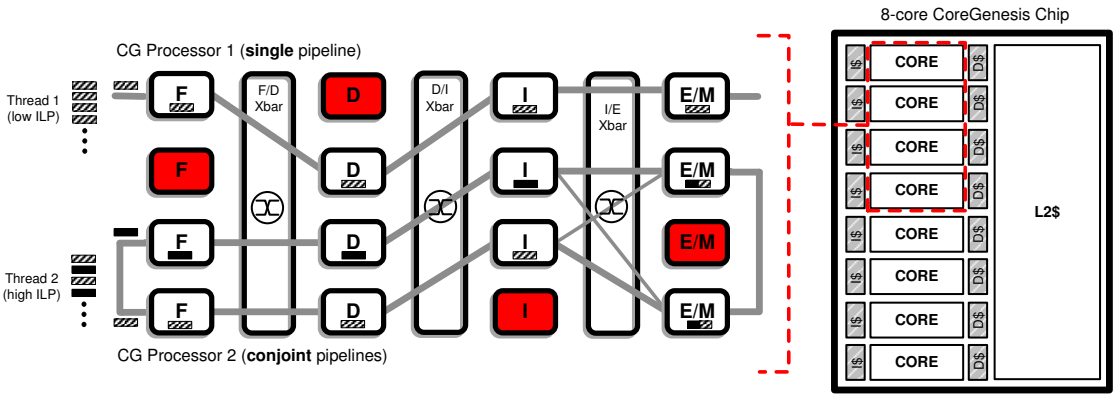



**Figure 5.3: An 8-core CoreGenesis (CG) chip with a detailed look at four tightly coupled cores. Stages with permanent faults are shaded in red. The cores within this architecture are connected by a high speed interconnection network, allowing any set of stages to come together and form a logical processor. In addition to the feed-forward connections shown here, there exist two feedback paths: E/M to I for register writeback and E/M to F for control updates. In CG processor 2 (conjoint pipelines), instructions (prior to reaching E/M stage) can switch pipelines midway, as a result of dynamic steering.**

As a basis for the CG design, an in-order core is used[1], consisting of five stages namely, fetch (F), decode (D), issue (I), execute/memory (E/M) and writeback [9]. Figure 5.3 shows the arrangement of pipeline stages across four interconnected cores and a conceptual floorplan of an 8-core CG chip. Note that all modifications introduced within CG are limited to the core microarchitecture, leaving the memory hierarchy (private L1 / unified L2) untouched. Further, the caches are assumed to have their own protection mechanism (like [7]), while CG tolerates faults within the core microarchitecture.

In Figure 5.3, despite having one stage failure (shaded) per core, CG is able to salvage three working pipelines. Further, given a set of active threads, CG can judiciously allocate these pipeline resource to them in proportion to their instruction level parallelism. For instance, in the figure, thread 1 (low ILP) is allocated one pipeline and thread 2 (high ILP) is allocated the remaining two pipelines.

[1]Deeper and more complex pipelines can be segmented at logical boundaries of elementary pipeline stages (F,D,I,E,W) to benefit from the CG approach.

Configurable performance helps CG in dealing with the software diversity present in modern day CMP systems. It keeps pipelines separate for throughput computing and dynamically configures two (or more) pipelines into a multi-issue processor for sequential workloads. This morphing of individual pipelines into a *conjoint processor* requires no centralized structures, maintains reliability benefits, and is transparent to the programmer. In Figure 5.3, CG processor 2 is an example of a conjoint processor assimilated using two pipeline stages of each type. As part of a conjoint processor, the two pipelines cooperatively execute a single thread. The instruction stream is fetched alternately by the two pipelines - odd numbered ops by one pipeline and the even numbered ops by the other. All instructions are tagged with an *age* to maintain the program order during execution and instruction commit. Regardless of where an instruction is fetched, it can be executed on either of the two pipelines depending upon its source operands. We refer to this as *instruction steering*. An instruction executing on the same pipeline that fetches it is said to be straight-steered, while that executing on some other pipeline is said to be cross-steered. This dynamic instruction steering is performed with an objective of minimizing data dependency violations, and is critical for achieving true multi-issue performance. CG employs a compiler level analysis for statically identifying data dependency chains (Section 5.3.5) and the issue stage applies this knowledge (during run-time) to steer instructions to the most suitable pipeline.

The natural support for fine-grained reconfiguration allows CG to achieve its second objective of throughput sustainability. For instance, in Figure 5.3, CG is able to efficiently salvage the working stages from the pool of defective components to form functional processors. By the virtue of losing resources at a smaller granularity, isolation of broken pipeline stages reaps far better rewards than traditional core disabling. To realize its reliabil-

ity benefits, the CG system relies on a fault detection mechanism to identify broken stages and a software configuration manager to consolidate the working ones (by reprogramming the crossbars). Fault detection can be achieved using a combination of manufacture-time and in-field periodic testing. Chapter IV discusses an instance of periodic testing solution for in-field fault detection.

### 5.3.2 Challenges

Although the performance and reliability benefits of its configuration flexibility are substantial, there are a number of hurdles faced by the CG architecture. There are four principal challenges, and they span correctness and performance issues for both single pipeline processors as well as conjoint pipelines processors:

**Control flow management:** The decoupled nature of the CG pipeline makes global signals such as pipeline flush and stall infeasible. In the context of a single pipelines, the control flow management is crippled by the absence of a global flush signal. The problem is even more severe in the case of conjoint processors. Pipeline fetch stages need to read complementary instructions from a single program stream, and make consistent decisions about the control flow (i.e., whether to take a branch or not).

**Register data flow management:** Back-to-back register data dependencies are typically handled by the operand bypass network, which relies on timely inter-stage communication. Unfortunately, the decoupled design of CG pipelines makes the bypass network impractical. In the case of conjoint processors, this problem is further aggravated by the presence of cross pipeline register dependencies. The decentralized

**Table 5.2: CoreGenesis (CG) challenges. The challenges can be classified on the basis of single and conjoint pipeline configurations. The check marks (✓) are used for solutions that were straightforward extension of prior work on decoupled architectures. Whereas the question marks (?) are open problems that are solved in this chapter.**

| | Control flow | Register data flow | Memory data flow | Instruction steering |
|---|---|---|---|---|
| Single pipeline | ✓ | ✓ | N/A | N/A |
| Conjoint pipelines | ? | ? | ? | ? |

instruction execution needs a mechanism to track dependencies, detect violations, and replay instructions for guaranteeing correctness.

**Memory data flow management:** Memory instructions are naturally serialized in the case of a single pipeline CG processor, as all of them reach the same memory stage. However, similar to register data flow violations, memory data flow violations can also occur between pipelines of a conjoint processor, leading to a corruption in global state.

**Instruction steering:** In a conjoint processor, issue stages have the option to straight steer the instructions to same pipeline or cross steer it to the other pipeline. This decision has to be dynamically made for every instruction such that the number of cross pipeline data dependencies is minimized. A recent study by Salverda et. al [89] establishes that steering is central to the challenge of in-order pipeline fusion, and further concludes that a hardware-only steering solution is impractical.

Table 5.2 summarizes all the challenges in the context of single and multiple pipelines working as a logical processor. A subset of these challenges have been solved (marked with a ✓) by a prior work, StageNet(SN) [43]. SN is a decoupled pipeline microarchitecture for fine-grained fault tolerance. The interconnection bandwidth solution from SN is generic

and applies to both single/conjoint scenarios.

The control, register data flow, memory data flow, and instruction steering solutions for conjoint processors are contributions of this chapter (marked with a *?*). All of these are new mechanisms, and were made harder by the fact that unlike a true multi-issue machine (and even Core Fusion [52]), CG does not have centralized structures, and needs to get performance by combining very loosely coupled resources. For the sake of completeness, in the descriptions that follow, we also provide a quick overview of the solutions for single pipeline case from [43].

### 5.3.3 Microarchitectural Details

This section describes microarchitectural changes needed by the CG architecture, a majority of which are clever tricks to detect control and data flow violations in a distributed fashion. The relatively complex task of instruction steering is off-loaded to the compiler (Section 5.3.5).

#### 5.3.3.1 Control Flow

**Single Pipeline.** For a single pipeline CG processor, the absence of a global pipeline flush signal complicates the control flow management. In the event of a branch mis-prediction, the decoupled pipeline needs a mechanism to squash the instructions fetched along the incorrect path. The introduction of a 1-bit stream identification (SID) to all the in-flight instructions targets this problem [43]. The basic idea is to use the SID for distinguishing instructions on the correct path from those on the incorrect path. The fetch and the execute stages maintain single bit SID registers, both of which are initialized to the same value (the discussion here is simplified, the actual scheme adds a SID register to every stage).

The fetch SID is used to tag all incoming instructions. And, the execute stage matches an instruction's SID tag against the execute SID before letting it run. If at any point in time, a branch instruction is resolved as a mis-prediction by the execute stage, the execute SID is toggled and the update is sent to the fetch stage. All in-flight instructions that are tagged with the stale SID are recognized (by the execute) to be on the incorrect path and are systematically squashed over time. In parallel to this squashing, after receiving the branch update from the execute, the fetch toggles its own SID and starts fetching correct path instructions. Note that a single bit suffices here because the pipeline execution model is in-order and can have only one resolved branch mis-predict outstanding at any given time (since all instructions following it become invalid).

**Conjoint Pipelines.** In a dual-pipeline conjoint processor, one pipeline is designated as the *leader* and other as the *follower*. To balance the usage, both pipelines fetch alternate instructions from the program stream, i.e., if leader fetches from $PC$, follower fetches from $PC+4$. The logical program order is maintained by tagging every instruction with a unique (monotonically increasing) `age` tag. Fetch stages are augmented with age counters (offset by 1) that are incremented in steps of two whenever an instruction is fetched and tagged. Thus, the leader pipeline will tag instructions with ages 0, 2, 4, and so on; and follower will tag them with ages 1, 3, 5, and so on. By virtue of interleaving program counter values, both pipelines together fetch the complete program stream and record the program order in the `age` tags. These tags are later used by the execute $\rightarrow$ issue crossbar (EI xbar) to commit instructions in the program order.

The above description of distributed fetch works fine until a branch instruction is en-

**Table 5.3: Control cases. Each case represents a pair of consecutive program instructions in a 2-issue conjoint processor. The first and second rows in this table show the instructions fetched in the leader and follower pipelines, respectively.**

| **Case 1** branch not taken | **Case 2** branch not taken | **Case 3** branch taken | **Case 4** branch taken |
|---|---|---|---|
| OP | BR | OP | BR |
| BR | OP | BR | OP |

countered. For proper operation, CG needs a decentralized control handling mechanism that keeps both pipelines in sync when making a control decision. The control flow can encounter four distinct cases shown in Table 5.3.

Cases 1 and 2 are the most straightforward ones, as the branch is not taken. Both pipelines continue as normal as the branch has no impact on the control flow. For case 3, we need both pipelines to take the branch simultaneously. This can be achieved if their branch predictors completely mirror each other and same address look-up is performed by both pipelines. We maintain this mirroring by sending all branch prediction updates (from execute/memory stage) to both fetch stages. For consistent look-ups, the leader pipeline addresses its branch predictor using $Leader\_PC + 4$, and the follower addresses it using $Follower\_PC$ (and by design $Follower\_PC = Leader\_PC + 4$). As both the predictors are synchronized, they will return the same prediction and target address. Finally, for case 4, we again need both pipelines to take the branch. In addition to the mechanism for case 3, the follower pipeline must also invalidate its $OP$ which is on the wrong path. A simple logic is added to the decode stage to carry this out. The decode stage invalidates any operation that is 1) in the follower pipeline *and* 2) is predicted as a taken branch by the fetch *and* 3) is not a real branch instruction.

In the case of a branch mis-predict, the squashing of instructions for conjoint processors is a direct extension of the SID scheme presented for single pipelines. In conjoint proces-

sors, both pipelines maintain a single logical value for the SID, and all branch resolution updates are sent back concurrently to the fetch stages.

#### 5.3.3.2 Register Data Flow

**Single Pipeline.** The data forwarding within a single pipeline can be emulated using a small *bypass* cache in the execute stage. The key idea is to use this bypass cache for storing results from recently executed instructions, and supplying them to later instructions. The experiments in [43] show that a bypass cache that holds last six results is sufficient.

**Conjoint Pipelines.** For conjoint processors, the data flow management gets involved due to the distributed nature of execution. The instructions are issued and executed on different pipelines, and cross-pipeline register data dependencies can occur frequently (instruction fetched by pipeline X, but needs register produced by pipeline Y). In an ideal scenario, we would like issue stages to always steer the dependent instructions to the execute which most recently produced the source values. More of this discussion on instruction steering follows later in Section 5.3.5. Nevertheless, in a practical design, the steering mechanism is bound to make some mistakes as each pipeline's issue stage has incomplete information about the in-flight instructions. Our solution, in a nutshell, is to have each pipeline maintain a local version of the outstanding data dependencies, and monitor write-backs by the other pipelines to detect any data flow violations that might have occurred. Upon detecting such a violation, a replay is initiated.

The first requirement for data flow management is proper maintenance of the register file. The register files for all pipelines (that constitute a conjoint processor) are kept coherent with each other. This is achieved by sending all register write-backs to both issue

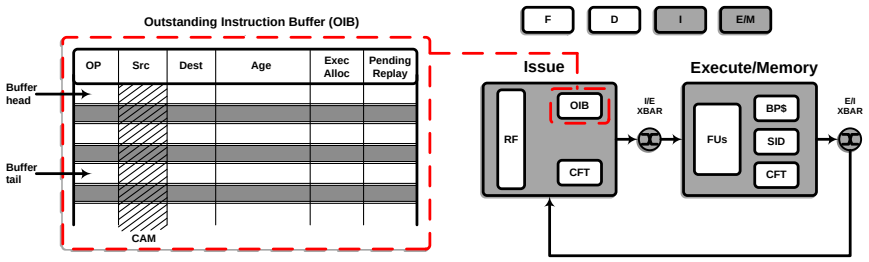



**Figure 5.4: CG pipeline back-end with structures for detecting register data flow violations and initiating replays. The outstanding instruction buffer (OIB) and current flow tag (CFT) registers are the two additions for conjoint processors. Also shown here is the bypass cache (BP$) for data forwarding within a single pipeline.**

stages simultaneously, similar to the way Alpha 21364 [3] kept its two clusters consistent. Further, the write-backs from the two pipelines are serialized by the network interface between the execute and the issues stages. The crossbar switch prioritizes the write-back based on the age tag of the instructions, maintaining correct program commit order. This way, cross-pipeline data dependencies, which are sufficiently far away in the program, go through the register file. However, all the instructions that are issued before their producers have written back to the register file remain vulnerable to undetected data flow violations.

To catch such undetected data flow violations, each pipeline can track locally issued (in-flight) instructions and monitor the write-backs to detect any data dependency violations. We accomplish this using a new structure in the issue stage named *outstanding instruction buffer* (OIB) (see Figure 5.4). The OIB is similar in concept to the reorder buffer in an OoO processor. However, it is much smaller in size, and needs to store only 5 (pipeline depth from issue to write-back) instructions in the worst case. Each instruction entry in the OIB stores: (1) op code, (2) sources, (3) destination, (4) age tag, (5) execute stage allocation (execute stage where the instruction was steered), and (6) pending replay bits (one per source operand). The OIB behaves as a CAM for its second field (instruction source). Pending replay bit for a source operand denotes whether it can cause a data flow

violation. Instructions are inserted into the OIB at the time they are issued. At the time of an instruction write-back, following actions take place:

- The destination value ($R_{dest}$) of the instruction writing-back ($I_{wb}$) updates the register file. The corresponding OIB entry for $I_{wb}$ is also freed.
- $R_{dest}$ is used to do a CAM look-up in the OIB. This returns any in-flight instruction ($I_{in\_flight}$) that uses $R_{dest}$ as a source.
- If $I_{in\_flight}$ was sent to the same execute stage where $I_{wb}$ executed, then the bypass cache would have successfully forwarded the register value. The pending replay bit is reset (to 0) for this source operand of $I_{in\_flight}$.
- If $I_{in\_flight}$ was sent to some other execute stage, then a data flow violation is possible and the pending replay bit for this source is set (to 1).

Over time, the replay bit for a source operand can get set/reset multiple times, with the final write to it made by the closest producer operation for every consumer. If an issue stage receives a write-back for an instruction with a pending replay bit set for any of its source operands, it implies that the producer of value(s) for this instruction has executed on an execute stage different from where this instruction was steered. And, therefore, a data flow violation has occurred. A replay is initiated at this point (replay mechanism is discussed later in this section).

#### 5.3.3.3 Memory Data Flow

To provide correct memory ordering behavior in a conjoint pipelines processor, we use a local store queue in the issue stages that monitors load operations performing write-back for store-to-load forwarding violations, and a speculative store buffer in the execute/memory

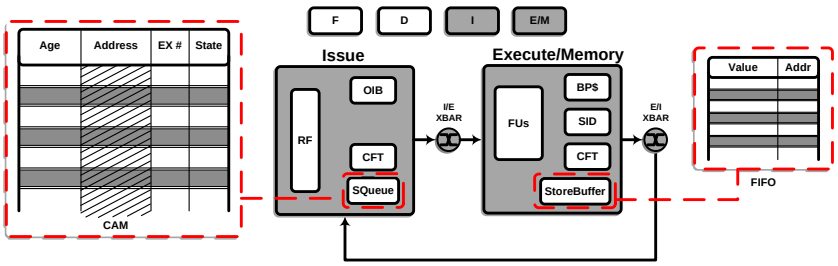



**Figure 5.5: CG pipeline back-end with an emphasis on structures added for handling memory data flow violations.**

stage to allow delayed release of memory store operations (to save against accidental memory corruption). Note that cache hierarchy is left unmodified in CG. L1 caches are private to the pipelines, single ported and naturally kept coherent by standard cache coherence protocols.

Figure 5.5 shows the back-end of a CG pipeline with an emphasis on structures needed for proper memory handling. A store buffer (*StoreBuffer*) is added to the execute/memory stage to hold onto the store values before they are released to the memory hierarchy. Although a common structure in many processors, in CG, the store buffer also serves the purpose of keeping speculative stores from corrupting memory state. A store queue (*SQueue*) is added to the issue stages to tabulate the outstanding store instructions, and their present states. Every store instruction can have two possible states. All issued store instructions are entered into the local store queue and get into the *store sent* state. Write-back for this store instruction confirms that it is not on an incorrect execution path. At this point, a *pseudo commit* signal is sent (over the same crossbar switch) to the execute/memory stage that executed this store, and the store instruction state becomes *pseudo commit sent*. Upon receiving this signal, the execute releases the store value at the head of the store buffer to the memory. This way, only stores on the correct path of execution update the memory.

There are three possible cases involving the memory operations that need a closer scrutiny (see Table 5.4).

**Table 5.4: Memory flow cases. Each case represents a pair of instructions that are flowing together in a 2-issue conjoint processor.**

| | Case 1 | Case 2 | Case 3 |
|---|---|---|---|
| Leader pipeline | $BR$ (mis-predicted) | $ST_1$ | $ST$ |
| Follower pipeline | $ST$ | $ST_2$ | $LD$ |

In case 1, a mis-predicted branch occurs right before a store in the program order. Since this store is already executed by the execute/memory stage, its value is entered into the store buffer. Fortunately, in accordance to the commit order, the branch operation writes back before the store. Thus, the store never gets to write-back and does not release a pseudo commit for itself. Eventually this store is removed from the store buffer when the mis-predicted branch flushes the pipeline. In case 2, the pseudo-commit is released for $ST_1$ before $ST_2$. Thus, to the memory hierarchy, the correct ordering is presented. In case 3, when the load is about to commit, both issue stages check if any of the outstanding stores conflicts with this load (using the store queues). If there is indeed such a store that precedes the load in the program order (based on age), and was sent to a different execution stage, then a replay is initiated starting from this load.

#### 5.3.3.4 Replay Mechanism

The replay mechanism adds a single bit of state in the issue and execute/memory stage called the *current flow tag* (CFT), and leverages the OIB in the issue stage for re-streaming instructions (see Figure 5.4). The CFT is a single bit (similar to the SID for branches) to identify the old (wrong) instructions from the new (replaying) instructions in the back-end.

All issued instructions are tagged with the CFT bit. The *head* and the *tail* pointers in the OIB mark the window of in-flight instructions, which are replayed in the event of a register or memory data flow violation. The violation is first identified by any one issue stage, which consequently sends out a *flush* instruction to both execute stages. This flips the CFT bit, resets the bypass cache and clears the store buffer. Following this, other issue stages are sent replay signals, and all of them start re-issuing instructions from their respective OIBs (starting at the head pointer) and tagged with an updated CFT bit. The old instructions, tagged with a stale CFT, are uniformly discarded by both issues during the write-back.

### 5.3.4 Interconnection

CG interconnection network is a simple, one-hop connection. It employs bufferless, non-blocking crossbars to connect adjacent levels of pipeline stages. This allows all pairs of stages, that share a crossbar, to communicate simultaneously. As an interface to the interconnection network, pipeline stages maintain a latch on both inputs and outputs. This makes the interconnection network a separate stage, and thus, it does not interfere with critical paths in the main processor stages.

In order to make the basic crossbar design suitable for the CG architecture, three features are required:

**Multicast:** The CG depends on the capability of the interconnection to send one value to multiple receivers. For instance, write-backs are sent to both issue stage register files simultaneously.

**Instruction steering:** CG requires capability to steer instructions from issue to the appropriate execute stage. A single (header) bit in the instruction payload is added to

specify the output (execute stage) an instruction wants to reach.

**Age prioritization:** In the case of write-backs, older instructions have to be given priority. This requires an addition to the router to let it prioritize packets (instructions in our case) on the basis of their age.

**Synchronized transfer:** Within CG, a pair of instructions is transferred from one level of stages to the next level synchronously. Thus, the interconnection crossbars need to wait for data to be available on both input ports, before transmitting it.

Crossbar switch fabrics with crosspoints can support multicast by setting the crosspoint gate logic to high for multiple outputs. A recently proposed SRAM based crossbar architecture, named XRAM [92], demonstrates this ability with a low power and area overhead. The instruction steering and age prioritization can be added in the wrapping logic around the crossbars. However, the XRAM paper suggests that these features can also be implemented using circuits.

**Crossbar reliability, power and timing:** In order to protect the interconnection network, fault tolerant version of the crossbars are used in CG. This is similar to the approach in [42]. The interconnection power can be broken into crossbar power and interconnection link power. Both of these are accounted for in our evaluations, as per the methodology in [119]. The absence of buffers in our network significantly cuts down on this overhead. And finally, we model interconnection link latency using intermediate pitch wire model from ITRS 2008 in 65nm technology, and make sure that it does not exceed critical paths of pipeline stages.

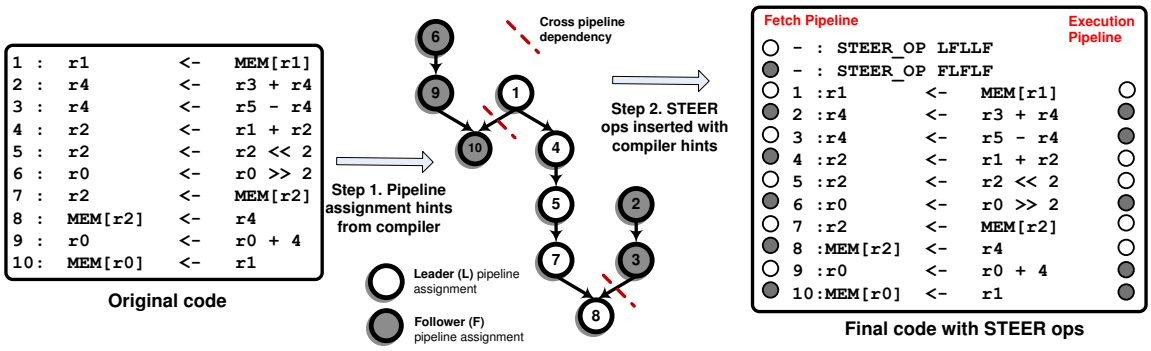



**Figure 5.6: Instruction steering. The white nodes indicate instructions assigned to the leader pipeline while the shaded nodes correspond to the follower pipeline. The instruction fetch is perfectly balanced between the two pipeline, but the execution is guided by the steering.**

### 5.3.5 Instruction Steering

CG depends upon intelligent steering of instructions between conjoint pipelines in order to minimize performance degradation from data dependency replays. The instruction steering decisions need to be made at the time of instruction issue. Broadly speaking, the objectives of instruction steering are two-fold: 1) balance the workload on the two pipelines, and 2) minimize the number of replays. Our experiments showed that using a purely hardware based solution for dynamic steering is neither cheap nor effective for in-order pipeline fusion. This concurs with the conclusion of [89]. Thus, CG adopts a hybrid software/hardware approach for instruction steering. In a nutshell, a compiler pass is used to assign instruction streams to the pipelines. These hints are then encoded into *steering* instructions that are made part of the compiled application binary. The hardware recognizes these special steering instructions and uses them to effectively conduct dynamic steering.

Steering instructions between different pipelines in a conjoint processor is analogous to data-flow graph (DFG) partitioning for clustered VLIWs. The goal is to obtain a balanced workload that takes advantage of hardware parallelism (multiple clusters) and reduces the

need for inter-cluster moves (transferring values between clusters). Leveraging generic clustering algorithms to form instruction streams for CG is fairly straightforward. When cross-pipeline dependencies cannot be avoided, the CG equivalent of an inter-cluster move is the replay mechanism described in the previous section. Further, CG's broadcast-based write-back ensures that any dependent instructions that are separated by more than $n$ intervening instructions will not incur a replay even if they are steered to different execute stages, where $n$ is the issue-to-writeback latency. Therefore, the two main objectives of clustering algorithms, minimizing inter-cluster moves and overlapping moves with other computation, naturally result in instruction streams that are amenable to the CG architecture. For our evaluations, we used the well known Bottom-Up Greedy (BUG) [33] clustering algorithm to generate hints for steering.

A *STEER_OP* instruction is introduced in order to encode this compiler-generated steering information. Two such instructions are inserted (for leader and follower pipelines) at the beginning of every instruction block (basic block / super block). *STEER_OP* instructions are simply bit encoding of the pipeline assignment for every instruction within that block (multiple instructions are inserted for large code blocks).

Figure 5.6 shows an example of the complete hybrid steering setup in action. The first step consists of performing the BUG clustering algorithm in the compiler. The second step encodes the clustering algorithm suggested pipeline assignments and embeds them as *STEER_OP* (top two instructions in the final code, 'L' here stands for leader pipeline assignment and 'F' for follower pipeline assignment). When the leader pipeline fetches its *STEER_OP LFLLF*, it learns the steering directions for instruction 1 (L), 3 (F), 5 (L), 7 (L) and 9 (F). The follower pipeline behaves analogously.

### 5.3.6 Configuration Manager

CG requires a software level configuration manager for supervising the system-wide reliability and performance configuration. The inputs to this manager are: a) list of working components (pipeline stages and interconnection crossbars) and b) profile of jobs active in the system. When invoked, the configuration manager first assesses the maximum number of logical pipelines it can salvage out of the system. Following this, it distribute the working pipelines to the active workloads. In our evaluation, we assume a simple policy where: 1) all workloads are assigned a single pipeline, 2) any remaining pipelines are allocated to the threads on the basis of ILP (instruction-level parallelism) available in the same. The configuration manager is re-invoked every time a failure occurs or the workload set changes. The frequency of the former is in the order of months, whereas that of the latter is in seconds.

### 5.3.7 Instruction Flow Example

Figure 5.7 shows a generic code snippet accompanied by a set of instruction position tables showing the execution progress in a 2-wide CG processor (at some representative clock ticks). The example presented here has three implicit assumptions: 1) one-cycle instruction transfer time over the interconnection, 2) five-cycle delay for communicating values produced in one pipeline to the other, and 3) single instructions flow in the pipeline, instead of bundles of operations to amortize network transmission costs [43]. All of these assumptions are only for the sake of the example, and actual values can differ in a real program execution. The discussion of the CG processor state at the representative clock ticks is provided below:

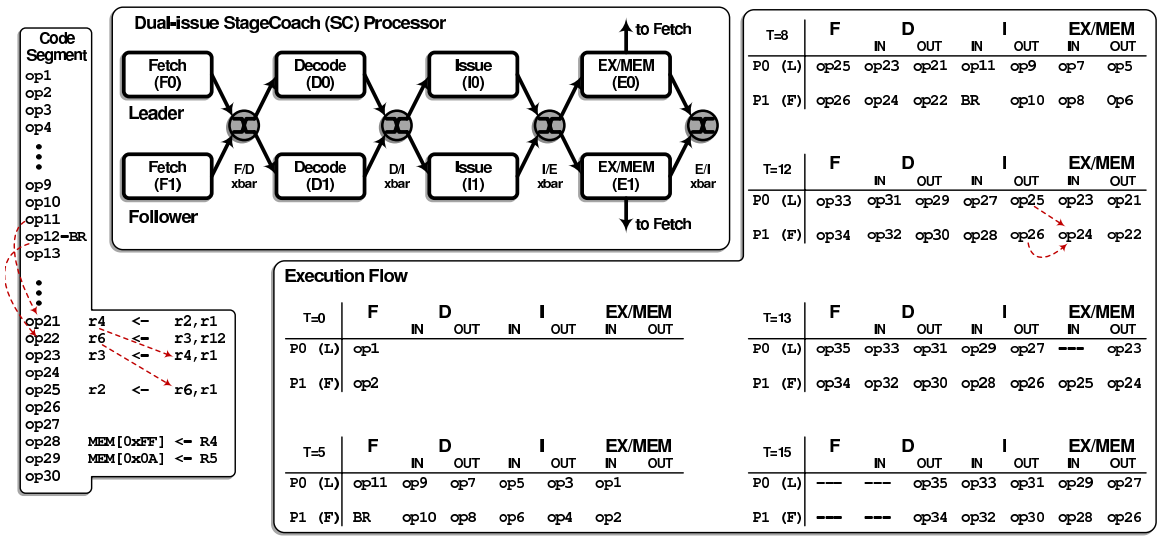



**Figure 5.7: A dual-issue CG processor executing a sample code under optimistic conditions, i.e. no control, data or memory violation occurs.**

**Time 0:** Leader and follower pipelines fetch $op1$ and $op2$ respectively. This mechanism can be easily realized in hardware by keeping the follower pipeline's PC at an offset of single instruction width.

**Time 5:** Assuming all instruction cache hits, after five cycles, both pipelines would be full. The follower pipeline (P1) gets a branch operation ($op12$) in this cycle. The branch is taken, and both pipelines jump to the target program location, while maintaining their PC offset.

**Time 8:** By this time, the processor has successfully jumped to the new location and both pipelines are filled up with more instructions ($op21$-$26$). The older instructions have been retired in-order. A register flow dependency exists between $op21 \rightarrow op23$ for $R4$ and $op22 \rightarrow op25$ for $R6$. The dependency for $R4$ is shared by operations in the same (leader) pipeline, whereas the dependency for $R6$ is cross-pipeline.

**Time 12:** $Op23$ is at the input latch of the execute block in the leader pipeline. The value for $R4$ from $op21$ gets forwarded to $op23$ through the bypass cache that exists within

each pipeline. However, the cross pipeline dependency of $op25$ for $R6$ requires it to be cross steered to pipeline P1. Thus, $op25$ and $op26$ both compete for P1's execute stage.

**Time 13:** $Op25$ is cross steered to the follower pipeline, and is thus successfully able to obtain the $R6$ value forwarded from $op22$. No operation is issued to the leader pipeline during this cycle, and a pipeline bubble is introduced.

**Time 15:** $Op29$ and $op28$ (store operations) are issued to the leader and the follower pipelines, respectively. The store addresses for these two operations do not conflict.

The above example is a collection of fortunate cases where the CG architecture operates at full dual-issue bandwidth without running into any correctness issues or performance bottlenecks. In reality, the decentralized nature of the two pipelines, and consequently the local decision made by each of them could lead to several instruction replays.

## 5.4 Evaluation

### 5.4.1 Methodology

A comprehensive set of tools are used for the evaluation of CG. The evaluation setup spans program compilation and microarchitecture level simulation, down to area, power and wearout modeling.

**Compilation for instruction steering.** The Trimaran compilation system [111] is used to perform the BUG clustering algorithm [33] for instruction steering. Inter-cluster move latency of five cycles is used as an input to the algorithm.

**Table 5.5: Architectural parameters.**

| Baseline architecture | |
|---|---|
| Pipeline | 4-stage in-order OR1200 RISC [76] |
| Frequency | 400 MHz |
| Area | $0.71mm^2$ (65nm process) |
| Power (baseline OR1200 core) | $94mW$ |
| Branch predictor | Global, 16-bit history, gshare predictor, BTB size - 2KB |
| L1 I$, D$ | 4-way, 16 KB, 1 cycle hit latency |
| L2 $ | 8-way, 64 KB (per core), 5 cycles |
| Memory | 40 cycle hit latency |
| **CG specific parameters** | |
| Interconnection | full non-blocking crossbars, 64-bit wide, bufferless |
| Outstanding instruction buffer (OIB) | 5 entries |
| Store queue, store buffer sizes | 3, 3 |
| Bypass cache size | 6 |

**Microarchitectural simulation.** The microarchitectural simulator for CG models a group of 4-stage in-order pipelines (similar to the OR1200 core [76]) interconnected to form a network of stages. The simulator was developed using the Liberty Simulation Environment [113] from Princeton. The architectural attributes are detailed in Table 5.5. The L2 cache is unified and its size is $64\ KB \times$ *the number of cores*. The original OR1200 pipeline is also used as the baseline for single-thread performance. The architectural simulations are conducted for benchmarks chosen from three sources: SPEC2000int, SPEC2000fp and multimedia kernels.

**Area overhead (for design blocks and wires).** Industry standard CAD tools with a library characterized for a 65nm process are used for estimating the area of design blocks. A Verilog description for the OR1200 microprocessor was obtained from [76]. Most CG modifications: OIB, SQ, SB, bypass cache, etc., are essentially small memory structures, and their areas are estimated using similar sized CAM structures. All non-memory structures, such as replay logic, stream identification control, and crossbars, are implemented as Verilog modules to obtain accurate area numbers. The area for the interconnection wires

between stages and crossbars is estimated using the same methodology as in [64, 52], with the intermediate wiring-pitch (at 65nm) taken from the ITRS road map [53].

**Power and thermal modeling.** Power dissipation for various modules in the design is simulated using Synopsys Primepower an execution trace of OR1200 running media kernels. The crossbar power dissipation was simulated separately using a representative activity trace. The crossbar Verilog was placed and routed using Cadence Encounter before running it through Primepower. The stage to crossbar interconnection power was calculated using standard power equations [119] with capacitance from Predictive Technology Model [85] and intermediate wiring-pitch from 65nm node (ITRS [53]). The thermal modeling was conducted using HotSpot 3.0 [47].

**Wearout modeling.** For wearout modeling, mean-time-to-failure (MTTF) was calculated for various components in the system using the empirical models found in [101]. An entire core was qualified to have an MTTF of 10 years. The calculated MTTFs are used as the mean of the Weibull distributions for generating times to failure (TTF) for every module (stage/crossbar) in the system. For the sake of consistency in comparisons, wearout modeling makes assumptions similar to those in [43].

The MTTF of 10 years was chosen as a rough estimate for the future technologies: 22nm and beyond. Note that the 65nm technology node was only used to get power and area overheads for comparisons.

**Quantitative comparison against other schemes.** For experiments involving multicores, CG is compared against two other systems: 1) A conventional CMP chip where, a core is considered to be faulty when any of its modules fail; 2) a SN chip [43], as it shares

similarities with CG in the way it tackles reliability.

### 5.4.2 Single-thread performance

Configurable performance in CG relies on its ability to accelerate single-thread performance by conjoining *in-order* pipelines. Figure 5.8 shows a plot comparing the performance of four CG configurations normalized to the 1-issue in-order baseline (OR1200). The plot also includes a 2-issue in-order baseline for the sake of comparisons. The CG configurations are expressed as:

number_of_pipelines_conjoint $\times$ issue_width_of_pipeline_stages.

The following configurations are examined: 1-issue (1x1) CG single pipeline, 2-issue (2x1) CG conjoint pipelines, 2-issue (1x2) CG single pipeline, and 4-issue (2x2) CG conjoint pipelines.

**Conjoining single-issue stages.** This compares a (1x1) CG pipeline and a (2x1) CG conjoint pipeline against the two baselines. All pipeline stages are inherently single-issue in this set up. The 1-issue CG pipeline (1x1) performs roughly 10% worse than the 1-issue baseline, primarily due to the inter-stage transfer inefficiencies from decoupling (this is similar to results in the SN work [43]). On the other hand, the (2x1) CG conjoint pipeline was found to deliver a consistent performance advantage over the 1-issue baseline, while lagging behind the 2-issue baseline. The gains are most prominent for the SPECfp and kernel benchmarks. In fact, for some of the kernel benchmarks, almost a 2X performance gain was seen while using the conjoint processor. The availability of long and independent data dependence chains in these benchmarks made this result possible.

In contrast, a few of the benchmarks showed negligible to negative performance im-

provements while using the conjoint processor, namely 176.gcc, 197.parser and 177.mesa. This was due to the lack of independent streams of instructions in these workloads. Instructions in these benchmarks typically formed long dependence chains, and the compiler pass (for steering) ended up allocating most instructions to the same pipeline (to minimize the replay cost). This resulted in a nearly complete serialization of the program, rendering half of the execution resources useless. The few cases where instructions were steered to different pipelines lead to data flow violations and worsened the overall performance by initiating the replay. Barring these three benchmarks, the rest of the results strongly favor the conjoint pipelines CG processor design. On average, a 48% IPC gain is seen over the single pipeline CG processor.

**Conjoining dual-issue stages.** This compares a (1x2) CG single pipeline processor, a (2x2) CG conjoint pipeline processor and a 2-issue baseline processor. All pipeline stages are inherently dual-issue in this set up. A single logical pipeline in this system would behave as a dual-issue processor. Using the CG conjoining principles, any two dual-issue pipelines can be then combined to form a quad-issue CG processor. The (2x2) 4-issue conjoint pipeline shows a 35% improvement in performance over the (1x2) 2-issue pipeline, and a 25% improvement in over the 2-issue baseline. Note that by making the pipeline stages dual-issue, the fault isolation granularity for the system is reduced by half. This discussion is continued later in this section along with the reliability implications.

Our experiments with conjoining more than two pipelines (both single and dual-issue) at a time did not show very favorable results. The two main reasons for this were: 1) the limited availability of independent data-flow chains, and 2) the constraints placed by an

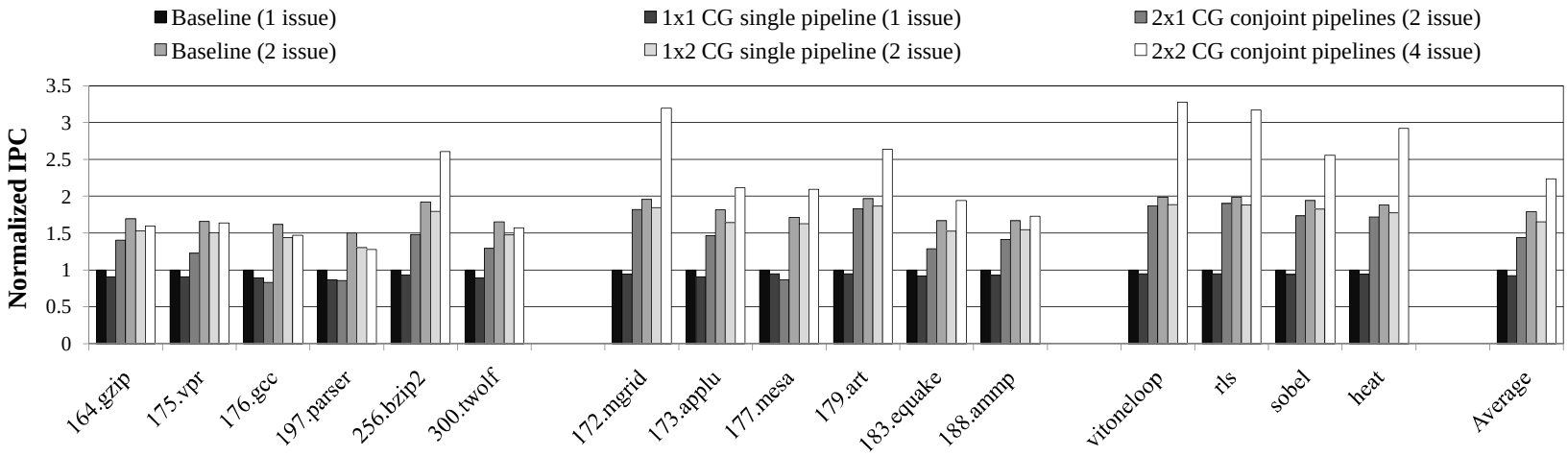



**Figure 5.8: Single thread performance results for CG normalized to a single-issue in-order processor. The configurations are expressed as (number_of_pipelines_conjoint X issue_width_of_pipeline_stages).**

in-order issue architecture.

**Replay costs.** The performance advantage of a conjoint CG processor is largely determined by the efficiency of instruction steering in balancing the load between the two pipelines, while minimizing replays. Here, we analyze the cost of these replays in a 2-issue (2x1) CG conjoint pipelines processor. Figure 5.9 shows three components of the total execution time for all the benchmarks: memory flow (MemFlow) violation replay cycles, register flow (RegFlow) violation replay cycles and normal operation cycles. A majority of the benchmarks devote a small fraction of their execution time to the replay cycles, with an average of 15%. Out of the total replay cycles, memory replay contributes a negligible fraction. This is an expected result because memory replay only happens when a store to load forwarding is missed by the system, which by itself is a rare event for in-order processors. From the perspective of power efficiency, these results are encouraging because only a very small percentage of the work performed by the system goes to waste. Note that a low number of replay cycles does not necessarily imply good benchmark performance. For instance, all instructions in a conjoint processor can be steered to the same pipeline resulting in zero replays (no cross-pipeline dependency). However, no speedup compared to the baseline would be observed.

### 5.4.3 Energy-efficiency Comparison

Energy-efficiency of designs can be compared using $BIPS^3/watt$ as a metric [20]. This metric is more sensitive to performance changes, and optimizing for it yields the same results as optimizing for $ED^2$ (energy times delay squared). Figure 5.10 shows the average IPC and $BIPS^3/watt$ comparisons for the four CG configurations normalized to

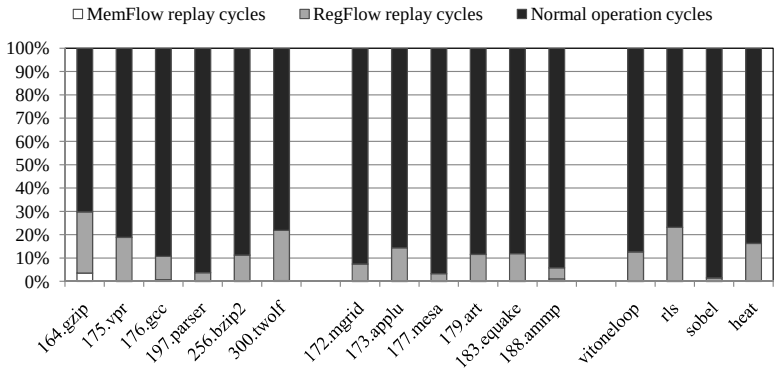



**Figure 5.9: Contribution of memory replay cycles, register flow replay cycles and normal operation cycles to the total computational time of individual benchmarks running on a 2-issue conjoint processor. On an average, the replays contributed to about 15% of the execution time.**

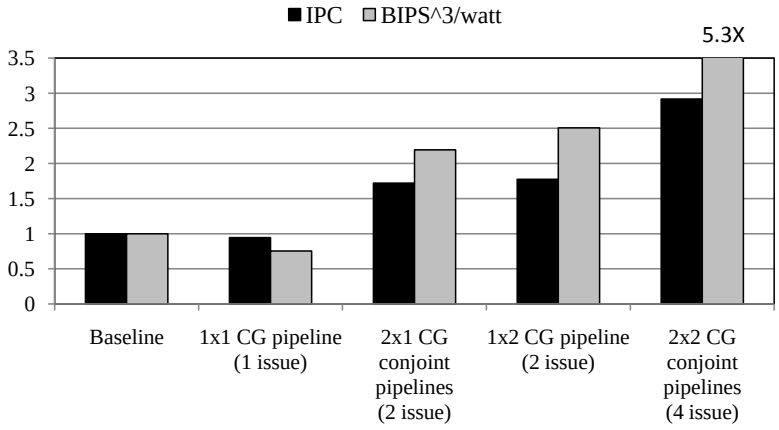



**Figure 5.10: Comparing IPC and energy efficiency ($BIPS^3/watt$). The baseline is a single-issue in-order core (OR1200).**

the single-issue baseline processor. When going from the baseline to the single-issue CG pipeline, about 20% energy efficiency is sacrificed. However, the superior performance in wider-issue configurations, significantly improves CG's energy efficiency.

### 5.4.4 Multi-workload throughput

Performance of a CMP system can be measured either as the latency of thread execution (single-thread performance, prior experiment) or the rate at which jobs complete (system throughput). For the throughput comparison, three systems were compared against one

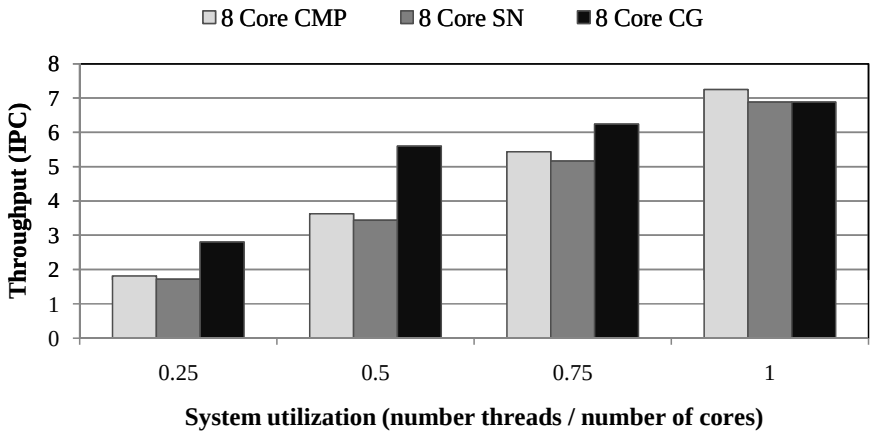


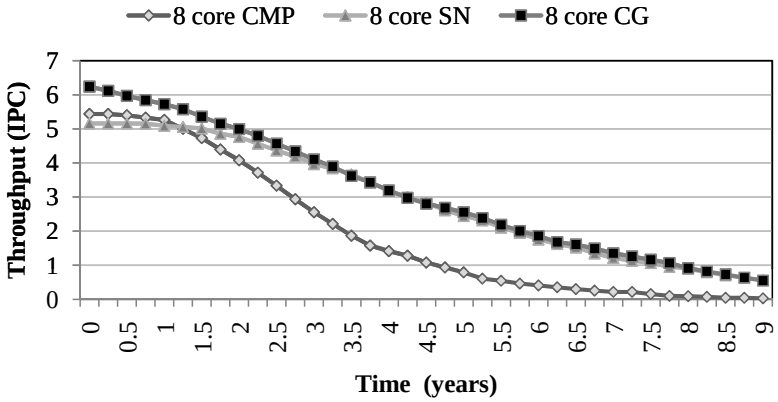

**Figure 5.11: Throughput comparison of 8-core CMP, SN and CG systems at different levels of system utilization. A utilization of 0.5 implies that 4 working threads are assigned to the 8-core system. At this utilization, CG multicore delivers 46% throughput advantage over the baseline CMP.**

another: an 8-core CMP, an 8-core SN [43] and an 8-core CG. A core here refers to a single issue in-order pipeline resource, thus an 8-core SN and CG would have eight pipelines interconnected. The system utilization was varied from 0.25 occupancy to 1.0 occupancy. This refers to the number of threads assigned to the system versus its capacity (measured as number of cores). Monte-Carlo experiments were conducted by varying the set of threads allocated to the system at each utilization level. Figure 5.11 shows the final throughput results from this experiment. At the peak utilization level (1.0), the 8-core CMP delivers the best throughput. This is due to the performance advantage the baseline processor has over both single pipeline SN and CG processors (see single-thread performance results above). Further, the throughput of SN and CG are identical because CG defaults to using one pipeline per thread in this peak utilization scenario. As the system utilization is lowered, CG is able to leverage the idle pipeline resources to form conjoint processors. Thus, the CG system consistently delivers the best throughput at all utilization levels $< 1$, which is a realistic expectation for over-provisioned systems.

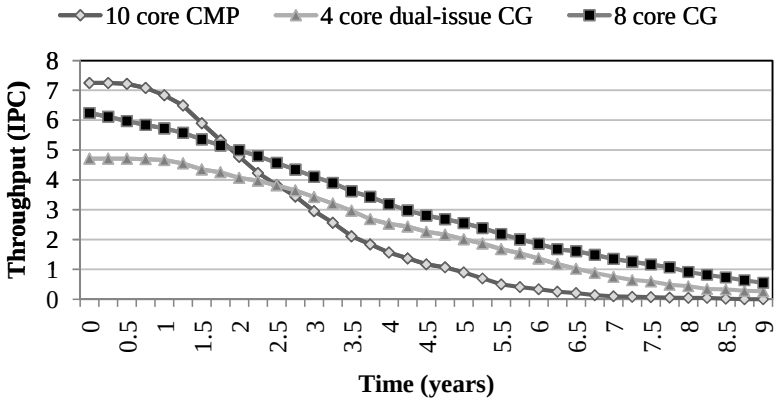
### 5.4.5 Fault tolerance

The experiments so far have targeted the performance aspect of the CG architecture. In order to evaluate its reliability in the face of wearout failures, we conducted some experiments that track the throughput of the system over the course of its lifetime. For these experiments, the stages/crossbars fail as they reach their respective time-to-failures (TTFs). The system gets reconfigured over its lifetime whenever a failure is introduced. Broken stages are isolated using interconnection flexibility, and fault tolerant crossbars naturally handle crosspoint failures. A software configuration manager is re-invoked every time a failure occurs or the workload set changes. We assume a simple reconfiguration policy where: 1) all workloads are assigned a single pipeline, 2) any remaining pipelines are allocated to the threads on the basis of available ILP. The throughput of the system is computed for each new configuration based on the number of working logical pipelines and the workloads assigned to them. Monte-Carlo simulations are run for 1000 chips to get statistically significant results. The average system utilization for these experiments is kept at $0.75$. Since the throughput delivered by the CG system improves as the system utilization is lowered (see Figure 5.11), the CG results reported here are conservative.

Figure 5.12(a) shows the throughput over the lifetime for three systems: an 8-core CMP, an 8-core SN and an 8-core CG. CG clearly outperforms both of the other systems for the entire lifetime. Early on, CG achieves a throughput advantage by utilizing the idle pipelines (only 6 threads are active, leaving 2 pipelines free) to form conjoint processors. The regular CMP and SN systems cannot benefit from this. Later in the lifetime, CG sustains a throughput advantage over the CMP by effectively salvaging the working stages and main-




(a) Throughput over the lifetime of 8-core CMP, SN and CG at a fixed utilization of 0.75. After a few initial years, CG's throughput settles down to that of a SN system.




(b) Throughput over the lifetime of 10-core CMP, 4-core dual-issue CG and 8-core CG at a fixed utilization of 0.75. CG system shows the most convincing results among all the three configurations considered.

**Figure 5.12: Lifetime reliability experiments for the various CMP, SN and CG systems. Only wearout failures were considered for this experiment.**

taining a higher number of working pipelines. For instance, the CMP system's throughput drops below 2 $IPC$ around the $3.5$ year mark, whereas the CG system throughput breaches that level around the $6$ year mark. The gains add up over the lifetime, and *cumulative work done* (integral of throughput over the lifetime) advantage of CG is $68\%$ over the baseline CMP. Also note that CG's throughput converges with that of SN in the later part of the lifetime. This happens when the number of threads assigned to the system exceeds the number of working pipelines, and CG is left with no option but to default back to single pipeline processors.

Figure 5.12(b) compares two more system configurations to the 8-core CG: a 10-core CMP and a 4-core dual-issue CG. The 10-core CMP is chosen to have an area-neutral comparison with the CG system. The area overhead for CG is about 20% (discussed later), translating to roughly two cores for an 8-core CG system. The results show that early in the lifetime, the 10-core CMP dominates the other two configurations. This is expected as it starts off with the maximum amount of resources. However, as the failures accumulate, it quickly loses its advantage. Beyond the two year mark, the 8-core CG consistently dominates the system throughput. The 4-core dual-issue CG system performs the worst among the three. There are two reasons for this: 1) it can run fewer threads concurrently (4-cores instead of 8/10), and 2) failures in stages result in a bigger resource loss (as each stage is dual-issue).

### 5.4.6 Area overheads

The area for various structures that are part of the CG architecture is shown in Table 5.6. The overhead percentages are relative to our baseline processor - the OR1200 core. A total of five interconnection crossbars are present in the CG architecture, but since the pipelines share crossbars, its overhead is not attributable to just one pipeline. For a case where eight pipelines are connected together to form CG, each of them bears $5/8^{th}$ of the crossbar overhead. With this assumption, the total area overhead for the CG architecture is 19.6% over a traditional CMP (containing OR1200 cores).

### 5.4.7 Power overheads

The power overhead in CG comes from three sources: crossbars, stage/crossbar interconnection and miscellaneous logic (extra latches, new modules). Table 5.7 shows the

**Table 5.6: Area overheads from different design blocks in CG.**

| Design block | Area ($mm^2$) | Percent overhead |
|---|---|---|
| Outstanding instructions buffer (OIB) (5 entries) | 0.037 | 5.7% |
| Store buffer (SB) (3 entries) | 0.015 | 2.3% |
| Store queue (SQ) (3 entries) | 0.021 | 3.4% |
| Bypass cache (6 entries) | 0.02 | 3.1% |
| Extra stage latches (input and output) | 0.0115 | 1.8% |
| Miscellaneous logic | 0.055 | 0.9% |
| 8x8 fault tolerant crossbar (with interconnection wires) five such crossbars are shared between eight pipelines | 0.025 | 3.9% |
| Total area overhead of CG | | **19.6%** |

breakdown, with total power overhead at $16.9\%$. The actual power numbers in the table are overheads for one CG pipeline while it is part of a 2-issue CG conjoint processor. Note that a part of this overhead will be there even in a traditional 2-way superscalar (relative to having 2 independent 1-way pipelines).

**Table 5.7: Power overhead for CG. These overheads are reported with OR1200 power consumption as the baseline.**

| Component | Power overhead pipeline ($mW$) | Percent overhead Percent overhead |
|---|---|---|
| Crossbars | 4.0 | 4.26% |
| Interconnection links | 5.8 | 6.19% |
| Other design blocks | 6.1 | 6.38% |
| Total power overhead | | **16.9%** |

## 5.5 Summary

In the multicore era, where on one hand abundant throughput capabilities are being incorporated on die, single-thread performance and power efficiency challenges still confront the designers. Further, the increasing process variation and thermal densities are stressing the limits of CMOS scaling. To efficiently address all these solutions, designers can no longer rely on an evolutionary design process. Further, simply combining existing research solutions for performance and reliability is neither easy nor cost-effective. In this chapter, we presented CoreGenesis, a highly adaptive multiprocessor fabric that was designed

with performance and reliability targets from the ground up. The interconnection flexibility within CoreGenesis not only ensures impressive fault-tolerance, but coupled with the addition of decentralized instruction flow management, it can also merge pipeline resources to accommodate dynamically changing application requirements. Our experiments demonstrate that merging of two pipelines within CoreGenesis can deliver on average 1.5X IPC gain with respect to a standalone pipeline. In a CMP, with only half of its cores occupied, this merging can enhance throughput performance by 46%. Finally, the lifetime reliability experiments show that an 8-core CoreGenesis chip increases the cumulative work done by 68% over a traditional 8-core CMP.

# CHAPTER VI

# Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing

## 6.1 Introduction

The traditional microprocessor was designed with an objective of running general purpose programs at a good performance, while treating the efficiency as a second order criteria. However, with the growing demand for higher performance and efficiency in modern day devices, there is an emerging need for architecture level solutions to tackle computational energy efficiency. The trend in the silicon integration is also reinforcing this need for energy-efficient architectures. Over the years, transistor densities and performance has continued to increase as per Moore's Law, however, the threshold voltage has not kept up with this trend. As a result, the per-transistor switching power has not witnessed the benefits of scaling, causing a steady rise in power density. Overall, this limits the number of resources that can be kept active on a die simultaneously [114]. An instance of this trend can be already seen in Intel's newest Nehalem generation of processors that boost the performance of one core, at the cost of slowing down/shutting off the rest of them.

Long before designers of server farms and desktop machines started caring about energy-efficiency, it has been actively pursued by embedded system designers. The concerns such

as a longer battery life and tolerable heat dissipation have pushed the embedded architectures to take extreme steps for saving energy. In this domain, a common practice has been to design specialized hardware units [77, 80, 93], accelerators [35, 122], and application specific instruction extensions [107] to save energy for their stable application set (various kernels for audio/video processing, compression, signal processing, etc). The computation in these applications is regularly structured, highly data parallel, and concentrated in tight inner-most loops, making it well suited to hardware specialization.

Unfortunately, this hardware specialization approach does not directly extend to programs such as desktop applications, SPEC integer suite, OS utilities, etc., for two primary reasons. First, these programs are highly irregular, contain a lot of control divergence and exhibit little data parallelism. Henceforth, we refer to these as *irregular codes*. The characteristics of irregular codes make them unsuitable for traditional accelerator designs. For instance, the large, unstructured, uncounted loops in these applications can not be mapped to the loop accelerators [35, 25] which can only support modulo-schedulable loops. Second, the general purpose application space is much more diverse and constantly evolving. Designing a custom hardware for each of these programs is neither cost-effective nor practical. Despite these challenges, a recent work [114] makes a case for application specific hardware in the context of irregular codes, claiming the large availability of dark silicon. However, once the targeted applications gets modified beyond a certain degree, the approach reverts back to software emulation on the main processor pipeline losing the efficiency benefits of the ASIC (application specific integrated circuit).

Another class of solutions to target irregular codes is the work on programmable functional units [24, 87]. However, their energy efficiency gains are small due to their program

scope (acyclic chain of operations), and architectural focus (processor back-end). Studies have shown that a large fraction of application energy is consumed by the processor front-end (fetch and decode) [31]. Thus, given the state of the current art, no clear path exists to design a reasonable energy-efficient architecture that can support irregular codes while also offering a flexibility to work across applications.

As a solution to energy-efficiency problem in this general purpose processing domain, this chapter proposes Green BERET (Bundled Execution of REcurring Traces). The BERET architecture is a configurable compute engine that achieves significant energy savings for the program regions mapped onto it, without sacrificing any performance. The first insight of this architecture is the use of *recurring traces* as a program construct for tackling irregular codes. A recurring trace [79, 40, 70] is a sequence of program instructions that repeatedly execute back-to-back with a high likelihood, despite the presence of intervening control divergences. As their first advantage, these traces give an appearance of structure to the irregular codes. Further, as these traces are significantly shorter than the original unstructured loops, BERET buffers them internally to eliminate redundant fetches and decodes for repeating instructions.

The second insight of this work is the use of *bundled execution* for these traces. Instead of executing one instruction at a time, BERET uses compiler analysis to break down traces into bundles of instructions. These bundles are essentially subgraphs from the trace-wide data flow graph. A major advantage of this bundled execution is that it significantly cuts down on the redundant register reads and writes for the temporary variables. Further, our analysis of application traces demonstrated that many subgraph structures are common across applications. Thus, given a diverse enough collection of subgraph execution blocks,

our compilation scheme is able to break down an application trace into constituent subgraphs from this collection. Overall, we consider this bundled execution model a trade-off design that lets us achieve efficiency gains close to an application specific data flow hardware while maintain application universality of regular Von Neumann execution model.

Leveraging these two insights, the BERET is designed as a subgraph-level compute engine for recurring traces. For the program traces offloaded to BERET, the energy savings primarily come from a) eliminating redundant fetches, decodes, and control management, and b) significantly reducing register reads and writes for temporary variables. The key contributions of this chapter can be summarized as follows:

1. A programmable compute engine for energy-efficient general purpose processing
2. Insight to exploit recurring instruction sequences (traces) as a means to tackle irregular codes
3. Compiler flow to map arbitrary program traces on a heterogeneous collection of subgraph execution blocks

## 6.2 A Case for Energy Efficient Trace Execution

In this section, we investigate the sources of inefficiency in a simple in-order RISC processor core, explore opportunities for energy savings, and propose our insights on designing a general purpose, energy-efficient compute engine. For a detailed comparison of our work to prior schemes, please refer to Section 6.5 and Table 6.1.

### 6.2.1 Pipeline Energy Distribution

In a conventional Von Neumann architecture, the processor spends a large amount of effort in supplying instructions and data values to the actual execution units [31]. For a

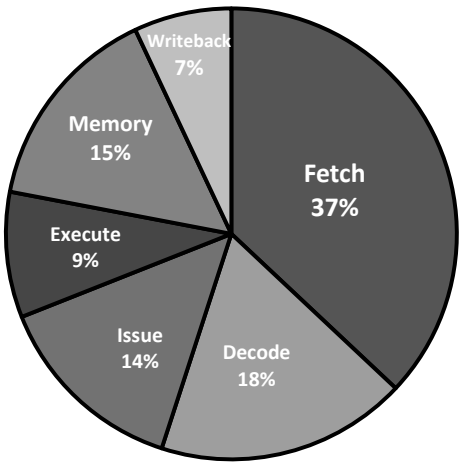



**Figure 6.1: The distribution of energy dissipation across pipeline stages in an in-order processor.**

better understanding of this behavior, we analyzed the per-stage energy distribution in a simple in-order RISC processor (modeled after an ARM core). Figure 1 shows this result, highlighting the large majority of energy dissipation attributable to the instruction supply (Fetch and Decode). The major component behind this was the instruction cache, which is not only a large structure, but needs to be accessed for every single dynamic instruction in a program. The second biggest energy draw was from the combined register read (Issue) and write back (Writeback) cost. This is representative of the data supply cost, along with the datapath memory access (Memory). The last stage in this tally, surprisingly enough, is the data computation (Execute). Once the instructions and data are delivered to an execution unit, only a small amount of energy is required to compute the result.

This analysis clearly highlights that a regular in-order pipeline has a severe imbalance in terms of where the energy is being spent. For a small fraction of compute energy, almost 8X more energy is taken up to deliver the instruction and data to the execute stage. On a positive note, this also indicates that methods targeting instruction and data supply energy

can achieve substantial savings.

### 6.2.2 Opportunities for Energy Saving

A significant source of this biased energy consumption is the lack of understanding a general purpose processor has for the underlying program structure. The hardware is typically agnostic of the presence of loops, live data values, data flow between instructions, chains of frequently occurring operations, and so on. This results in wasted effort for redundant instruction fetches and decodes (for repeating sequences such as loops), redundant register file reads and writes (for temporary / intermediate values), redundant forwarding and dependency checks for unrelated instructions, etc. Each of these redundant actions present an opportunity for energy savings.

A popular approach for reducing this wasted effort has been to introduce hardware specialization, in the form of ASICs [77, 80, 93], loop accelerators [35, 122], custom functional units, etc. The attempt here is to encode the program structure in the hardware, such that it can avoid wasted effort during execution. For instance, loop accelerators buffer the instructions in a loop, thereby avoiding the redundant instruction cache accesses [35]. The hardware specialization solutions work particularly well for applications that have regular structure, data parallel computations, and limited control divergence. Prime examples of this are media kernels, encoders, compression engines, image processing, etc. Henceforth, we refer to such applications as *regular codes*.

### 6.2.3 Limitations for Irregular Codes

In addition to regular codes, energy-efficiency is equally important for applications in desktop computing, SPEC integer suite, OS utilities, libraries, etc. Unfortunately, the

concept of hardware specialization, does not scale to this application class because:

1. **Large and irregular loops:** The programs are highly irregular and contain a lot of control divergence. More specifically, the loops are usually large, uncounted (while loops), and contain deeply nested if-then-else statements. These characteristics are unfavorable for a specialized hardware design because: a) an ASIC designed for this will be very large (due to code divergence / loop size), and have a very low utilization (only single execution path would be taken); b) on the other hand, loop accelerators would fail to work as they can only handle modulo-schedulable loops.
2. **Too many applications that are also regularly modified:** Even if one could somehow design ASICs for these *irregular codes*, a large number of such ASICs will be required to keep up with the application diversity and code modifications. This is unlike embedded systems that have a limited number of relatively stable, well structured kernels.

### 6.2.4 Energy Efficiency for Irregular Codes

Due to the aforementioned reasons, achieving general purpose, energy-efficiency for irregular codes has long remained a tough challenge. In this work, we build upon two insights for solving this problem:

1. **Structuring the Irregular Code using Traces:** Often times, the dynamic behavior of irregular codes exhibits a regular structure. In the literature, this regular structure has been referred to as *traces* [40], *frames* [79] and *superblocks* [70] (in compilers). Traces are defined as sequences of instructions that have a high likelihood of executing back to back, despite the presence of intervening control divergences. These can be identified both statically and dynamically, covering roughly 70% of dynamic instructions [79].

   In the scope of this work, we focus on a subset of traces that also loop around with a

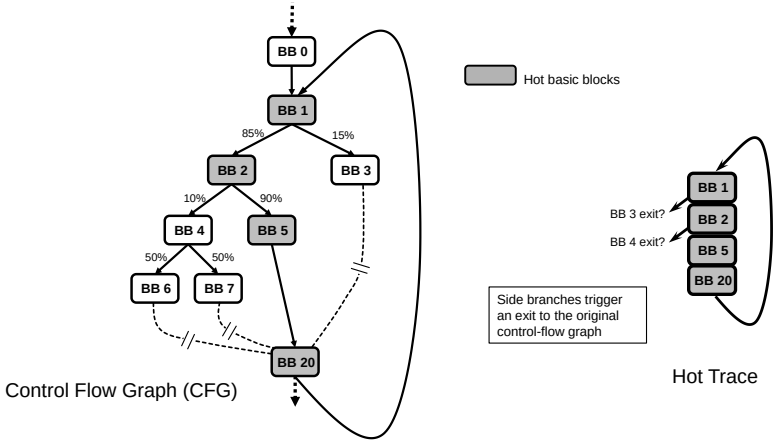



**Figure 6.2: Extracting a looped trace from an irregular control flow graph. We refer to these as *hot traces*, and use them as a construct that runs on our energy-efficient hardware design.**

high probability, and refer to them as *hot traces*. Figure 6.2 shows an example of an irregular CFG, with the extracted hot trace. These hot traces not only render a regular structure to the CFG, but in addition, their looping nature is favorable to instruction supply energy savings.

2. **Generalizing Across Applications:** Working with hot traces eliminates the differences due to control flow between application codes, leaving behind only data flow variations. Further, we observed that hot traces can be segmented into small data flow subgraphs, many of which are common across applications. Consequently, as we demonstrate later, given a diverse enough collection of subgraph execution units, a compilation scheme can be formulated to break down a trace into constituent subgraphs from this collection. The use of subgraph-based computation is also favorable for data supply energy savings.

The next section uses these two insights to design a general purpose, energy-efficient trace execution engine.

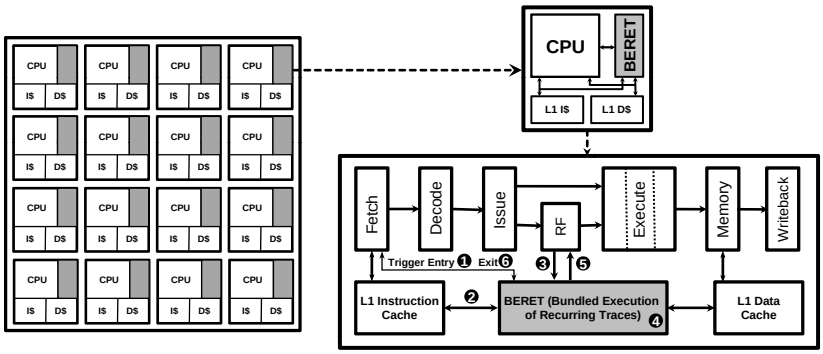



**Figure 6.3: Deployment of BERET at multicore level and its integration within a single processor core.**

## 6.3 The BERET Architecture

### 6.3.1 Overview

The proposed design, named BERET, is a configurable co-processor optimized for energy-efficient execution of hot traces from a program flow. These hot traces are short, logically *atomic*, single-entry, single-exit program regions with a high probability to loop back. Further, the BERET hardware executes these traces in bundles of instructions rather than individual instructions. One can think of these instruction bundles as data flow subgraphs from the trace. As a result of these high level design choices, several avenues of energy savings follow. First, the short program traces are stored inside BERET hardware, this eliminates redundant instruction fetches as traces loop around. Second, the instruction bundles from traces are encoded as BERET microcode, eliminating the need for decode. Third, use of instruction bundles helps in reducing unnecessary storage and retrieval of temporary values. And finally, the simplified design due to small storage structures, fewer pipeline latches, no control flow, also contributes towards energy savings.

Conceptually, every core in a system can be augmented with an instance of BERET execution engine. Figure 6.3 shows this setup, and integration of BERET within a pipeline

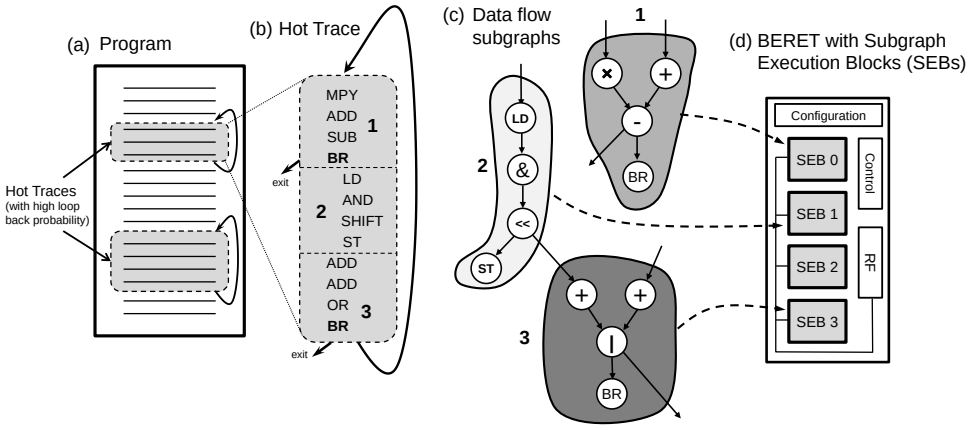



**Figure 6.4: The process of mapping hot traces in a program to the BERET hardware: (a) shows a program segment with two hot traces, (b) a closer look at a trace with instructions and two side exits, (c) illustrates the break-up of trace code into data flow subgraphs, and (d) mapping of subgraphs to subgraph execution blocks (SEBs) inside the BERET hardware.**

while sharing the same cache hierarchy. During a program's execution, whenever a (statically marked) hot trace is encountered, the fetch stage transfers control to the BERET hardware (Step 1 in Figure 6.3). BERET loads the configuration corresponding to this trace from the instruction cache (Step 2), and reads register live-ins for this region of code (Step 3). At this point, the execution control has successfully transferred to BERET and it acts as an independent entity (Step 4). Internally, BERET executes the trace at the granularity of data-flow subgraphs, and repeats the sequence until a trace exit is flagged. More discussion about the BERET microarchitecture, challenges for trace exits, and corresponding solutions, follow in Section 6.3.2. Once a trace exit is identified, the live-outs from this execution are written back to the pipeline register file (Step 5). And finally, a trigger is sent to the pipeline fetch stage, to start the regular program execution (Step 6).

Utilizing the BERET hardware involves identifying hot traces in a program's execution, and appropriately mapping them to the underlying BERET execution engine. Figure 6.4 shows a high level view of this process. The first step is to identify hot traces (Figure 6.4(a))

from the program execution that are good candidates for using BERET. The selected traces are frequently occurring sequence of program instructions that loop around, and rarely take a side exit. For every such hot trace, the instruction sequence is broken down into data flow subgraphs (Figure 6.4(b,c)). The subgraphs, if desired, can span across control instructions within a trace. In fact, the larger window of instructions visible in a trace supports this notion, and helps in identifying longer chains of connected operations. Finally, these subgraphs are mapped onto a heterogeneous set of subgraph execution blocks (SEBs) within the BERET hardware (Figure 6.4(d)).

In the above discussion, the latter few steps of dividing up a trace into subgraphs and mapping them to SEBs are interdependent, and thus, need to be handled concurrently. Section 6.3.3 details our compiler analysis and mapping algorithms for a near-optimal breakdown of traces into subgraphs supported by the BERET hardware. In order to decide this set of SEBs, we performed detailed analysis on traces from SPEC integer benchmarks, Linux utilities, encryption and media kernels. Section 6.3.4 discusses this procedure and also uses trace analysis to guide sizing of various microarchitectural sub-components within BERET (internal register file, configuration RAM, etc.).

The above described execution model of the BERET microarchitecture is quite effective at saving energy. These savings can be broadly attributed to reducing: 1) instruction fetch, decode cost and 2) register access cost. First, once BERET is initialized with a trace to execute, there is no further instruction cache access. This eliminates redundant instruction cache access, fetch stage logic, and decode logic for the repeated sequence of instructions within a trace. Second, the register file accesses are cheaper as well as less frequent in the BERET design. The small size of the BERET internal register file makes the

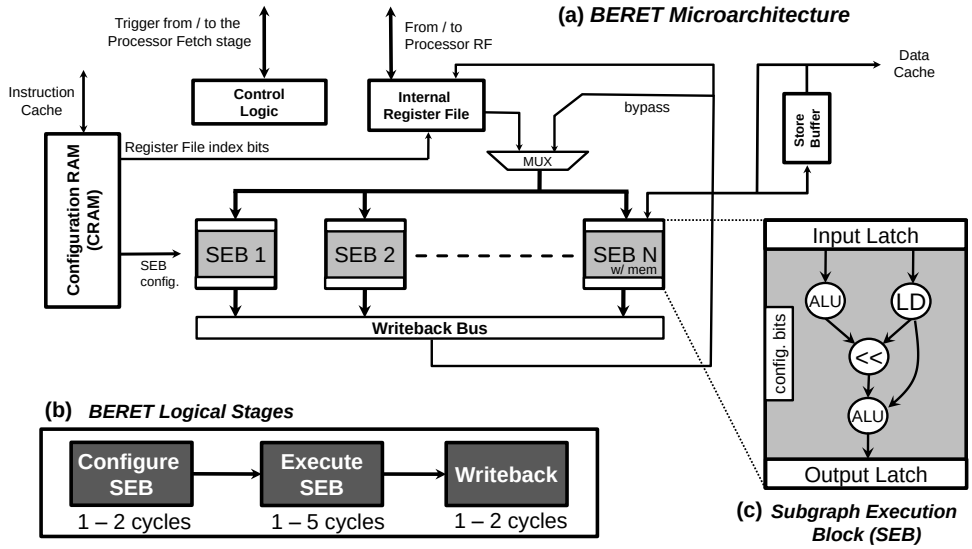



**Figure 6.5: The BERET Microarchitecture: (a) the block diagram of the BERET hardware, (b) logical stages in the microarchitecture, and (c) a closer look at a subgraph execution block (SEB).**

accesses cheaper, while the subgraph execution model minimizes register reads and writes for intermediate values in a program data flow.

### 6.3.2 Hardware Design

Unlike a regular pipeline, the BERET hardware deals with the execution of a small snippet of code (∼20 instructions), containing a small number of a live registers (∼6), and no internal control divergence. Further, the execution is conducted at the granularity of data-flow subgraphs, instead of individual instructions. These differences guide the following discussion about the design and working of BERET.

#### 6.3.2.1 Basic Microarchitecture

Once a trace is configured on BERET, it acts as an independent execution engine. Given the small size of a trace, and no internal control divergence, the BERET microarchitecture has a simplified front-end. However, it allocates significantly more resources to the exe-

cution back-end for running a wide variety of data-flow subgraphs. Figure 6.5(a) shows a block diagram of the BERET microarchitecture. Here, the configuration RAM (CRAM) stores the microcode for subgraphs in a trace, register file is for the internal data state, subgraph execution blocks (SEBs) are the equivalent of functional units, and control logic is to orchestrate the operation. In reality, the control logic is distributed across the entire fabric, with connections to virtually every component. The block diagram hides these connections for the sake of clarity.

Logically, BERET execution can be divided into three stages: 1) Configure SEB, 2) Execute SEB, and 3) Writeback results (Figure 6.5(b)). For every subgraph in the trace, the first step is sending (microcode) configuration bits to the mapped SEB. During this configuration stage, the register file inputs are also read into the input latch of an SEB. In the second stage (execute SEB), the SEB that has its inputs latched, configuration defined, and is in possession of the execution token, fires its functional units. The execution can take multiple cycles depending upon the subgraph depth (longest chain of instruction dependencies). Once the execution completes, the SEB sends the result on the writeback bus, and broadcasts an execution token. This token is now taken up by some other ready-to-execute SEB, and the pipelined execution continues. A more detailed stage-by-stage description follows below:

**1. Configure SEB:** The task of this stage is to sequence through the subgraphs in a trace, and configure SEBs to execute them. The configuration for the entire trace is stored on the CRAM. For each subgraph, this contains the SEB mapping, the register live-ins and live-outs, literal inputs, and mode bits for functional units within the SEB. In the first cycle,

configuration bits are sent to the corresponding SEB, and register file access is made for two live-in values. In the second (optional) cycle, two more register live-ins can be read, or, when needed, the values are bypassed from the last executed subgraph.

**2. Execute SEB:** The second stage is responsible for the actual data computation on the SEBs. A SEB starts its execution when all the inputs are latched, configuration bits are available, and it possesses the execution token. The execution token is used as a serializing method to enforce in-order execution of subgraphs, and it keeps shuttling between SEBs. The execution can take multiple cycles, depending upon the subgraph depth, and concludes with values recorded in the output latch. In the event of cache miss, just like a regular pipeline, the SEB also stalls while waiting for the value.

Each SEB or subgraph execution block (Figure 6.5(c)) is an interconnected set of functional units (ALU, shifter, multipliers, etc), that represent a data-flow pattern. The number of functional units per SEB vary from two to six in our design space exploration (Section 6.3.4). The SEB structure has an input latch for live-ins, an output latch for live-outs, and a latch to store configuration bits. For every subgraph mapped, these bits decide the active functional units, their modes (add, subtract, etc), and flow of values between them. The selection of a good set of SEBs is central to the efficiency gains from mapping traces to BERET, and the pertaining discussion is presented in Section 6.3.4.

**3. Writeback:** This third and final stage is responsible for writing back the results from the last concluded subgraph execution to the BERET register file. All SEBs share a common writeback bus for this purpose, and any SEB that has its outputs ready, can request it.

Due to the enforcement of in-order subgraph execution, there can never be a contention for this bus.

#### 6.3.2.2 Handling Trace Exits

The microarchitectural description in the previous section assumes a straightforward execution scenario with indefinitely looping traces. However, in reality, the trace conditions would eventually dictate an exit, and a consistent program state has to be transferred back to the main processor. This is even more challenging when a side exit is taken in the middle of trace execution, because 1) the subgraphs are formed across control divergence boundaries, and assume that all instructions in the trace window execute in every iteration; 2) temporary register variables are excluded when mapping a trace to BERET, hence some of the live-ins required on the exit edge might not even be available.

There are two parts to resolving this challenge. First, BERET needs a mechanism to detect when a side exit is taken by a trace. Second, BERET is required to maintain a committed state (at iteration boundaries) as well as per iteration speculative state. In the case of a side exit, the committed state (from last trace iteration) is copied back to main processor, which resumes execution from the trace head.

**Detecting Side Exits:** We first convert all the branches in the trace with assert operations (similar to [79]). The functional units within SEBs recognize this operation, and raise an exit flag whenever an assert computes to a true condition. Whenever any of the SEBs flag an exit, the control logic initiates the copying out of the committed state.

**Maintain Speculative and Committed State:** For recovering from early trace exits, BERET needs to maintain a committed state from the last completed trace iteration. There are two parts of this state maintenance: register file state and memory state. For register files, BERET uses a design similar to the concept of shadow register files. Essentially, every logical register maintains two physical versions in the register file. The even iterations of the trace write back to version 0 of registers, and odd iterations write back to version 1. As the code is linear within a trace, every iteration will produce exactly the same set of live-outs. Thus, at any point in time, the committed register state from the last iteration is available for recovery.

To maintain the memory state from the end of previous iteration, BERET buffers the stores from the current iteration. The store buffer releases them when the current iteration successfully completes. The size of this store buffer is relatively small, as the number of stores in most traces stayed around four (Section 6.3.4).

#### 6.3.2.3 Processor Interfacing

The main processor requires two modifications to interface with BERET. First, the fetch stage maintains a table of trace header addresses in the loaded program. Whenever the program counter hits any of these locations, the fetch sends an *entry* trigger and the corresponding trace configuration address to BERET. The BERET hardware loads the configuration using the instruction cache, runs through the trace, and returns with an *exit* trigger to the fetch stage. The second modification allows the main processor's register file to be directly addressed by the BERET. This is required by the BERET to read the trace live-ins (at entry) and write back the trace live-outs (at exit). Note that no extra read / write ports

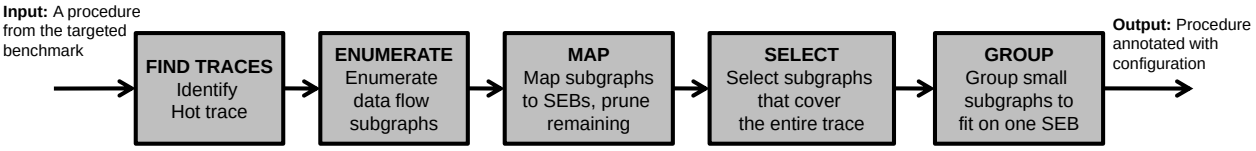



**Figure 6.6: The steps of identifying hot traces in a procedure and mapping them to the BERET hardware for energy-efficient execution.**

are necessary in the register file, as it is accessed in an exclusive manner (either by main processor or by BERET).

### 6.3.3 Mapping Traces to BERET

We use a comprehensive compiler flow to identify and map hot traces from a program onto the BERET hardware. While mapping, the objective is to segment an identified hot trace into a minimum number of subgraphs, each of which can execute on a SEB in the BERET hardware. Figure 6.6 shows the discrete steps involved in this flow, each of which is elaborated below:

**Find Traces:** Given a procedure, the objective of this step is to identify traces that have a very high probability to loop back, and rarely take side exits. For this step, we leverage the previously proposed Superblock identification heuristic [70]. Superblock formation is a static compiler analysis that groups together program basic blocks with a high likelihood of executing one after another. This gives the compiler opportunity to perform optimizations on a larger window of instructions. From the set of Superblocks composed in a procedure, for BERET mapping, we select the ones that have a looped structure (branch from the last basic block to the first basic block), with an 80% probability to loop back.

**Enumerate:** In this step, all data-flow subgraphs are enumerated from the given trace. The subgraphs can range in size from one operation, all the way to a pattern of 4-6 interconnected operations. Since we are enumerating all subgraphs, an operation can appear in more than one subgraph.

**Map:** This steps checks the feasibility of which data-flow subgraphs can actually run on the BERET hardware, and prunes away the rest. The mapping phase iterates over the enumerated subgraphs, and attempts to map each of them to a SEB in BERET. If the subgraphs can map to multiple SEBs, the mapping to the smallest SEB is recorded. On the other hand, if the subgraph does not map to any SEB, it is discarded.

**Select:** The input to this step is a set of SEB executable data-flow subgraphs from a hot trace. The selection phase is responsible for choosing the smallest subset of these subgraphs, while covering all instructions in the trace. This is equivalent to the set covering problem, which is NP-hard. We model it as a unate covering problem, and solve it using a branch and bound heuristic.

**Group:** Many of the subgraphs under utilize the SEB where they are mapped. This phase coalesces disconnected subgraphs, wherever it is possible, and places them on a single SEB.

After these steps for mapping, the compiler generates a configuration RAM code for this trace, and embeds it into the program binary. For ISA compatibility reasons, this configuration can be added as a part of the global data segment. Note that the compiler does not replace the original set of basic blocks in the program, as the execution reverts back to them in cases of early trace exit. Further, this keeps the code compatible on machines that

do not have the BERET hardware.

### 6.3.4 Design Space Exploration: SEBs and other parameters

The previous sections assume a fixed hardware design for BERET, including the set of SEBs and sizes for different structures within the microarchitecture. This section explains our methodology to arrive at these design specifics. All experiments here were conducted on traces from SPEC integer benchmarks, Linux utilities, encryption kernels, and media kernels.

#### 6.3.4.1 Determining SEB Collection

The objective of this study was to define the smallest collection of SEBs that exhibit a good mapping behavior for the traces in our benchmarks. Where, a good mapping implies that traces get divided into a small number of large subgraphs (say, subgraphs containing four to five instructions on average). Large subgraphs are better as they imply fewer CRAM accesses, more internal data forwarding, less number of register file accesses, and overall better energy savings. However, SEBs that can handle large subgraphs are also more inflexible, forcing the need to have a bigger collection of the same.

We resolve this situation by performing a subgraph exploratory study. First, we defined a set of *hypothetical SEBs* ranging in sizes from two execution units, all the way to six execution units. The execution units for this study were assumed to be universal, which can support any instruction type. Several interconnection patterns between these execution units were also incorporated, including linear chain, triangular formation and diamond formation. Next, the entire set of program traces were compiled for this collection of SEBs, gathering statistics on the frequency and instruction pattern of different subgraphs mapped

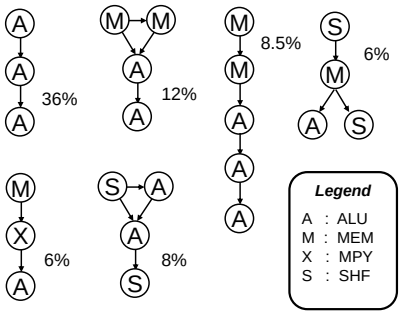



**Figure 6.7: The top six specialized SEBs from the final set of eight used in the BERET design. The percentages indicate the frequency of their occurrence in program traces.**

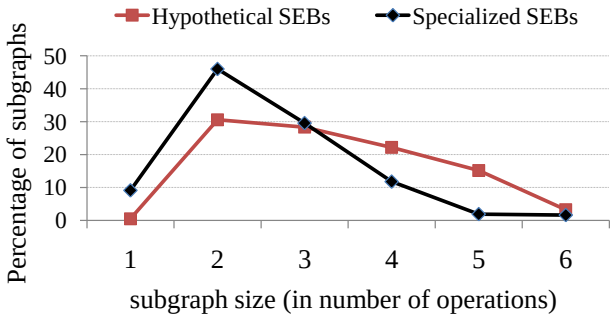



**Figure 6.8: The percentage distribution of subgraph sizes across all traces when using the hypothetical SEBs and our final selection of specialized SEBs. The average size of subgraphs for hypothetical SEBs at 3.26 was only marginally better than the same for our specialized SEBs at 2.56.**

to each SEB. From this list, we selected top eight *specialized SEBs* based on their frequency of occurrence across all traces, while maintaining a diversity in their sizes. The execution units specialization was limited to four types: ALU, Shifter, Multiplier or Memory Access Unit. Figure 6.7 shows top six specialized SEBs from our set of eight.

Figure 6.8 shows the distribution of subgraph sizes (across all traces) when using hypothetical SEBs and finally chosen specialized SEBs. The average subgraph size is 2.56 for our specialized SEBs, which is reasonably close to the best possible 3.26 in the case of hypothetical SEBs.

#### 6.3.4.2 Microarchitectural Parameters

Some of the important design parameters in the BERET hardware are the sizes of the CRAM, register file and store buffer. To determine a reasonable value for these parameters, we collected various statistics from the traces across all benchmarks. For CRAM size, we analyzed the distribution of number of subgraphs per trace. As much as 90% of the traces had number of subgraphs less 12. Consequently, we fixed the CRAM size at $16 \times 64$ bits

(our subgraph encoding fits in roughly 64 bits). For register file size, we analyzed the distribution of maximum live values per trace. This led to a register file with 8 entries. Finally, for the store buffer sizing, we looked at the distribution of store operations per trace, leading us to a store buffer with 6 entries.

## 6.4 Evaluation

### 6.4.1 Methodology

In order to evaluate the potential of the BERET design, we developed a comprehensive methodology involving compiler analysis for the identification and mapping of traces, an architectural simulator for performance, CAD tools for synthesis, place and route, power, area and finally, an energy simulator for computing total energy consumption while running a trace on BERET. Details about each of these components, along with benchmarks and baseline description follows below.

**Benchmarks:** A unique attribute of the BERET architecture is its relevance to both regular as well as irregular code based applications. The benchmark set was chosen to represent both these classes. We selected nine benchmarks from the SPEC integer suite (164.gzip, 175.vpr, 181.mcf, 197.parser, 254.gap, 256.bzip2, 401.bzip2, 429.mcf, 445.gobmk), three linux utilities (grep, cmp, lex), two encryption kernels (rc4, pc1) and five benchmarks from the MediaBench suite (cjpeg, gdmdecode, gsmencode, pgpdecode, pgpencode).

**Baseline Processor:** The ARM1176 [8] was chosen to be the baseline processor for comparison, a widely used processor in smart phones and portable electronics. Being a single-issue in-order pipeline, we consider the ARM1176 to be an aggressive baseline for showing energy efficiency improvements. According to the ARM website [8], an 800MHz

ARM1176 synthesized at 65nm technology node consumes roughly 160mW, which includes 16 KB level 1 instruction and data caches.

**Compiler Infrastructure:** The Trimaran compilation system [111] was used to implement the compiler flow that identifies hot traces and maps them to the BERET hardware. The trace identification component was implemented in OpenIMPACT (the front-end and profiling engine of Trimaran), whereas, the hardware mapping algorithms were implemented under ELCOR (back-end of Trimaran).

**Performance Simulation:** Cycle accurate simulators were used to model the performance of the baseline processor, as well as the execution time of traces mapped onto BERET. For the baseline single-issue in-order processor, we used SIMU, a performance simulator which is a part of the Trimaran package. A separate trace-based performance simulator was developed to measure the runtime of traces on the BERET hardware. This also accounted for the cost of execution control transfers between the main processor and BERET.

**Power and Area Estimation:** We implemented the BERET hardware in Verilog, and used a full CAD flow to synthesize (Synopsys Design Compiler), place and route (Cadence Encounter) and estimate power (Primetime PX). This was performed at the IBM 65nm technology node, while targeting a clock period of 1.25ns. This analysis gave us the power and area for all structures in the BERET hardware. Cache access power was estimated separately using CACTI [73] on a 16 KB, 4-way set associative cache.

**Energy Simulator:** The energy simulator was modeled after the BERET performance simulator. During the execution of a trace, it accumulates the energy consumed based on

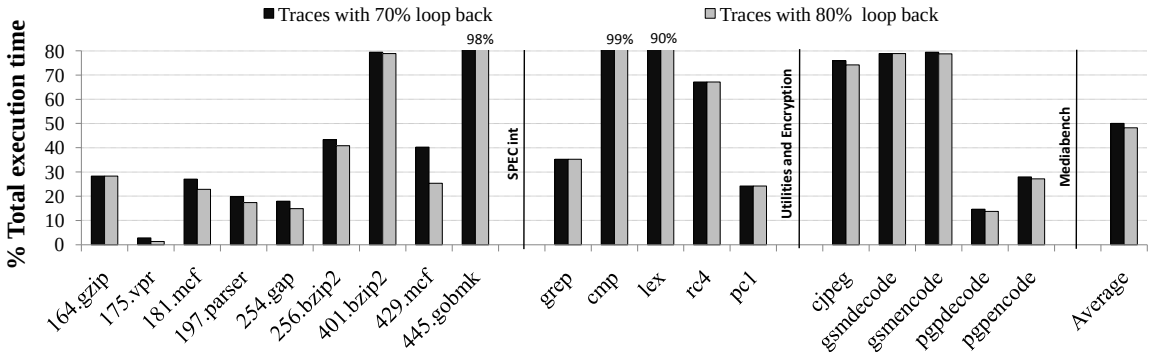


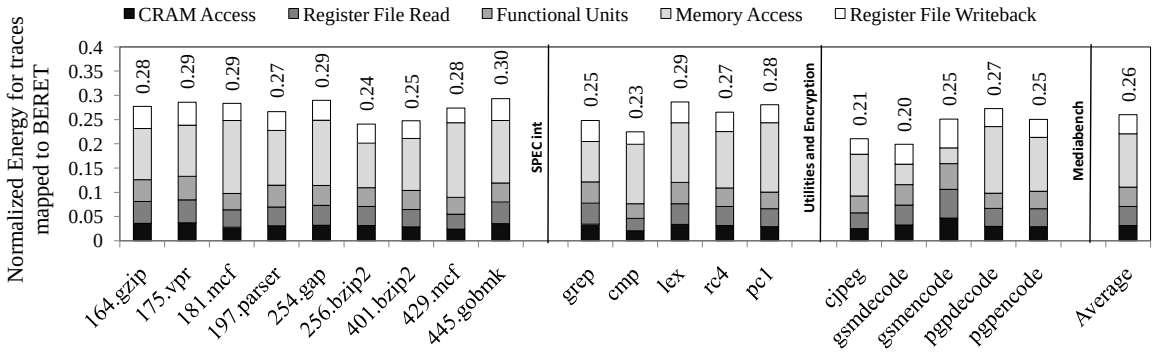

**Figure 6.9: Fraction of execution time spent in hot traces.**

the number of activations for structures within BERET. For every activation, the average power for the structure is extracted from the CAD synthesis.

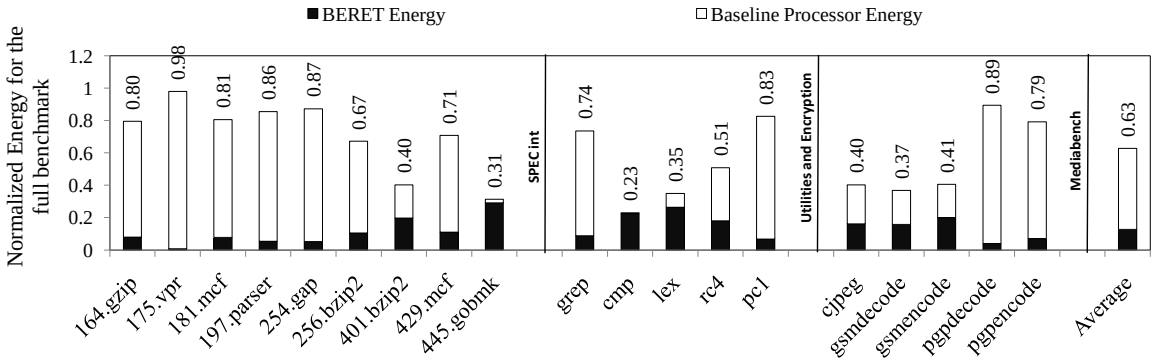
### 6.4.2 Results

#### 6.4.2.1 Execution Time Coverage of Traces

The fraction of a program's execution time spent in the hot traces determines the overall benefits from utilizing the BERET hardware. In our experiments, we used a static compiler analysis implemented in the Trimaran Compiler to identify hot traces. The results are shown in Figure 6.9. The first set of bars is for traces that loop back in 7 out of the 10 cases, whereas the second set is for traces that loop back in 8 out of the 10 cases. Almost all the benchmarks were found to spend at least 15% of their execution time in hot traces, with many spending as much as 70%. These results are especially encouraging as the benchmarks from SPEC2006 integer suite, which were compatible with our compilation flow, exhibit a large trace coverage. Also note that while we use a static compilation flow in our evaluation, it is our belief that a better hot trace coverage can be garnered by a dynamic compilation framework. That would translate into even higher energy savings.




(a) Hot trace energy consumption while they ran on the BERET hardware (normalized to main processor).




(b) Full benchmark energy savings while using the BERET hardware in conjunction with the main processor.

**Figure 6.10: Energy consumption relative to the baseline.**

### 6.4.2.2 Energy Savings

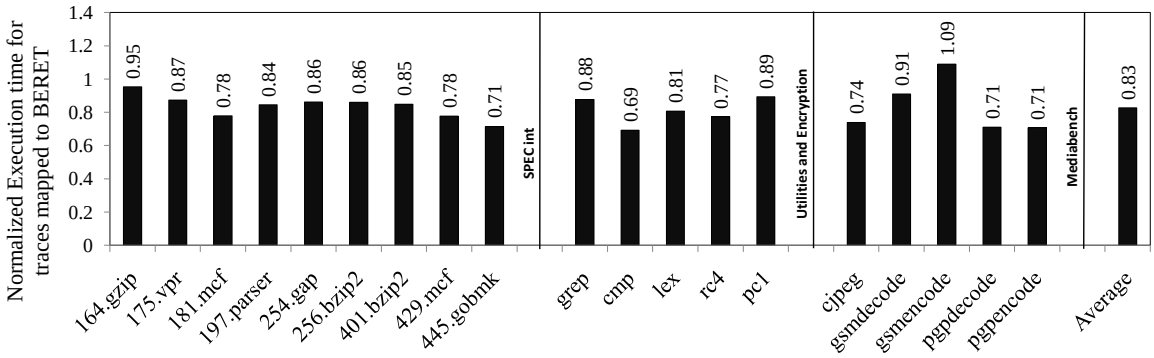
Figure 6.10a shows the normalized energy dissipation for the program regions running on BERET. The numbers shown also take into account the energy for transferring data and control in/out of the BERET hardware. On average, the proposed design reduces energy by a factor of 4X over a single-issue in-order core. For reference, a carefully designed ASIC can give anywhere between 10-50X energy reduction for regular codes. However, unlike BERET, they are not programmable across applications. Further, for irregular codes, ASICs cannot be expected to reach the same level of efficiency due to their diverse control and memory access patterns. The absolute energy dissipation by the BERET hardware was roughly 50pJ per instruction.

The breakdown in the bars depicts the energy spent by various structures within BERET. On average, the distribution of energy between structures was found to be relatively uniform with the exception of memory access, which also dominates the total energy dissipation. Energy savings in BERET are focused around the instruction supply (fetch, decode) and register data supply (fewer registers, reduced temporary variable accesses), with peripheral benefits from eliminating pipeline latches. Given that memory data supply is not really targeted by the BERET design, it is not surprising that it dissipates the maximum energy.

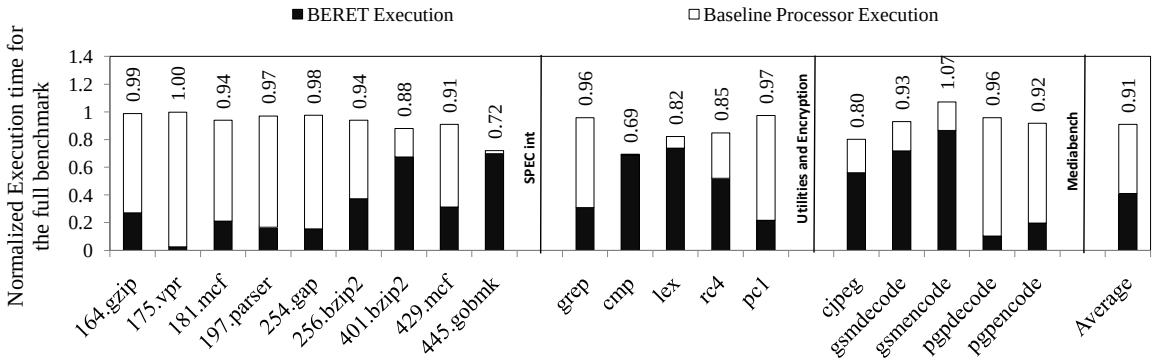
Figure 6.10b shows the energy numbers for the complete application runs. As expected, the overall benefits are correlated to the fraction of a program covered by the hot traces 6.9. The program portions that get mapped to the BERET hardware (black) garner significant energy savings, while the rest of them (white) dissipate the standard energy on the main processor. The full benchmark energy savings ranged from 2% on 175.vpr to 77% on cmp, with an average of 37%.

#### 6.4.2.3 Performance Comparison

The primary objective of the BERET design was to target energy savings in irregular codes, without sacrificing any performance. Fortunately, the use of a bulk execution model, using subgraphs instead of isolated instructions, gives a performance edge to BERET in addition to its energy benefits. Figure 6.11a shows the normalized execution time for code regions mapped to the BERET hardware. Some of the benchmark traces exhibit as high as 29% performance improvement, with an average of 17%. This improvement stems from the instruction level parallelism (ILP) achieved within an SEB as it executes data flow




(a) Hot trace execution time while they ran on the BERET hardware (normalized to the main processor).

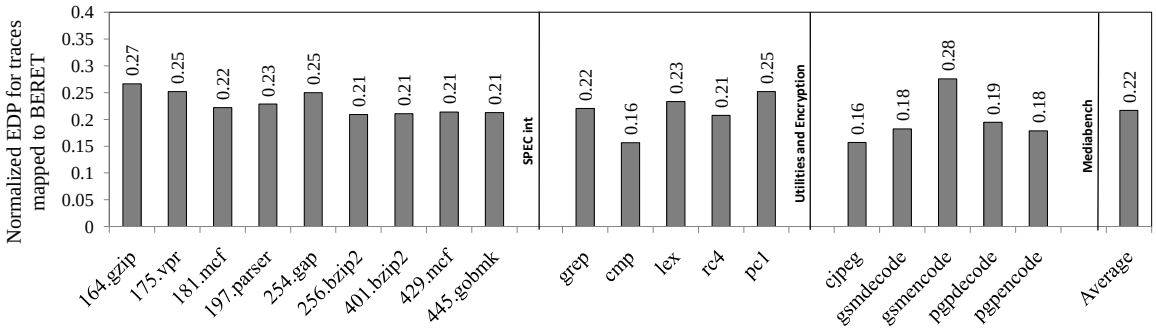



(b) Full benchmark execution time while using the BERET hardware in conjunction with the main processor.

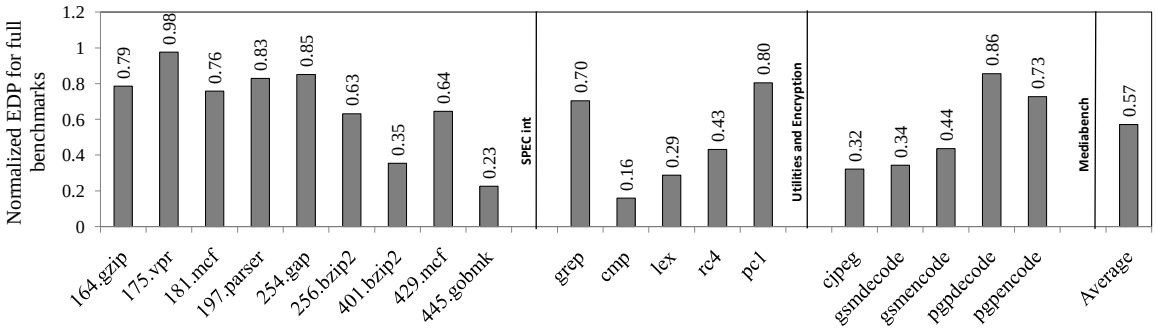
**Figure 6.11: Execution time relative to the baseline.**

subgraphs. For instance, a subgraph containing two `add` instructions feeding their outputs to a `xor` instruction would finish in two cycles, resulting in an IPC of *1.5*. Prior schemes on custom instruction execution [87, 27, 24, 107] depend on this exact behavior for their sizable performance improvements.

Figure 6.11b shows the execution time improvement for the entire benchmark execution. As in the case of energy savings, the performance improvements get diluted in accordance with the fraction of hot trace coverage. Overall, the execution time for the benchmarks evaluated is reduced by 9%.




(a) Hot trace EDP while they ran on the BERET hardware (normalized to the main processor).




(b) Full benchmark EDP while using the BERET hardware in conjunction with the main processor.

**Figure 6.12: EDP relative to the baseline.**

#### 6.4.2.4 Energy-Delay Product Improvement

The last comparative result we investigated is the Energy-Delay Product (EDP). Though architectural solutions often have the ability to trade performance for energy, improving both simultaneously is a difficult problem. EDP is a neutral metric for evaluating whether an architectural solution is an overall win while considering both performance and energy impact. Figure 6.12a shows the EDP improvement for the regions mapped to the BERET hardware, and Figure 6.12b is for the full benchmark execution. Amongst the SPEC integer benchmarks, 401.bzip2 and 445.gobmk stand out in terms of EDP improvements. Both these benchmarks spend a majority of their time in the BERET hardware, gathering energy as well as performance benefits. Across all benchmarks, the average EDP improvement was observed to be 43%.

**Table 6.1: Comparison to Prior Work.**

| | BERET | ASIC [77, 93] ASIP [44] | Loop Accelerators [36, 35, 121] | C-Cores [114] | ELM [31] | Programmable FUs [24, 87] |
|---|---|---|---|---|---|---|
| **Energy Savings** | High | V.High | V.High | High | V.High | Low |
| **Multiple Applications** | Yes | No | No | No | Yes | Yes |
| **Irregular Codes** | Yes | No | No | Yes | No | Yes |
| **Area** | Medium | Large | Medium | Large | Large | Small |
| **Processor Integration** | Co-processor | Stand-alone | Co-processor | Stand-alone | Stand-alone | In-pipeline |
| **Program Scope** | Traces | Full | Loops | Functions | Full | Op-chains |
| **Performance** | Medium | V.High | High | Medium | Medium | High |

#### 6.4.2.5 Area Overhead

The final design of the BERET architecture consisted of a 128 byte CRAM, 8-entry register file, 6-entry store buffer, a heterogeneous set of 8 SEBs, and miscellaneous logic and interconnects. The total area for this in the 65nm technology node (after place and route) was $0.396mm^2$. In the same technology node, the area of an ARM1176 core is $1.94mm^2$.

## 6.5 Related Work

The architectural designs for performance and energy have been an active area research for a long time. In this landscape (see Table 6.1), BERET stands out by being a general purpose compute engine that provides high energy-efficiency for regular (e.g., media kernels) as well as irregular codes (e.g., desktop applications and SPEC int). Further, the BERET design is a low cost engine that can be attached as a co-processor to the main pipeline, without any elaborate hardware or programming paradigm changes.

Specialized hardware designs [77, 80, 93] and instruction set extensions [44, 107] have long been a source of performance and energy efficiency for computations such as media kernels [54], encryption, signal processing [49]. ASIC designs are a good example of this, and get on the order of 40-50X energy efficiency improvements over simple RISC

processors. Loop accelerator (LA) [36] designs are a limited form of ASICs that target modulo-schedulable, regular loop bodies with highly predictable memory access patterns. More recently, some flexibility has also been incorporated in these LAs [35, 122, 25] to generalize them for more than one application. BERET differs from such LAs and traditional ASICs in two ways: 1) it targets *irregular codes*, that are heavily control divergent, hard-to-parallelize, and not well-suited to modulo scheduling; 2) it is general purpose and not application specific.

Irregular codes have also been targeted by a recent work titled Conservation Cores (C-Cores) [114]. C-cores borrows insights from prior spatial computation solutions [21] and synthesizes application-specific hardware for energy-efficiency improvements. However, this scheme requires an independent co-processor for every application, imposing heavy area and design time costs. In contrast, BERET engine is general purpose and not tied to any application or domain.

Another approach for irregular codes has been the use of subgraph accelerators like CCA [24], PRISC [87] to improve their performance. These solutions propose adding a customizable functional unit within the processor, that can improve performance for a range of data flow subgraphs encountered during a benchmark run. Unfortunately, the efficiency gains from these schemes are limited as they target only the back-end energy savings (data supply). The instruction supply still does all the redundant fetches and decodes. On the other hand, BERET targets both instruction and data supply savings.

ELM [31] is a programmable processor design dedicated to both instruction and data supply energy savings. Although it achieves considerable efficiency improvements, the targeted applications are regular kernels from the embedded systems.

The BERET design bears some resemblance to data flow machines, as it breaks down the recurring traces from a benchmark into constituent data flow subgraphs. However, the full blown data flow designs such as WaveScalar [104] and TRIPS [90] are more performance centric, and introduce large area and complexity costs. Another related effort is the Braids [112] architecture, that converts the pipeline back-end into a series of (homogeneous) subgraph execution units, called braid execution units (BEUs). However, unlike BERET, the Braids architecture is performance centric, and works towards achieving aggressive issue-widths in simple in-order cores.

Processor energy savings have also been actively pursued by major chip manufacturers. Some of the popular solutions have been Trace Caches [50] introduced by Intel Pentium 4 line of chips, and Loop Stream Detector (LSD) [97] introduced in the recent Intel Nehalem microarchitecture. Trace Caches buffers the sequence of instructions in a pre-decoded form, and for the addresses available in Trace Cache, the back-end directly reads them from there. This removes branch predictor and decoder activation cost, while also garnering execution speed-up. The LSD design is conceptually very similar. However, instead of storing traces, it can buffer loops with fewer than 28 micro-ops (compare this to Trace Cache that can store 12-K micro-ops). For any program loop that can be accommodated within LSD, instruction fetch and decode energy is saved. Unfortunately, a large fraction of loops in irregular codes are large, and cannot benefit from LSD. Further, both these solutions still incur the energy expenses from inefficiencies in the processor back-end.

Finally, reconfigurable architectures have been used in the past for performance and energy improvements. Garp [45] and Chimaera [120] use an FPGA-like substrate to map instruction sequences. Garp can also handle tight inner-most loops from an application.

However, the use of a reconfigurable fabric, and dependence on regular code behavior limits their overall usability and impact on general purpose energy efficiency.

## 6.6 Summary

With the growing importance of energy conservation in all domains of computing, there is a clear need for architects to develop efficiency solutions that apply to general purpose computing. This is especially true given that the embedded systems approach of designing special purpose hardware does not scale to the requirements of irregular and diverse code base in general purpose application space. Towards this end, this chapter identified the challenges posed by irregular codes, and developed BERET, an energy-efficient architecture for general purpose programs. Further, the BERET architecture is not application specific and can be programmed to deliver efficiency improvements for virtually any recurring trace of instructions. Fundamentally, BERET relies on these recurring traces to cut down on redundant instruction fetch and decode energy, and a bundled execution model to reduce register file access energy. We applied BERET on a variety of benchmarks from SPEC integer suite, Linux utilities, and MediaBench. On average, we found that BERET can reduce energy by a factor of 4X for the program regions it executes. The average energy savings for the entire application was 37% over a single-issue in-order processor.

Going forward, there are several avenues to improve the capability of BERET architecture. The current system relies on static compiler analysis to identify traces that can be mapped to BERET. Using a dynamic compiler analysis can significantly improve the code regions found, benefiting the energy savings for the full applications. Further, a performance side can also be added to BERET design. Presently, the system enforces serial exe-

cution of instruction bundles (subgraphs). By relaxing this constraint, a significant amount of ILP can be derived. Overall, we believe that BERET is well positioned as an execution engine for energy and performance gains in future computing systems.

# CHAPTER VII

# Conclusions

Performance has long been the primary design criteria for microprocessor architects. In the past decade, however, aggressive technology scaling has introduced newer dimensions and constraints to the processor design challenge. The issues range from designs nearing the power/thermal limitations to extreme process variation and wearout failures in the manufactured parts. The paradigm shift to multicore architectures has countered power and thermal issues to a certain extent, but the reliability against process variation and wearout has not benefited by much. Furthermore, good single-thread performance, a requirement for most applications, has suffered from this transition towards multicore designs. The confluence of these issues is creating an urgent demand for architectural innovations to efficiently address them.

The adaptive architectures presented in this thesis are our attempt in this direction. As part of this work, we have explored a variety of architectural solutions, supplemented by compiler techniques, to tackle reliability, performance and energy-efficiency demands expected in future systems. A common philosophy across all the solutions presented here has been their ability to adapt and reconfigure as per any static and/or dynamic variations

in the targeted system.

The first architecture presented in this thesis, StageNet, tackles the problem of unreliable silicon. This work contributes to the area of permanent fault recovery and reconfiguration by proposing a radical architectural shift in processor design. Motivated by the need for finer-grain defect isolation, networked pipeline stages were identified as an effective trade-off between cost and reliability enhancement. StageNet is also the first work to introduce a fully stage-by-stage decoupled pipeline microarchitecture. This allows flexible sharing of stage level resources between individual cores, without any significant performance loss (less than 10%).

For upcoming technology generations, StageNet can be employed for combating wearout failures as well as yield improvements. Even in scenarios without failures, the interconnection flexibility can be exploited to mitigate process variation. For instance, slower and faster stages can be connected together to balance off their timing requirements. Finally, for more distant future technologies (such as carbon nanotubes), StageNet will likely need to be used in conjunction with other methods to combat high failure rates.

More generally speaking, the decoupled microarchitecture of StageNet is a unique addition to the inventory of concepts applied by chip architects today. In addition to fault isolation, the decoupling techniques developed have broader applications. First, the stream-id concept can be extended for speculation in aggressive superscalar processors. Multiple execution paths can be tracked and squashed selectively for higher instruction level parallelism. Second, the bypass caches can be used to avoid global forwarding logic in deep pipelines. And finally, macro-ops can be effectively used for power/energy savings as they amortize the cost of instruction flow throughout the pipeline.

Alongside repair, detection of a failure is an equally important reliability challenge. This was the focus of our adaptive online testing framework. The key insight here was to leverage low level sensors to assess failure probability of various system resources, and suitably apply the tests. This way, a healthy system uses a fraction of resources for testing compared to another one nearing its time to failure. This scheme showed as much as 80% reduction in test cost. Overall, our efforts in reliability suggest that systematic introspection and architectural flexibility go a long way in saving fault tolerance costs.

While the original StageNet design was a reliability only solution, we soon observed that its interconnection flexibility is capable of providing more than just fault-tolerance. In the third architectural solution, named CoreGenesis, we built upon the StageNet design to form a unified performance-reliability solution. CoreGenesis enhances the base reliability architecture of StageNet with mechanisms to merge individual pipelines and form wider-issue processors. This adds a capacity for a higher single-thread performance to the baseline architecture. The CoreGenesis architecture, apart from achieving key milestones such as unified performance-reliability solution, configurable performance for in-order cores, no centralized structures, etc., also demonstrates that hardware costs (interconnection flexibility, in this case) can be effectively amortized across multiple challenges.

The last contribution of this thesis deals with the energy-efficiency challenge in the context of general purpose computing. Traditionally, the approach for improving efficiency has been through domain-specific and application-specific hardware design. However, this does not scale to the requirements of irregular and diverse code base in general purpose application space. In the proposed solution, named BERET, we leverage the recurring traces in irregular codes, and a bulk execution model to develop an energy-efficient co-

processing engine. The use of recurring traces allows BERET to cut down on redundant instruction fetch and decode energy, while the bulk execution model reduces register file accesses. With the major sources of inefficiencies covered in general purpose computing, BERET garners up to 4X savings for targeted program regions. Further, this demonstrates that a notable fraction of processor energy spending can be eliminated for program regions that exhibit (temporal) phases of regular and predictable behavior.

To conclude, it is our belief that future architectures have to look beyond evolutionary changes to sustain the benefits from technology scaling. The solutions presented in this thesis are a bold step forward in that direction.

# BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] M. Agarwal, B. Paul, and S. Mitra. Circuit failure prediction and its application to transistor aging. In *Proc. of the 2007 IEEE VLSI Test Symposium*, Apr. 2007. 84, 91

[2] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 470–481, 2007. 4, 15, 20, 47, 55, 83, 114

[3] Alpha. 21364 family, 2001. http://www.alphaprocessors.com/21364.htm. 124

[4] AMD. Amd 12-core opteron 6174 processor, 2011. http://www.amd.com/us/products/server/processors/6000-series-platform/Pages/6000-series-platform.aspx. 2

[5] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. Teramac – configurable custom computing. In *Proc. of the 1995 International Symposium on FPGA's for Custom Computing Machines*, pages 32–38, 1995. 47, 48

[6] A. Andrzejak, M. Arlitt, and J. Rolia. Bounding the resource savings of utility computing models, Dec. 2002. HP Laboratories, http://www.hpl.hp.com/techreports/2002/HPL-2002-339.html. 79

[7] A. Ansari, S. Gupta, S. Feng, and S. Mahlke. Zerehcache: Armoring cache architectures in high defect density technologies. In *Proc. of the 42nd Annual International Symposium on Microarchitecture*, pages 100–110, 2009. 116

[8] ARM. Arm11. http://www.arm.com/products/CPUs/families/ARM11Family.html. 23, 168

[9] ARM. Arm9. http://www.arm.com/products/CPUs/families/ARM9Family.html. 116

[10] T. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, 1999. 84

[11] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proc. of the 33rd Annual International Symposium on Microarchitecture*, pages 337–347, 2000. 113

[12] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004. 47, 114

[13] K. Batcher and C. Papachristou. Instruction randomization self test for processor cores. In *Proc. of the 1999 IEEE VLSI Test Symposium*, page 34, Washington, DC, USA, 1999. IEEE Computer Society. 88

[14] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and

J. Smullen. Nonstop advanced architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, June 2005. 15, 47

[15] K. Bernstein. Nano-meter scale cmos devices (tutorial presentation), 2004. 1, 13

[16] J. A. Blome, S. Feng, S. Gupta, and S. Mahlke. Self-calibrating online wearout detection. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 109–120, 2007. 3, 14, 41, 53, 69, 84, 87, 91

[17] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005. 1, 13, 51

[18] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proc. of the 2004 International Conference on Dependable Systems and Networks*, page 51, 2004. 4, 15, 47

[19] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc. of the 38th Annual International Symposium on Microarchitecture*, pages 197–208, 2005. 3, 14, 47

[20] D. Brooks, V. Tiwari, and M. Martonosi. A framework for architectural-level power analysis and optimizations. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000. 140

[21] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–26, 2004. 176

[22] L. Chen, S. Ravi, A. Raghunathan, and S. Dey. A scalable software-based self-test methodology for programmable processors. *Proc. of the 40th Design Automation Conference*, pages 548–553, June 2003. 88

[23] A. Christou. *Electromigration and Electronic Device Degradation*. John Wiley and Sons, Inc., 1994. 3, 14

[24] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Dec. 2004. 149, 173, 175, 176

[25] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, June 2008. 149, 176

[26] N. Clark, A. Hormati, S. Mahlke, and S. Yehia. Scalable subgraph mapping for acyclic computation accelerators. In *Proc. of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 147–157, Oct. 2006. 33

[27] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 129–140, Dec. 2003. 173

[28] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation.

In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 97–108, 2008. 69, 84

[29] K. Constantinides, S. Plaza, J. A. Blome, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. Bulletproof: A defect-tolerant CMP switch architecture. In *Proc. of the 12th International Symposium on High-Performance Computer Architecture*, pages 3–14, Feb. 2006. 19, 47

[30] W. Culbertson, R. Amerson, R. Carter, P. Kuekes, and G. Snider. Defect tolerance on the teramac custom computer. In *Proc. of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 116–123, 1997. 47

[31] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *IEEE Computer*, 41(7):27–32, July 2008. 150, 151, 175, 176

[32] R. Das, I. L. Markov, and J. P. Hayes. On-chip test generation using linear subspaces. In *Proc. of the 2006 IEEE European Test Symposium*, pages 111–116, Washington, DC, USA, 2006. IEEE Computer Society. 88

[33] J. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, Cambridge, MA, 1985. 113, 131, 134

[34] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 7–18, 2003. 69

[35] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 313–322, Feb. 2009. 6, 149, 153, 175, 176

[36] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Compiler-directed synthesis of multi-function loop accelerators. In *Proc. of the 2005 Workshop on Application Specific Processors*, pages 91–98, Sept. 2005. 175, 176

[37] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 149–159, Dec. 1997. 113

[38] S. Feng, S. Gupta, A. Ansari, and S. Mahlke. Maestro: Orchestrating lifetime reliability in chip multiprocessors. In *Proc. of the 2010 International Conference on High Performance Embedded Architectures and Compilers*, pages 186–200, Jan. 2010. 47

[39] J. Friedrich et al. Desing of the power6 microprocessor, Feb. 2007. In *Proc. of ISSCC*. 87

[40] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 173–181, June 1998. 150, 154

[41] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. Adaptive online testing for efficient

hard fault detection. In *Proc. of the 2009 International Conference on Computer Design*, 2009. 69

[42] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. Stageweb: Interweaving pipeline stages into a wearout and variation tolerant cmp fabric. In *Proc. of the 2010 International Conference on Dependable Systems and Networks*, June 2010. 129

[43] S. Gupta, S. Feng, A. Ansari, J. A. Blome, and S. Mahlke. The stagenet fabric for constructing resilient multicore systems. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 141–151, 2008. 52, 75, 83, 86, 89, 97, 99, 107, 108, 110, 111, 119, 120, 123, 132, 136, 137, 142

[44] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 37–47, 2010. 175

[45] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, Apr. 1997. 177

[46] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(1):33–38, 2008. 107

[47] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, and S. Ghosh. Hotspot: A compact thermal modeling method for cmos vlsi systems. *IEEE Transactions on*

*Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, May 2006. 18, 72, 136

[48] H. Inoue, Y. Li, and S. Mitra. Vast: Virtualization-assisted concurrent autonomous self-test. In *Proc. of the 2008 International Test Conference*, Sept. 2008. 84

[49] T. Instruments. Tms320c2x user's guide, Jan. 1993. 175

[50] Intel. Intel xeon processor with 512 kb l2 cache, 2004. 177

[51] Intel. 6-core intel core i7-970 processor, 2011. http://ark.intel.com/Product.aspx?id=47933. 2

[52] E. Ipek, M. Kirman, N. Kirman, and J. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, pages 186–197, 2007. 10, 107, 108, 111, 120, 136

[53] ITRS. International technology roadmap for semiconductors 2008, 2008. http://www.itrs.net/. x, 56, 73, 136

[54] H. Kalva. The H.264 video coding standard. *IEEE MultiMedia*, 13(4):86–90, 2006. 175

[55] E. Karl, D. Blaauw, D. Sylvester, and T. Mudge. Multi-mechanism reliability modeling and management in dynamic systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(4):476–487, Apr. 2008. 91, 101

[56] E. Karl, P. Singh, D. Blaauw, and D. Sylvester. Compact in situ sensors for monitoring nbti and oxide degradation. In *2008 IEEE International Solid-State Circuits Conference*, Feb. 2008. 41, 53, 69, 84, 85, 87, 91, 102

[57] E. Karl, D. Sylvester, and D. Blaauw. Analysis of system-level reliability factors and implications on real-time monitoring methods for oxide breakdown device failures. In *Proc. of the 2008 International Symposium on Quality of Electronic Design*, pages 391–395, Washington, DC, USA, 2008. IEEE Computer Society. 85, 91

[58] V. Kathail, M. Schlansker, and B. Rau. HPL-PD architecture specification: Version 1.1. Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories, Feb. 2000. 42

[59] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner. Picoserver: using 3d stacking technology to enable a compact energy efficient chip multiprocessor. *ACM SIGPLAN Notices*, 41(11):117–128, 2006. 13, 16

[60] C. Kim, S. Sethumadhavan, M. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler. Composable lightweight processors. In *Proc. of the 40th Annual International Symposium on Microarchitecture*, pages 381–393, Dec. 2007. 107, 111, 112, 115

[61] A. KleinOsowski, K. KleinOsowski, and V. Rangarajan. The recursive nanobox processor grid: A reliable system architecture for unreliable nanotechnology devices.

In *International Conference on Dependable Systems and Networks*, page 167, June 2004. 48

[62] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, Feb. 2005. 13, 16

[63] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 81–92, Dec. 2003. 5, 107, 108, 111

[64] R. Kumar, N. Jouppi, and D. Tullsen. Conjoined-core chip multiprocessing. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 195–206, 2004. 73, 136

[65] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a RAW machine. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 46–57, Oct. 1998. 43

[66] X. Liang and D. Brooks. Mitigating the impact of process variations on processor register files and execution units. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 504–514, 2006. 54, 70

[67] X. Liang, R. Canal, G.-Y. Wei, and D. Brooks. Replacing 6t srams with 3t1d drams in the l1 data cache to combat process variability. *IEEE Micro*, 28(1):60–68, 2008. 81

[68] Y. Lin et al. Soda: A low-power architecture for software radio. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, pages 89–101, June 2006. 6

[69] T.-H. Lu, C.-H. Chen, and K.-J. Lee. A hybrid software-based self-testing methodology for embedded processor. In *2008 ACM symposium on Applied computing*, pages 1528–1534, New York, NY, USA, 2008. ACM. 88, 89, 93, 102, 103

[70] W. mei W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, 7(1):229–248, May 1993. 150, 154, 164

[71] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008. 3, 14

[72] S. Mishra and M. P. adn Douglas L. Goodman. In-situ sensors for product reliability monitoring, 2006. http://www.ridgetop-group.com/. 87

[73] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *IEEE Micro*, pages 3–14, 2007. 169

[74] H. H. Najaf-abadi and E. Rotenberg. Architectural contesting. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 189–200, 2009. 107, 113

[75] U. Nawathe et al. An 8-core, 64-thread, 64-bit, power efficient SPARC SoC (Niagara2), Feb. 2007. In *Proc. of ISSCC*. 2

[76] OpenCores. OpenRISC 1200, 2006. http://www.opencores.org/projects.cgi/web/or1k/openrisc_1200. 17, 23, 71, 72, 135

[77] M. Papadonikolakis et al. Efficient high-performance ASIC implementation of JPEG-LS encoder. In *Proc. of the 2007 Design, Automation and Test in Europe*, pages 159–164, Apr. 2007. 6, 149, 153, 175

[78] A. Paschalis and D. Gizopoulos. Effective software-based self-test strategies for on-line periodic testing of embedded processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):88–99, Jan. 2005. 88

[79] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001. 150, 154, 162

[80] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavorial synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, June 1989. 6, 149, 153, 175

[81] L.-S. Peh and W. Dally. A delay model and speculative architecture for pipelined routers. In *Proc. of the 7th International Symposium on High-Performance Computer Architecture*, pages 255–266, Jan. 2001. 44, 51, 54

[82] M. Postiff, D. Greene, S. Raasch, and T. Mudge. Integrating superscalar proces-

sor components to implement register caching. In *Proc. of the 2001 International Conference on Supercomputing*, pages 348–357, 2001. 40

[83] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, June 2009. 47, 48

[84] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 111–122, 2002. 83, 90, 94, 105

[85] PTM. Predictive technology model. http://ptm.asu.edu/. 136

[86] J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits, 2nd Edition*. Prentice Hall, 2003. 73

[87] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable function units. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, Dec. 1994. 149, 173, 175, 176

[88] B. F. Romanescu and D. J. Sorin. Core cannibalization architecture: Improving lifetime chip performance for multicore processor in the presence of hard faults. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008. 48, 107, 108, 111

[89] P. Salverda and C. Zilles. Fundamental performance constraints in horizontal fu-

sion of in-order cores. In *Proc. of the 14th International Symposium on High-Performance Computer Architecture*, Feb. 2008. 111, 112, 119, 130

[90] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003. 177

[91] S. Sarangi, B. Greskamp, R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas. Varius: A model of process variation and resulting timing errors for microarchitects. In *IEEE Transactions on Semiconductor Manufacturing*, pages 3–13, Feb. 2008. 51, 54, 72, 101

[92] S. Satpathy, Z. Foo, B. Giridhar, D. Sylvester, T. Mudge, and D. Blaauw. A 1.07 tbit/s 128128 swizzle network for simd processors. In *Proc. of the 2010Symposium on VLSI Technology*, 2010. 129

[93] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002. 6, 149, 153, 175

[94] L. Shang, L. Peh, A. Kumar, and N. K. Jha. Temperature-aware on-chip networks. *IEEE Micro*, 2006. 43

[95] P. Shivakumar, S. Keckler, C. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proc. of the 2003 International Conference on Computer Design*, page 481, Oct. 2003. 4, 13, 15, 19, 47, 83

[96] D. Siewiorek and R. Swarz. *Reliable Computer Systems: Design and Evaluation, 3rd Edition*. AK Peters, Ltd., 1998. 3, 14

[97] R. Singhal. Inside intel next generation nehalem microarchitecture, 2008. http://software.intel.com/file/18976. 177

[98] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995. 111, 112

[99] D. Sorin, M. Martin, M. Hill, and D. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. *Proc. of the 29th Annual International Symposium on Computer Architecture*, pages 123–134, 2002. 83

[100] L. Spainhower and T. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(6):863–873, 1999. 47

[101] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, pages 276–287, June 2004. 18, 20, 136

[102] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Proc. of the 2004 International Conference on Dependable Systems and Networks*, pages 177–186, June 2004. 101

[103] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 520–531, June 2005. 4, 13, 15, 47, 72

[104] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003. 177

[105] D. Sylvester, D. Blaauw, and E. Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Journal of Design and Test*, 23(6):484–490, 2006. 4, 15, 20, 47, 114

[106] D. Tarjan, M. Boyer, and K. Skadron. Federation: Repurposing scalar cores for out-of-order instruction issue. In *Proc. of the 45th Design Automation Conference*, June 2008. 10, 107, 111, 112

[107] Tensilica Inc. *Diamond Standard Processor Core Family Architecture*, July 2007. http://www.tensilica.com/pdf/Diamond WP.pdf. 6, 149, 173, 175

[108] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 363–374, June 2008. 54

[109] Tilera. Tile64 processor - product brief, 2008. http://www.tilera.com/pdf/. 2

[110] A. Tiwari and J. Torrellas. Facelift: Hiding and slowing down aging in multicores.

In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 129–140, Dec. 2008. 81

[111] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/. 42, 134, 169

[112] F. Tseng and Y. N. Patt. Achieving out-of-order performance with almost in-order complexity. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 3–12, June 2008. 177

[113] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, S. Malik, and D. I. August. The liberty simulation environment: A deliberate approach to high-level system modeling. *ACM Transactions on Computer Systems*, 24(3):211–249, 2006. 29, 42, 71, 135

[114] 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 *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–218, 2010. 2, 5, 148, 149, 175, 176

[115] K. Wang and C.-K. Wu. Design and implementation of fault-tolerant and cost effective crossbar switches for multiprocessor systems. *IEE Proceedings on Computers and Digital Techniques*, 146(1):50–56, Jan. 1999. 68, 75

[116] C. Weaver and T. M. Austin. A fault tolerant approach to microprocessor design. In

*Proc. of the 2001 International Conference on Dependable Systems and Networks*, pages 411–420, Washington, DC, USA, 2001. IEEE Computer Society. 3, 14

[117] D. Wilson. The stratus computer system. *Resilient Computing Systems*, 1:208–231, 1986. 47

[118] E. Wu, J. M. McKenna, W. Lai, E. Nowak, and A. Vayshenker. Interplay of voltage and temperature acceleration of oxide breakdown for ultra-thin gate oxides. *Solid-State Electronics*, 46:1787–1798, 2002. 3, 14

[119] T. T. Ye, L. Benini, and G. D. Micheli. Analysis of power consumption on switch fabrics in network routers. In *Proc. of the 39th Design Automation Conference*, pages 524–529, 2002. 129, 136

[120] Z. A. Ye et al. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 225–235, 2000. 177

[121] S. Yehia et al. Exploring the design space of LUT-based transparent accelerators. In *Proc. of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 11–21, Sept. 2005. 175

[122] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 277–288, 2009. 6, 149, 153, 176

[123] S. Zafar et al. A model for negative bias temperature instability (nbti) in oxide and high k pfets. In *Symposium on VLSI Technology*, pages 45–50, 2004. 3

[124] J. Zeigler. Terrestrial cosmic ray intensities. *IBM Journal of Research and Development*, 42(1):117–139, 1998. 3, 14