Improving Soft Error Reliability in Modern Processors

Verbeteren van de betrouwbaarheid met betrekking tot tijdelijke fouten
in hedendaagse microprocessors

Ajeya Naithani

Promotor: prof. dr. ir. L. Eeckhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de ingenieurswetenschappen: computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. K. De Bosschere
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2019 - 2020

UNIVERSITEIT
GENT

ISBN 978-94-6355-323-0
NUR 980, 987
Wettelijk depot: D/2019/10.500/131

Examination Committee

Prof. Hennie De Schepper, *chair*
Department of Electronics and Information Systems
Ghent University, Belgium

Prof. Lieven Eeckhout, *supervisor*
Department of Electronics and Information Systems
Ghent University, Belgium

Prof. Koen De Bosschere, *secretary*
Department of Electronics and Information Systems
Ghent University, Belgium

Prof. Bart Dhoedt
Department of Information Technology
Ghent University, Belgium

Dr. Stijn Eyerman
Intel, Belgium

Prof. Antonio Gonzalez
Department of Computer Architecture
The Polytechnic University of Catalonia (UPC), Spain

Prof. Timothy M. Jones
Computer Laboratory
University of Cambridge, United Kingdom

Reading Committee

Prof. Koen De Bosschere
Department of Electronics and Information Systems
Ghent University, Belgium

Prof. Bart Dhoedt
Department of Information Technology
Ghent University, Belgium

Dr. Stijn Eyerman
Intel, Belgium

Prof. Antonio Gonzalez
Department of Computer Architecture
The Polytechnic University of Catalonia (UPC), Spain

Prof. Timothy M. Jones
Computer Laboratory
University of Cambridge, United Kingdom

To my family

Contents

Acknowledgements

I express my deep gratitude toward my adviser, Professor Lieven Eeckhout, for his valuable guidance and support during my PhD. He is a very hardworking and caring advisor. He taught me the process of conducting high-quality research in a systematic and insightful manner. Despite being always overwhelmed with work, the amount of time he devotes to the development of his students surpasses all expectations of an amazing PhD advisor. The lessons I learned from him will always be at the foundation of my professional life; it has been an honor to work under his supervision and learn from him.

I am really honored to have Professor Antonio Gonzalez and Professor Timothy Jones on my committee. I sincerely thank them for reading and evaluating my dissertation; their feedback was valuable in improving the quality of the dissertation. I also thank them for their willingness to travel to Ghent, despite their busy schedules.

It was a privilege to have Stijn Eyerman as my mentor during the first year of my PhD. Stijn taught me the fundamentals of processor architecture and the process of submitting a top-conference paper. Very special thanks to him for patiently listening to my questions and elegantly teaching me the process of refining details.

The HiPEAC summer school is not only the best venue to learn about emerging areas within computer architecture but also a place to have detailed discussions with experts about your topic of PhD research. I benefited tremendously from attending two editions of the summer school. I sincerely thank Professor Koen De Bosschere for organizing the high-quality summer school. I collaborated with Josue Feliu on several research projects and I thank him for the detailed technical discussions we had over the past four years. Almutaz Adileh was always ready to help, and I thank him for providing constructive feedback on several occasions during my PhD.

Many thanks to my PerformanceLab colleagues: Shoaib Akram, Sander De Pestel, Kristof Du Bois, Cecilia Gonzalez-Alvarez, Wim Heirman, Kartik Lakshminarasimhan, Wenjie Liu, Yuxi Liu, Jennifer Sartor, Sam Van den Steen, Lu Wang, Shiqing Zhang, and Xia Zhao, for discussions during the group meetings and making the work environment cheerful and productive. I would also like to thank the department staff for their help with administrative and tech-

nical issues; special thanks to Marnix for all the help with arranging conference travels.

I want to thank my friends in Ghent: Prashant Desai, Vijaykumar Guddad, Amit Kulkarni, Manoj Purohit, and Santanu Sahoo, for many enjoyable moments during my PhD. I am grateful to my friends, Shashank Joshi and Gaurav Pokhriyal, for warmly hosting me at their places in New Delhi during my visits to India.

I will always remain indebted to my parents for their endless love, support, and encouragement. No words can express my gratitude toward my brothers, sister, and sisters-in-law, for always being available to help and support me during every step of my PhD journey.

My loving wife, Akanksha, left her home to join me in a foreign country during the first year of my PhD. She has been a true companion during the highs and lows of my PhD journey, and the successful completion of this dissertation would not have been possible without her unwavering support and love. I owe her for being incredibly understanding and making my life fun-filled and adventurous.

Summary

Technology scaling and reduced operating voltages have rendered soft errors or transient faults of critical concern with respect to reliability in modern-day computer systems. Soft errors due to radiation and energy particle strikes may result in spurious bit flips that corrupt the architectural state leading to reduced system reliability, increased vulnerabilities, unexpected data loss, and catastrophic system crashes. Memory structures such as caches and translation lookaside buffers (TLBs) are commonly protected using error detection and correction. Core microarchitecture on the other hand is more difficult to protect and has become increasingly vulnerable to soft errors not just because of technology scaling, but also because of microarchitecture enhancements. Several of the superscalar out-of-order core structures, such as the reorder buffer (ROB) and issue queue, have increased dramatically over the past decade, e.g., the ROB and issue queue increased from 128 and 36 entries in Intel's 2008 Nehalem microarchitecture, to 224 and 97 entries in the current Skylake microarchitecture, respectively. Larger structures contain more architectural state and therefore increase the vulnerability to soft errors.

The research conducted in this dissertation focuses on mitigating the chance of an application encountering soft errors in modern processors. We have tackled the problem of high soft error vulnerability along three axes. First, we tackle soft error vulnerability on heterogeneous chip-multiprocessors (HCMPs). We propose *reliability-aware scheduling* to map applications to core types to reduce vulnerability to soft errors in heterogeneous multicores. Second, we observe that a large core microarchitectural state is maintained inside an out-of-order core, especially while running memory-intensive applications. In response, we propose *dispatch halting* to minimize the amount of vulnerable state in the core when load instructions access memory. Third, we analyze the vulnerability reduction delivered by runahead execution, an effective prefetching technique. Runahead execution improves reliability as it executes instructions speculatively. We contribute *precise runahead execution*, a new runahead execution paradigm that outperforms prior runahead proposals while improving reliability compared to an out-of-order processor.

Improving Reliability on Heterogeneous Multicores. Reliability-aware scheduling minimizes the vulnerability of a multiprogram workload running on an HCMP. The HCMP in our setup offers two core types: big out-of-order

cores and small in-order cores. The vulnerability of an application running on a core is determined by three key factors: (1) the size of the core microarchitecture structures, (2) the number of processor bits exposed by the committed instructions of an application, and (3) performance of the application on the core. We demonstrate that a straightforward characteristic of an application, such as memory intensity or compute intensity, is not indicative of the overall vulnerability of the application. In fact, the net vulnerability of an application is the outcome of a complex interaction of various application characteristics, and a dynamic mechanism must be developed for executing an application on the most appropriate core type in an HCMP.

We also observe the lack of a suitable reliability metric for multiprogram workloads executing on an HCMP. While soft error rate (SER) accurately captures the vulnerability of a single program on a core, SER cannot be used as a metric to assess the vulnerability of an application running as part of a multiprogram workload in an HCMP. This is because SER does not account for the performance impact of shared resource contention and core heterogeneity in an HCMP. We introduce a novel metric, *system soft error rate (SSER)*, that accounts for the impact shared resource contention and core heterogeneity has on per-application performance. The reliability-aware scheduler monitors vulnerability on either core type for all of the co-running applications, and schedules the applications to big and small cores for improved overall system reliability. The scheduler adapts to dynamic phase changes during application execution while relying on SSER for quantifying the system reliability of multiprogram workloads. The scheduler leverages a counter architecture to track occupancy in various hardware structures. Relative to a performance-optimized scheduler, the proposed reliability-optimized scheduler improves soft error reliability by 25.4% on average degrading performance by only 6.3%.

Improving Reliability on Out-of-Order Cores. Modern out-of-order cores feature large microarchitectural structures including the ROB, issue queue, load queue, store queue and register file. Since the instructions are committed in program order, if the instruction at the head of the reorder buffer is a long-latency load waiting for data to return from memory, all instructions following the load also wait inside the pipeline, thus getting exposed to soft errors. For memory-intensive applications, we show that 67% of the soft error vulnerable state is exposed while waiting for memory accesses. Dispatch halting is a microarchitectural technique to prevent instructions from encountering soft errors during memory accesses. Dispatch halting has two variants: proactive dispatch halting and reactive dispatch halting. Proactive dispatch halting relies on a load miss predictor to predict long-latency load instructions, and halts the dispatch after a long-latency load. All instructions after the load are buffered in the front-end. To compensate for the performance degradation caused by not generating memory-level parallelism after the load, proactive dispatch halting speculatively executes a copy of the selected instructions from the buffered instructions. Reactive dispatch halting, in contrast, does not employ a predictor. Instead, it waits for the back-end to be static for sufficient time, and then marks the back-end as speculative. Reactive dispatch halting also buffers instructions

in the front-end and replays them when the long-latency load is about to return. By building on the premise that it is better to speculate than execute normally under a memory access, dispatch halting is able to improve the mean time to failure for a set of representative memory-intensive benchmarks by more than 2×.

Evaluating Reliability and Performance of Runahead Execution. In addition to exposing soft error vulnerability, it is equally detrimental to performance when the head of the ROB is blocked by a long-latency load instruction. In runahead execution, once the ROB fills up after a blocked head, the processor enters into speculative mode, and executes future instructions speculatively to generate memory prefetches. When the blocking load returns, the pipeline is flushed and the core fetches and processes instructions again starting from the blocking load. Runahead buffer, the latest improvement upon runahead execution, finds the most dominant chain of instructions leading to a blocking load, and stores it in a buffer. The chain of instructions is then executed from the buffer and the front-end is turned off to save power. Runahead execution executes instructions speculatively, hence it does not expose vulnerable state in this duration.

We identify three major shortcomings for runahead execution: (1) flushing and refilling the pipeline incurs overhead, (2) prefetch coverage is limited, and (3) runahead does not exploit short runahead intervals for generating prefetches. We propose precise runahead execution (PRE) to eliminate these shortcomings. PRE builds on the key observation that when entering runahead mode, the processor does not need to release state as it has enough issue queue and physical register file resources to speculatively execute instructions. PRE instead uses a novel register renaming mechanism to quickly free physical registers in runahead mode. In addition, PRE pre-executes only those instructions in runahead mode that lead to blocking load instructions. Finally, PRE optionally buffers decoded runahead micro-ops in the front-end to save energy. For a set of memory-intensive applications, we show that PRE achieves an additional 18.2% performance improvement over the recent runahead proposals while at the same time reducing energy consumption by 6.8%. PRE improves soft error reliability by 28% on average, compared to an out-of-order processor.

Samenvatting

Steeds kleinere transistors en gereduceerde voedingsspanning hebben van betrouwbaarheid met betrekking tot tijdelijke fouten (Eng. *soft errors* of *transient errors*) een belangrijk ontwerpscriterium gemaakt. Een tijdelijke fout treedt op ten gevolge van kosmische straling of energiedeeltjes en kan leiden tot bitfouten die de architecturale toestand compromitteren. Dit kan leiden tot incorrecte uitvoeringen, onverwacht dataverlies en systeemfalen. Geheugenstructuren zoals caches en TLBs (Eng. *translation look aside buffers*) worden typisch beschermd m.b.v. foutdetectie en -correctie. De structuren in de processorkern zijn daarentegen moeilijker te beschermen. Deze structuren zijn niet enkel vatbaarder geworden voor tijdelijke fouten door technologieschaling maar ook door wijzigingen in de microarchitectuur. Verschillende structuren in een hedendaagse *out-of-order* processor zijn dramatisch toegenomen in afmetingen, b.v., het *reorder buffer (ROB)* en het uitvoeringsbuffer (Eng. *issue buffer*) zijn toegenomen van respectievelijk 128 en 26 elementen in Intel's Nehalem microarchitectuur uit 2008 tot 224 en 97 elementen in de hedendaagse Skylake microarchitectuur. Grotere processorstructuren bevatten meer architecturale toestand en zijn dus meer vatbaar voor tijdelijke fouten.

Dit doctoraat heeft als doel het verbeteren van de betrouwbaarheid in hedendaagse processors door de kans op een tijdelijke fout te reduceren. Dit gebeurt op drie fronten. Ten eerste pakken we betrouwbaarheid aan in heterogene chip-multiprocessors (HCMPs). We stellen een nieuwe techniek voor, *reliability-aware scheduling*, die de betrouwbaarheid van een HCMP verbetert door toepassingen dynamisch in te roosteren op verschillende processortypes op basis van de uitvoeringseigenschappen van de toepassingen. Ten tweede stellen we vast dat een hedendaagse out-of-order processor een grote architecturale toestand opbouwt, in het bijzonder bij uitvoering van geheugenintensieve computertoepassingen. We stellen *dispatch halting* voor, een techniek die de vatbare toestand in de processor reduceert wanneer leesoperaties in de processorkern wachten op het geheugen, en dit zonder aan prestatie in te boeten. Ten derde bestuderen we de impact van *runahead*-uitvoering, een vorm van *prefetching*, op betrouwbaarheid. We stellen *precise runahead execution (PRE)* voor om tegelijkertijd de prestatie te verbeteren van runahead-uitvoering en tegelijkertijd de betrouwbaarheid te verbeteren t.o.v. een conventionele out-of-order processor.

Betrouwbaarheid verbeteren in HCMPs. Een heterogene chip-multiprocessor bestaat typisch uit een aantal hoog-performante processorkernen en een aantal laag-vermogen processorkernen. Het inroosteren van computertoepassingen op verschillende processortypes heeft een grote impact op de betrouwbaarheid van een heterogene chip-multiprocessor. Door toepassingen met verschillende uitvoeringseigenschappen in te roosteren op verschillende processortypes kan het vatbaar zijn voor tijdelijke fouten aanzienlijk gereduceerd worden. Het vatbaar zijn voor fouten wordt immers bepaald door drie factoren: (1) de grootte van de verschillende structuren in de microarchitectuur, (2) het aantal processorbits dat vatbaar is voor fouten tijdens de uitvoering van een computertoepassing, en (3) de prestatie van de toepassing op elk van de processortypes. We tonen aan dat het niet mogelijk is een eenvoudige eigenschap te bepalen die de betrouwbaarheid van een computertoepassing karakteriseert, zoals b.v. de graad aan rekenintensiteit of geheugenintensiteit. We stellen echter vast dat het vatbaar zijn voor tijdelijke fouten het resultaat is van een complexe interactie tussen verschillende uitvoeringseigenschappen van een computertoepassing en de onderliggende hardware. We dienen dus een techniek te ontwikkelen die de meest betrouwbare processorkern voor een gegeven computertoepassing dynamisch bepaalt.

Om dit onderzoek te kunnen verrichten, dienen we eerst een metriek te bepalen die de betrouwbaarheid van een HCMP quantificeert bij het simultaan uitvoeren van meerdere computertoepassingen. Ofschoon *soft error rate (SER)* een nauwkeurige metriek is voor een enkele computertoepassing op een enkele processorkern, is SER niet bruikbaar voor meerdere toepassingen op een HCMP; SER houdt immers geen rekening met de impact van het delen van resources en heterogene processorkernen. We stellen daarom een nieuwe metriek voor, namelijk *system soft error rate (SSER)*, die de impact van gedeelde resources en verschillende processorkernen op de prestatie in rekening brengt bij het bepalen van de betrouwbaarheid van een HCMP. De planner, de *reliability-aware scheduler*, monitort de betrouwbaarheid van een computertoepassing op de verschillende processorkernen en roostert de toepassingen vervolgens in op de processorkernen teneinde de betrouwbaarheid van het ganse systeem te verbeteren. De planner past de inroostering dynamisch aan naargelang de uitvoeringskarakteristieken van de toepassingen. De planner maakt hierbij gebruik van een tellerarchitectuur die de vatbare architecturale toestand bijhoudt in de verschillende processorkernen. De voorgestelde planner verbetert de betrouwbaarheid van een HCMP met gemiddeld 25,4% met een kleine impact op prestatie van 6,3% t.o.v. een planner die de prestatie maximaliseert.

Betrouwbaarheid verbeteren in een out-of-order processorkern. Hedendaagse out-of-order processorkernen bevatten een aantal grote structuren zoals het reorder buffer (ROB), instructiebuffer, leesbuffer (Eng. *load queue*), schrijfbuffer (Eng. *store queue*) en het registerbestand. Vermits instructies de architecturale toestand wijzigen in programmavolgorde, blokkeert de processor vaak wanneer een leesoperatie dient te wachten op data van het geheugen. De leesoperatie verhindert hierbij de daaropvolgende instructies waardoor een grote architecturale toestand opgebouwd wordt in het reorder buffer die vatbaar

is voor tijdelijke fouten. Voor geheugenintensieve computertoepassingen blijkt 67% van de toestand die vatbaar is voor tijdelijke fouten een gevolg te zijn van leesoperaties die het reorder buffer blokkeren.

Dispatch halting is een microarchitecturale techniek die de betrouwbaarheid van een processor verbetert in het geval van een geheugenoperatie. We stellen twee varianten voor: proactieve en reactieve dispatch halting. Proactieve dispatch halting voorspelt toekomstige leesoperaties met een lange latentie (m.a.w. leesoperaties die leiden tot een miss in de caches op de processorchip), en blokkeert de dispatch trap in de processorpijplijn. Instructies opgehaald na de leesoperatie worden gebufferd in het begin van de pijplijn. Teneinde geen prestatieverlies te leiden, voert proactive dispatch halting deze instructies speculatief uit om onafhankelijke geheugenoperaties parallel te kunnen afhandelen. Na het deblokkeren van de pijplijn worden de instructies opnieuw (niet-speculatief) uitgevoerd. Reactieve dispatch halting daarentegen maakt geen gebruik van voorspelling. Reactieve dispatch halting observeert de pijplijn en indien de pijplijn blokkeert omwille van een geheugenoperatie, treedt de pijplijn in speculatieve modus en worden toekomstige instructies speculatief uitgevoerd. Dispatch halting verdubbelt de betrouwbaarheid van een out-of-order processorkern door de microarchitecturale toestand t.g.v. een leesoperatie speculatief te maken, en dus niet langer vatbaar voor tijdelijke fouten.

Prestatie en betrouwbaarheid van runahead-uitvoering. Het blokkeren van de processorpijplijn door een leesoperatie leidt niet alleen tot de accumulatie van architecturale toestand die vatbaar is voor fouten, het leidt ook tot een aanzienlijk prestatieverlies. Runahead-uitvoering is een microarchitecturale techniek waarbij de processor in speculatieve modus gaat wanneer het reorder buffer vol is t.g.v. een geblokkeerde leesoperatie. Hierbij worden toekomstige instructies speculatief uitgevoerd teneinde toekomstige geheugenoperaties data te laten ophalen alvorens de computertoepassing de data effectief nodig heeft (Eng. *prefetching*). Wanneer de leesoperatie de pijnlijn deblokkeert, wordt de pijplijn genullifieerd en worden de instructies die speculatief uitgevoerd werden na de leesoperatie opnieuw opgehaald en uitgevoerd, dit maal niet speculatief. *Runahead buffer*, de meest recente verbetering t.o.v. runahead, identificeert de meest dominante keten van afhankelijke instructies van de blokkerende leesoperatie, bewaart deze in een buffer, en voert enkel deze keten van afhankelijke instructies uit terwijl het begin van de processorpijplijn afgeschakeld wordt om energie te besparen. Runahead voert instructies speculatief uit en genereert dus geen toestand die vatbaar is voor fouten.

We identificeren drie tekortkomingen m.b.t. runahead-uitvoering: (1) het nullifiëren en opnieuw opvullen van de pijplijn leidt tot een niet te verwaarlozen overhead, (2) het aantal nuttige prefetches is beperkt, en (3) korte runahead-intervallen worden niet geëxploiteerd. Om aan deze tekortkomingen tegemoet te komen, stellen we *precise runahead execution (PRE)* voor. De observatie waarop PRE gebaseerd is dat de processor voldoende resources heeft om instructies speculatief uit te voeren wanneer runahead-uitvoering gestart wordt. Het is m.a.w. niet nodig de processortoestand te nullifiëren. PRE gebruikt een nieuw registerhernoemingsschema om fysieke registers te herge-

bruiken tijdens runahead-uitvoering. Bovendien voert PRE enkel instructies uit die leiden tot een blokkerende leesoperatie. Tenslotte houdt PRE optioneel instructies bij in het begin van de pijplijn om energie te besparen. We tonen aan dat PRE de prestatie verbetert met gemiddeld 18,2% en het energieverbruik reduceert met 6,8% voor geheugenintensieve computertoepassingen t.o.v. recent voorgestelde runahead-technieken. Bovendien wordt de betrouwbaarheid verbeterd met gemiddeld 28% t.o.v. een conventionele out-of-order processor.

List of Figures

List of Tables

List of Abbreviations

ABC	ACE Bit Count
ACE	Architecturally Correct Execution
ALU	Arithmetic Logic Unit
ARF	Architectural Register File
AVF	Architectural Vulnerability Factor
CPI	Cycles Per Instruction
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
DUE	Detected Unrecoverable Error
ECC	Error-Correcting Code
EMQ	Extended Micro-op Queue
FIFO	First In First Out
FIT	Failure In Time
FP	Floating Point
FU	Functional Unit
GP	General Purpose
HCMP	Heterogeneous Chip-Multiprocessor
HVF	Hardware Vulnerability Factor
IFR	Instrinsic Fault Rate
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
IQ	Issue Queue

ISA Instruction-Set Architecture

L1D Level-1 Data Cache

L1I Level-1 Instruction Cache

L2 Level-2 Cache

L3 Level-3 Cache

LLC Last-Level Cache

LQ Load Queue

LRU Least Recently Used

MLP Memory-Level Parallelism

MMU Memory Management Unit

MPKI Misses Per Kilo Instructions

MQ Micro-op Queue

MTTF Mean Time To Failure

MTTR Mean Time To Repair

OoO Out-of-Order

OS Operating System

PC Program Counter

PRDQ Precise Register Deallocation Queue

PRE Precise Runahead Execution

PRF Physical Register File

PVF Program Vulnerability Factor

RAS Return Address Stack

RAT Register Alias Table

RAW Read After Write

ROB Re-Order Buffer

RTL Register-Transfer Level

SDC Silent Data Corruption

SER Soft Error Rate

SMT Simultaneous Multithreading

SPEC Standard Performance Evaluation Corporation

SQ Store Queue

SRAM Static Random Access Memory

SSER System Soft Error Rate

SST Stalling Slice Table

STP System Throughput

TLB Translation Look-aside Buffer

WAR Write After Read

WAW Write After Write

Chapter 1

Introduction

Reliability is of critical concern for modern processors as a consequence of the shrinking process technology and decreasing transistor geometries [31, 74, 77, 83, 129, 154, 178, 192, 222]. With each new generation, transistor count has increased exponentially, as predicted by Moore's law, leading to dense integration of transistors on chip. Such advancements have improved performance manifold, however, the performance benefit had stifling consequences on power and reliability. Although the power consumption of each transistor reduced commensurate with its voltage and size, the power density of the chip continued to increase with technology scaling, producing extremely hot processor chips that are challenging to cool.

Processor errors have also increased with device scaling. A processor can encounter either a hard error or a soft error. A hard error implies a permanent damage to the processor chip; hard errors are a result of defects in silicon. The increase in temperature due to increased power accelerates processor wear-out [193, 194], and the increase in the number of transistors increases the number of defects [189]. A soft error, on the other hand, is typically caused by external radiations; soft errors are temporary, random, and do not manifest themselves again. Soft error rate increases with each technology generation: From 180 nm generation to 16 nm generation, the soft error rate was expected to increase by 100× [25]. Soft errors can severely undermine the reliability of modern-day computer systems [31, 74, 77, 83, 129, 154, 178, 192, 222], as soft errors may result in silent data corruptions or unrecoverable behaviors like system crashes [17, 71, 105, 217]. While error detection and correction techniques exist for on-chip memory structures such as TLBs and caches [8, 90, 130, 189], the core's microarchitectural structures, in contrast, are hard to protect. The vulnerability of the microarchitecture has also increased not only because of technology scaling but also due to microarchitectural enhancements.

Improving application reliability to radiation-induced soft errors is the main focus of this dissertation — we analyze soft errors encountered by programs executing on modern processors, and propose cost-effective techniques to mit-

igate them. A large body of work over the past two decades has targeted soft error reliability, focusing on soft error estimation [120, 137, 144], modeling [22, 145, 191, 210] and optimization [30, 153, 190, 199, 214]. Earlier techniques used radiation-hardened circuits [30] or some form of redundancy for detection and recovery from soft errors [67, 170, 171, 186, 198, 209]. However, these techniques incur significant performance, area and power overheads [136]. In general, reliability improvement requires some form of redundancy in time and/or space, and the key challenge is to achieve *minimal* overheads in terms of performance, area and power. The techniques proposed in this dissertation achieve a significant improvement in soft error reliability while minimizing the associated overheads. Techniques that tolerate long memory accesses by speculatively generating memory prefetches also improve reliability. We conduct a reliability assessment of one such well-known technique called runahead execution. In runahead execution, once the instruction window of an out-of-order processor is stalled on a memory access, the processor speculatively executes future memory accesses to bring their data close to the core. We also eliminate the shortcomings of runahead execution and propose a novel runahead technique to improve the performance of single-threaded applications on out-of-order processors.

This chapter first provides a brief background on how device scaling trends have resulted in increased power and reliability problems. We then explain how extreme miniaturization, increased processor microarchitecture state, and emerging trends towards heterogeneous computing systems have enabled an exponential increase in soft error vulnerability on current processors.

1.1 Technology Scaling Trends

In this section, we recap how the quest for high performance through aggressive device scaling was worthwhile for a long time, followed by ultimately slowing down because of high power consumption, and how device scaling trends have also lowered the reliability in modern processors.

1.1.1 Improved Performance through Scaling

In 1965, Gordon Moore made a prediction, known as *Moore's law* [131], that the number of transistors on a chip will double almost every year; this prediction was later revised to the doubling of transistors every two years [132]. For the next five decades, chip designers could reduce the size of a transistor and integrate more transistors on the chip, leading to today's billion-plus transistor chips. In 1974, Robert Dennard laid out the rules, known as *Dennard scaling* [52], for the scaling of various device and circuit parameters with the transistor dimension. In particular, decreasing transistor size also requires an equal reduction in the supply voltage to keep the electric field constant. The switching charge of the transistor also decreases proportionately. At a lowered switching charge, the transistor can switch at a much faster rate, improving

delay of the circuit by the same factor. This faster switching with each new generation paved the way for continuous performance improvement of a processor. A smaller transistor consumes less power; however, the power density of the transistor remains constant. Precisely, reducing the dimension of a transistor by a factor of k reduces the voltage, switching charge, circuit delay, and current by the same factor of k, where k is a unit-less scaling constant. The power consumption of the transistor decreases by a factor of k^2 without affecting its power density. Therefore, following Dennard scaling, the processor performance can be improved by reducing the size of a transistor with each technology generation while reaping the benefits of the constant power density of the transistor.

In addition to technology scaling, microarchitectural innovations like pipelining, out-of-order execution, branch prediction, and sophisticated prefetching techniques contributed equally to the performance improvement of the processor. The abundance of transistors led to an increasing size of the core microarchitectural structures for improved performance. Overall, for increased performance, hardware designers could integrate more transistors on a given chip area and benefited from the better delay characteristics of the transistors with scaling.

1.1.2 Increased Power through Scaling

Although the semiconductor industry benefited tremendously from the device scaling trends outlined by Dennard scaling, however, all but the supply voltage did not scale according to the ideal rules set forth by Dennard scaling. Over the generations, the supply voltage scaled at a much slower rate than the dimension of a transistor. The growing difference in scaling trends between the supply voltage and transistor size resulted in an increasing power density of the chip. From the year 1980 to 2000, the microprocessor power increased by about two orders of magnitude [68]. In the mid 2000s, the voltage scaling trends — and frequency scaling trends in turn — encountered a complete halt as a consequence of the processor power reaching unsustainable levels. The critical concern of the increased total power was further exacerbated by the leakage power. The leakage power was largely unnoticed for several years as dynamic power was the key contributor to the total processor power. However, the leakage power started to dominate total chip power as supply voltage approached threshold voltage. Ideally, a transistor is "off" below a threshold voltage; however, in reality, there is always a leakage current flowing. With device scaling, both the supply and threshold voltages are reduced [103], but the decreasing threshold voltage increases leakage power exponentially. Increased power directly increases temperature, demanding better techniques for cooling the chips. Therefore, it was apparent that reducing supply voltage further (and threshold voltage in turn) could lead to extremely hot chips that can be cooled only using special facilities (for example, liquid cooling). The phenomenon of power constraints leading to the end of voltage and frequency scaling trends is also termed as "power wall" [101]. Innovations leading to the design of mod-

ern chip-multiprocessors [158] are in response to the restraints imposed by the power wall. Although, computing industry soon realized that even powering on more than a certain number of cores on a (processor) die simultaneously is also limited by yet another phenomenon known as "dark silicon" [59] — one of the reasons that the number of cores integrated on a general-purpose multicore processor is typically limited to a (couple) dozen.

1.1.3 Soft Error Reliability

Analogous to the power wall, another equally precarious issue of reliability crept in with device scaling. As transistor count increased, so did the complexity of designs; the number of hardware defects increased and the chip verification became an extremely difficult task. Even more prevailing were the problems caused by lowered switching charge. At such low levels, the transistors are extremely fickle, and a minor perturbation in the switching charge can possibly change the state of the transistor. The minimum amount of charge required to correctly distinguish the state of a transistor, SRAM cell or DRAM cell is known as its *critical charge*. If an energy particle emanating from radiations deposits a charge that surpasses the critical charge of a device, the state of the device can temporarily flip, leading to a fault. An error caused by such a fault is known as a *transient error* or *soft error*. Therefore, on present-day processors, the successful execution of a program relies on a large number of weak transistors as it executes through the pipeline.

1.2 Motivation

Radiation-induced energy particles — neutrons from deep space or alpha particles emanating from the packaging material of a chip — can interact with the charge representing the state of a semiconductor device, like a transistor or an SRAM cell, and lead to a soft error. In this section, we motivate the need to design novel mitigation techniques for improving soft error reliability on modern processors.

1.2.1 Increase in the Rate of Soft Errors

Soft errors were not a serious reliability threat for general-purpose computing systems before the mid 1990's. However, with continued technology scaling, as the operating voltage of a transistor approached to the threshold level, the probability of a transistor encountering a soft error increased exponentially [151]. There are three key contributors to the increased soft error rates in modern chips. First, the switching charge of a transistor (or SRAM cell), and its critical charge in turn, is lowered to a degree where it is severely susceptible to be superseded by the charge deposited by an energy particle strike; the state of the device can change at a much lower critical charge now.

Second, the number of the particles of low energy is two orders of magnitude more than the particles of high energy in the cosmic rays [182]. Therefore, as the critical charge of a device decreases as a consequence of scaling, there are more energy particles that can surpass the critical charge now, further exacerbating the problem. Third, as predicted by Moore's law, the number of transistors per unit area of chip increased exponentially, and the number of soft errors increases with the number of transistors. On the positive side, the deposited charge also depends on the area of a circuit exposed to the radiation. Therefore, the chance of a transistor encountering a soft error decreased in proportion to its size; however, the reduced flux per transistor, and thus improved soft error reliability per transistor, is easily overshadowed by the number of transistors on a chip and the increased degree of susceptibility of each transistor to soft errors due to its reduced critical charge. Overall, device scaling trends led to the development of smaller and faster devices, however, the same trends rendered modern chips extremely vulnerable to soft errors [25, 151, 199].

1.2.2 Increase in Core Microarchitectural State

The improved performance of a transistor following technology scaling brought a significant forward leap in the performance of microprocessors. Furthermore, the microarchitectural innovations allowed processor designers to push for even higher performance than predicted by the device scaling trends. To extract maximum parallelism available in software, processor pipelines became deeper and wider, allowing multiple instructions to be issued each processor cycle. Each pipeline stage was stretched to the smallest possible operation to minimize the clock period. Modern processors can dynamically examine a large number of instructions, issue them out of program order, and employ separate structures such as the load/store queue for better handling of memory operations. For high degrees of instruction-level and memory-level parallelism, there are hundreds of instructions in flight between the fetch and retirement stages of the pipeline. The size of the microarchitectural structures required to hold these instructions has also increased accordingly, leading to the development of large monolithic processors.

Although today's processor designers face several challenges for improving the performance of out-of-order cores, nevertheless, the current size of these cores already pose a severe reliability challenge. Especially, when a memory operation blocks the retirement stage of the pipeline, a large number of instructions are waiting inside the core, occupying several microarchitectural structures. An energy particle strike can easily change the state of an already fragile processor bit, possibly causing an incorrect update to the application state. Therefore, for applications frequently accessing memory, it is imperative to devise novel solutions to minimize their soft error vulnerability. This dissertation proposes one such technique — known as dispatch halting — that more than doubles the mean time to failure of memory-intensive applications. Latency-tolerant techniques that exploit speculation to improve performance also improve soft error reliability. This dissertation further investigates the

reliability improvement accrued by runahead execution, a well-known latency-tolerant technique, and proposes precise runahead execution which achieves much higher performance than the state-of-the-art latency-tolerant techniques.

1.2.3 Emergence of Heterogeneous Computing Systems

Chip-multiprocessors (CMPs) became widely popular in response to the constraints imposed by the power wall [52, 158]. Inevitably, application behavior and the requirement for resources vary not only across different applications but also among disparate execution phases of the same application. Soon, it was apparent that multiple homogeneous cores of a CMP do not match the varied demands from all applications, and the computing landscape plunged into an era of hardware heterogeneity or specialization. Heterogeneity can be noticed today in multiple forms, for example, single-ISA heterogeneous processors that support heterogeneity at the microarchitecture level [112, 113], processors that integrate a GPU alongside a general-purpose CPU [7, 89, 97], and large-scale accelerators [96, 164]. Consequently, a large number of on-chip cores demand high memory bandwidth, pushing the need for heterogeneity to the memory subsystem as well. Memory-side accelerators [81] and the integration of die-stacked DRAM alongside the traditional DRAM [44, 123] are examples of how high memory bandwidth demands from the processor chip led to the development of novel hybrid memory technologies.

We target soft error reliability in single-ISA heterogeneous multicore processors. Single-ISA heterogeneous chip-multiprocessors (HCMPs) were proposed as an alternative to improve the throughput and energy-efficiency of chip-multiprocessors (CMPs) [112, 113] designed using monolithic homogeneous cores. Industry examples of single-ISA heterogeneous multicores include ARM's big.LITTLE [72], NVidia's Tegra [155], and Intel's QuickIA [42]. The fundamental observation underpinning the development of HCMPs is that the applications running on a CMP are inherently diverse in nature, and therefore, they have different computing requirements. Allocating equal computing resources to all co-executing programs results in underutilized resources and poor energy-efficiency. Additionally, applications exhibit different phase behavior at runtime and it is better to assign resources according to an application's runtime demand. To adapt to the application characteristics at runtime, HCMPs provides multiple core types — differing in their performance, complexity, and power envelops — on the same chip. The applications are dynamically migrated among different core types by system software, while maintaining far superior energy-efficiency than their homogeneous counterparts. For multiprogram/multithreaded workloads comprising of more applications/threads than the number of cores integrated in a given area of a CMP, HCMPs provide better system throughput as well.

In contrast to the performance and power implications, how application running on a heterogeneous system impacts the reliability of other co-executing applications has been a largely unexplored topic. First and foremost, there is not even a suitable metric available for quantifying reliability when multiple

applications are simultaneously running on an HCMP. The research conducted in this dissertation is the first to assess and improve the overall reliability of multiple co-executing applications on a heterogeneous system.

1.2.4 Soft Error Reliability at Scale

Even though the chance of a single processor bit encountering a soft error is rare, it poses a severe threat to the reliable operation of today's high-end supercomputing systems as well as safety-critical systems (e.g., autonomous vehicles). For example, the 64 KB L1 caches of a 104 K-node BlueGene/L system encounter one soft error every four hours [27]. In our analysis of the out-of-order vulnerability in Chapter 4, the back-end comprises of approximately 5.6 KB (see Table 4.2). Under the same circumstances as the BlueGene/L system, this translates into one soft error every two days. Two widely popular metrics in the fault-tolerance community are failure in time (FIT) and mean time to failure (MTTF). FIT rate is defined as the total number of errors in a billion device hours [137], and MTTF represents the time between two errors, and it is inversely related to the FIT rate. A recent neutron beam experiment on an Intel Xeon Phi 3120A coprocessor chip (comprised of 57 in-order cores) reported a FIT rate of 193 for the cores only [90, 156]. *Summit* is the current fastest supercomputer with over 2 million cores [53]. Assuming the same FIT rate as Xeon Phi, Summit cores encounter 1 error every 5.4 days. Note this is even a conservative estimate as prior work reports an MTTF ranging between 6.5 to 40 hours in today's petascale systems [58, 211]. Without proper soft error mitigation techniques, MTTF in high-end systems will soon be lower than their mean time to repair. In addition, soft error rates are often an order of magnitude higher than hard error rates [88], and every increase in soft error MTTF has a significant impact on the maintenance and operational cost of a high-end system.

1.3 Key Contributions

Technology scaling, advances at the microarchitecture level, and the emergence of hardware heterogeneity has put forward a soft error reliability challenge that requires novel mitigation techniques with negligible cost. In this section, we present our contributions for improving soft error reliability on commodity processors. We first introduce our novel reliability metric, *system soft error rate (SSER)*, for quantifying soft-error vulnerability on an HCMP. Building on SSER, we present *reliability-aware scheduling* for improving soft-error reliability of multiprogram workloads on HCMPs. We then focus on out-of-order cores, and devise a microarchitectural technique, called *dispatch halting*, for decreasing the vulnerability of single-threaded programs on out-of-order processors. Additionally, we evaluate the reliability improvement incurred by different variants of runahead execution, a well-known mechanism that improves both performance and reliability. Finally, we propose *precise*

runahead execution, a latency-tolerance technique that achieves significantly higher performance than prior runahead techniques while also improving reliability compared to an out-of-order core.

1.3.1 Reliability Metric for Multiprogram Workloads

Soft error rate (SER) is accurate metric to capture the vulnerability of a single-program workload. However, in an HCMP, multiple applications are concurrently executing, and they impact the performance of each other because of interference in shared resources. Adding raw SER numbers for each application to calculate the overall soft error rate of an HCMP does not take into account the performance impact applications have on each other. At any time, one application can have a higher SER than others, however, the application with lower SER can take much longer to complete its execution. Adding raw SER values gives more weight to the application with higher SER and does not account for the impact of performance on reliability. Therefore, we propose system soft error rate (SSER), a new metric that accounts for the interaction between performance and reliability. SSER weights per-application SER by its relative slowdown while running on an HCMP with other applications.

1.3.2 Reliability-Aware Scheduling

An HCMP features different core types — for example, big out-of-order and small in-order cores. Applications exhibit different soft error reliability characteristics on big versus small cores. This provides considerable opportunity to improve system reliability through scheduling on HCMPs. An oracle offline analysis considering an HCMP with two small and two big cores shows that reliability-aware scheduling can improve system reliability by 27.2% on average and up to 62.8%, while degrading performance by at most 7% on average compared to performance-optimized scheduling.

Therefore, in this work, we propose a reliability-aware scheduler that samples the reliability characteristics of running applications on either core type, and dynamically schedules applications on big versus small cores to improve overall system reliability. The proposed scheduler leverages a low-overhead (296 bytes per core) counter architecture to track hardware occupancy. Reliability-aware scheduling improves system reliability by 25.4% on average and up to 60.2% compared to performance-optimized scheduling, while degrading performance by 6.3% only. The proposed scheduler is robust across core count, number of big versus small cores, and frequency settings. Moreover, as a side effect, reliability-aware scheduling reduces power consumption by 6.2% on average compared to performance-optimized scheduling.

This work was published in:

A. Naithani, S. Eyerman, and L. Eeckhout. Reliability-aware scheduling on heterogeneous multicore processors. In *Proceedings of the 23rd IEEE Sym-*

posium on High Performance Computer Architecture (HPCA), pages 397–408, 2017

An extended version of this work was published in:

A. Naithani, S. Eyerman, and L. Eeckhout. Optimizing soft error reliability through scheduling on heterogeneous multicore processors. *IEEE Transactions on Computers*, 67(6):830–846, 2018

1.3.3 Dispatch Halting

Modern out-of-order cores expose a large microarchitectural state while executing an application. The severity of the problem is further aggravated for memory-bound applications as a large microarchitectural state is maintained inside an out-of-order core while waiting for data to return from memory. The size of this microarchitectural state exposed to transient faults has increased with every new processor generation.

To address the issue of high vulnerability of memory-intensive applications, we propose *dispatch halting*, a microarchitectural technique to improve their soft error reliability on out-of-order processors. We propose two variants, proactive and reactive dispatch halting, which offer different trade-offs. Proactive dispatch halting prevents instructions following a long-latency load from allocating large back-end structures and generates memory-level parallelism from the front-end itself. Reactive dispatch halting, on the other hand, marks the back-end speculative if no instruction is committed for a certain number of cycles. The speculative instructions generate memory accesses and a copy of them is replayed later. Proactive and reactive dispatch halting improve mean time to failure (MTTF) by 1.42× and 1.72× on average for the entire SPEC CPU2006 suite, respectively, and by 1.77× and 2.23× for the memory-intensive benchmarks, with minimal impact on performance. Proactive dispatch halting incurs a modest chip area (1.8 KB) and small power overhead (2.7%), whereas reactive dispatch halting incurs no additional chip area and a modest power overhead (6.2%).

This work is currently under preparation for submission in:

A. Naithani and L. Eeckhout. Dispatch halting. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2020

1.3.4 Precise Runahead Execution

Traditional runahead execution [139, 141] improves processor performance by accurately prefetching long-latency memory accesses. When a long-latency load causes the instruction window to fill up and halt the pipeline, the processor enters runahead mode and keeps speculatively executing code to trigger accurate prefetches. Runahead buffer, a recent improvement [79], tracks the chain of instructions that leads to the long-latency load, stores it in a buffer, and executes only this chain during runahead execution, with the purpose of generating

more prefetch requests during runahead execution. Runahead execution and runahead buffer, while targeting performance, also improve reliability, since the instructions are executed speculatively after a full-window stall. Therefore, in this work, we conduct an experiment to quantify the impact of runahead techniques on reliability.

We also observe that all prior runahead proposals have shortcomings that limit performance and energy efficiency because they release processor state when entering runahead mode and then need to re-fill the pipeline to restart normal operation. This significantly constrains the performance benefits and increases the energy overhead of runahead execution. In addition, runahead buffer limits prefetch coverage by tracking only a single chain of instructions that leads to the same long-latency load. We propose precise runahead execution (PRE) to eliminate the aforementioned shortcomings of prior runahead techniques. For memory-intensive workloads, PRE achieves an additional 18.2% performance improvement over the recent runahead proposals while at the same time reducing energy consumption by 6.8%. Relative to an out-of-order core, PRE, runahead execution, and runahead buffer improve reliability by 28%, 44%, and 49%, respectively.

This work was published in:

A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. Precise runahead execution. *IEEE Computer Architecture Letters*, 18(1):71–74, 2019

An extended version of this work is accepted for publication in:

A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. Precise runahead execution. In *Proceedings of the 26th IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2020

1.4 Dissertation Overview

This dissertation is organized in six chapters.

Chapter 2 provides necessary background on soft errors and existing methods for assessing soft error vulnerability. It also briefly covers processor microarchitecture to ease the understanding of Chapters 4 and 5.

In Chapter 3, we demonstrate the need for reliability-aware scheduling, propose our novel metric to quantify soft error vulnerability of multiprogram workloads running on heterogeneous multicores, and we discuss the working of our scheduler in detail. We further evaluate the robustness of our scheduler across different core count, frequency settings, and heterogeneity. The chapter also evaluates the impact of reliability-aware scheduling on power, and compares performance-, power-, and reliability-optimized schedulers. In addition, we extend the reliability-aware scheduler to adapt under performance constraints and also evaluate the importance of reliability-aware scheduling when the vulnerability of on-chip caches is also accounted for.

Chapter 4 targets the vulnerability of memory-intensive single-threaded workloads on out-of-order cores. The chapter shows that a large microarchitectural state is exposed to soft errors while waiting for a memory access to return. The chapter proposes a novel technique, called dispatch halting, to address the issue of high vulnerability during memory accesses. The chapter further explains the microarchitecture for dispatch halting in great detail, and evaluates the impact of dispatch halting on performance and power.

Chapter 5 first lists the shortcomings of prior runahead proposals, and then builds the case for precise runahead execution, a new runahead technique to improve single-thread performance. The chapter explains how precise runahead is able to efficiently recycle resources without flushing the ROB, and presents the performance and energy benefits of this new runahead technique. It further evaluates the reliability improvement achieved by different variants of runahead execution.

Finally, in Chapter 6 we conclude the dissertation and discuss some possible future work.

Chapter 2

Background

In this chapter, we present the background on soft error reliability and processor architecture. Section 2.2 explains the terminology from the fault-tolerance domain — it introduces soft errors and explains the types of errors that computing systems encounter today. Section 2.3 provides an estimation methodology, known as architecturally correct execution (ACE) analysis, used for assessing soft error reliability in this dissertation. Section 2.4 explains another equally popular soft error estimation technique known as fault injection. In addition to the characteristics of an application, soft error reliability is also determined by the processor running the application. Therefore, Section 2.5 briefly explains the working of the in-order and out-of-order processors. Finally, Section 2.6 summarizes the chapter.

2.1 The Soft Error Problem

This dissertation focuses on radiation-induced soft errors or transient errors. While soft errors due to other sources such as process variability and system noise can be handled before a chip is shipped [134], full protection against radiation-induced soft errors requires incorporating proper error detection and correction mechanisms into the chip. Radiation-induced soft errors are caused by two types of radiation: first, from alpha particles emanating from packaging material of a chip [126], and second, from the atmospheric neutrons [18, 19, 129, 223]. As described in Section 1.2, the energy particles due to radiation can flip the state of a transistor or SRAM cell, leading to a soft error. The incoming neutron flux every hour at the sea level is about 13 neutrons for every cm^2 of area [93], and the neutron flux increases with the altitude. Therefore, the probability of encountering a soft error increases with altitude. Soft errors due to alpha particle contamination were first reported by Intel in 1978 [126], and soft errors due to atmospheric radiations were first reported by IBM in 1984 [222]. The server systems of the companies like Sun Microsystems and Hewlett-Packard have also encountered crashes due to soft errors [134].

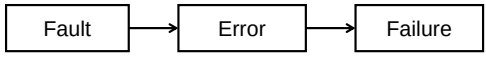

Figure 2.1: A fault is the underlying cause of an error which may possibly translate into a failure.

2.2 Terminology

2.2.1 Faults, Errors, and Failures

A *fault* is a defect in software or hardware. For example, a non-functioning transistor represents a hardware fault. Hardware faults are caused by imperfections in the manufacturing process of a silicon chip or due to interaction of the chip with its surroundings. Change in temperature or pressure can lead to a hardware fault. Similarly, energy particle strikes due to radiation can also lead to a hardware fault. Programming bugs that lead to unexpected behavior — for example, a division by zero or a data race — are software faults.

An *error* is the manifestation of a fault. For example, a bit changing from 0 to 1, or vice-versa, due to a malfunctioning transistor, is an error. An error guarantees the presence of a fault. However, a fault does not always lead to an error. For example, a fault in a flip-flop input value, when it is not latched to the output, does not lead to an error. Such a fault is called a *masked* fault.

A *failure* is defined as the inability of a system to meet certain requirements; these requirements can be in the form of correctness, timing deadline or some form of other guarantees. A failure is a possible outcome of an error. An error that is masked or corrected does not lead to a failure.

Depending on the nature of a fault, it belongs to one of the following three categories:

1. **Permanent faults:** A fault in a device is permanent if it requires to be fixed at the hardware level or it cannot be fixed. Such faults reappear upon every use of the device. Permanent faults are typically a result of the wearout of a device and mark the deteriorating lifetime of the device. Electromigration is an example of a permanent fault [69]. The error caused by a permanent fault is termed as a *permanent error* or *hard error*.

2. **Intermittent faults:** Intermittent faults only occur under certain conditions, for example, under elevated temperature. A stuck-at bit for an interval of time is an example of an intermittent fault. Voltage droops also lead to intermittent faults [69]. These faults indicate that the life of a device is nearing toward a permanent fault.

3. **Transient faults:** Transient faults are random and do not indicate a device lifetime problem. These faults can not be reproduced over multiple usages of the device. Transient faults are caused by energy particle strikes

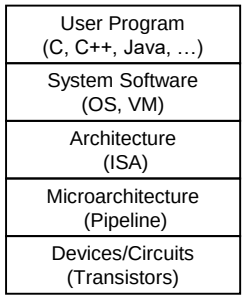

Figure 2.2: A program executes through several layers of abstractions. A fault at a lower layer can be masked from the higher layer of the abstraction stack.

on a gate or a transistor, which can potentially change the state of the electronic device.

Three widely popular metrics in the fault-tolerance community are intrinsic fault rate, FIT rate, and MTTF. FIT rate and MTTF can be used to express either the number of failures or the number of errors [134, 137, 214]. We review them in the context of soft errors below:

Intrinsic Fault Rate (IFR). A *raw error* occurs when a particle strike causes the state of a *one-bit* cell to flip or a logic element to produce incorrect output [120]; a raw error is analogous to a fault. IFR or *raw error rate* of a one-bit cell is defined as the probability for a raw error per second, or, in other words, the average number of raw errors per unit of time, for example, 10^{-6} per day. IFR depends on the circuit technology and the environment.

Failure In Time (FIT) rate. FIT rate is defined as the total number of errors in a billion device hours.

Mean Time To Failure (MTTF). MTTF represents the time between two errors, and it is inversely related to the FIT rate,

$$MTTF = \frac{1}{FIT} \tag{2.1}$$

However, FIT rate, being additive, is easier to comprehend.

2.2.2 Fault Masking and Fault Scope

A program executes through layers of abstraction from the user-level to the circuits. Only the input and output values communicated at the interface of two layers must be correct for correct execution of a program. Therefore, unless it impacts the output generated at a particular layer, a fault or an error at the layer of abstraction may not necessarily be visible at a higher layer. Sridharan et al. [191] propose the notion of the *system vulnerability stack*, which shows that a fault must be visible at each layer of the stack to lead to an error at the user level. For example, if both input logic values of a logical OR gate are 1,

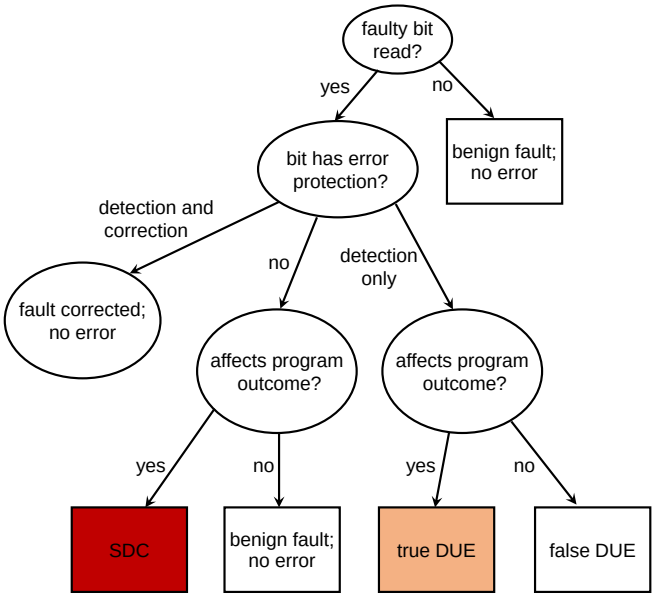

Figure 2.3: Different possible outcomes of a fault on a bit (Reproduced from [214].)

the outcome of the OR gate does not change if one of the input values changes from 1 to 0 due to a fault. This fault is visible at the circuit level but does not propagate to the higher layers in the stack. Such a fault is *masked* at the circuit level itself. A fault on an unallocated issue queue entry is also masked. Each layer can also implement its own error detection and correction capability. For example, Error Correcting Codes (ECC) employed at the lower levels of memory hierarchy (L2, L3 and DRAM) can detect and correct memory errors. That is, the error is corrected even before a load instruction moves the data from the L1 D-cache to a processor physical register, or even before the data reaches the L1 D-cache. Therefore, the error is visible within the boundary of the ECC but it is masked for the core microarchitecture. Additionally, a layer can raise an alarm when it detects an error, and notify the user or system software to correct the error, or even abort the program execution. For example, if the memory address generated by the Memory Management Unit (MMU) encounters a bit flip which changes the address to fall outside of the permitted address space of a process, the protection mechanism implemented by the OS will prevent the process from accessing the (corrupted) address. The OS will notify the user by generating an illegal exception or by simply terminating the process. Several prior works have studied the impact of masking at the various levels of the abstraction stack [50, 104, 137, 212, 214].

Figure 2.3, originally created by Weaver et al. [214], provides an alternative representation of how a fault in a bit can eventually lead to an error. The adverse eventual outcome of a fault can be either a *silent data corruption (SDC)* or a *detected unrecoverable error (DUE)*. SDCs, even at an extremely small

degree, can lead to catastrophic outcomes. Several prior works have researched into minimizing the level of SDCs in modern computing systems. This dissertation also minimizes the number of bits exposed to soft errors by a program, which directly translates into a reduced number of SDCs the program encounters. DUEs are less severe, as they either alert the user that the outcome is erroneous or simply abort the execution. Therefore, DUEs caused by soft errors can be corrected by re-executing the application.

2.3 ACE Analysis

Architecturally Correct Execution (ACE) analysis was proposed by Mukherjee et el. [137] to assess soft error vulnerability of a program. ACE analysis builds on the key idea that not all faults in a hardware structure — for example, a processor — affect the final program outcome. Under the same circuit technology and environment conditions, all structures of a pipeline are equally vulnerable to soft errors, as depicted by the raw error rate or intrinsic fault-rate of the structure. However, the impact of a fault on the hardware structure may not necessarily corrupt the program output. Therefore, providing equal protection against soft errors to all structures is not required. For example, a fault in the branch predictor can never lead to incorrect updates to the architectural state of a program. A branch predictor is only effective in improving the performance of an application. Therefore, no protection is required for the branch predictor. In contrast, the Program Counter (PC) must be protected, as the address of the next instruction to be executed is stored in the PC. Therefore, a fault in the PC will always lead to incorrect program execution. ACE analysis helps guide designers add varying degrees of protection to different pipeline structures depending on their vulnerability to soft errors.

ACE bit. A processor bit is defined as an *ACE bit* if the bit will cause an error during program execution when flipped, affecting user-visible state (program crash or wrong output). A bit that is not ACE is an *un-ACE bit*. All ACE bits must be correct for correct execution of a program on a processor. For example, all bits of the PC are ACE, and all bits of the branch predictor are un-ACE. A processor bit can be ACE in one cycle and un-ACE in another cycle. For example, the bits written by an instruction in a physical register are ACE before the instruction commits. However, once the (physical) register value has been propagated to the architectural state of the program upon commit, the physical register bits are no longer required, and are thus un-ACE. Overall, in a given cycle, a processor bit is either ACE or un-ACE.

ACE cycles. The ACE cycles for a bit are the total number of cycles the bit is ACE.

ACE Bit Count (ABC). ABC for a program running on a processor is the sum of the ACE cycles for all processor bits for the execution of the program. For a program running on a processor with N bits, ABC is expressed as:

$$ABC = \sum_{i=1}^{N} ACE_i \tag{2.2}$$

where ACE_i represents the ACE cycles for bit i.

Architectural Vulnerability Factor (AVF). AVF [137] is defined as the fraction of ACE bits to the total number of bits in a structure, core or the whole processor. For a processor bit, AVF is defined as the fraction of total cycles the bit was ACE. For example, if a program executes for 1 Billion cycles and a bit was ACE for only 500 Million cycles, the AVF of the bit equals 0.5 or 50%. AVF of the branch predictor is 0% and the AVF of the PC is 100%. For a complete execution, AVF of a program can be expressed as:

$$AVF = \frac{ABC}{N \times T} \tag{2.3}$$

with T the execution time of the program and N the total number of bits in the processor. Obviously, AVF is application-dependent, as some applications occupy more or fewer entries in the core structures, and/or have more or fewer wrong-path instructions. Alternatively, AVF represents the fraction of processor bits exposed by correct-path instruction of a program. A 35% AVF of a program implies that on average, 35% of the pipeline is occupied with correct-path state for the entire execution of a program.

Soft Error Rate (SER). SER is defined as the total number of errors on ACE bits encountered by a program per unit of time. SER is computed as follows:

$$SER = \frac{ABC}{T} \times IFR. \tag{2.4}$$

From Equation 2.4, we conclude that for the same execution time, circuit technology and environment, SER of a program is directly proportional to ABC. Therefore, a 10% decrease in ABC leads to a 10% reduction in the number of soft errors, assuming that execution time does not change.

ACE analysis is performed using a model of a hardware structure, for example, a processor simulator, and the accuracy of estimated AVF depends on the level of detail incorporated in the model [23]. The more details, the longer it takes to estimate the AVF. Nevertheless, ACE analysis is extremely fast and the guidelines from the early reliability estimates can improve the robustness of the hardware structure. Another benefit of ACE analysis is that the vulnerability estimation requires only a few runs of the model. We augment Sniper [32], a state-of-the-art processor simulator with ACE bit counters for evaluating novel techniques proposed in this dissertation for improving reliability. Section 3.4 explains the counter mechanism in detail.

2.4 Fault Injection

The architectural vulnerability factor can also be calculated by performing fault injection campaigns. Fault injection involves altering a (hardware) bit and comparing the outcome of a program against a fault-free outcome. If there is a mismatch between the two outcomes, the correctness of the hardware bit is a must for correct program execution; otherwise, the fault is masked. The architectural vulnerability factor of a program running on the hardware structure is the fraction of the total number of mismatched outcomes to the total number of fault injections.

Fault injection can be performed either at the hardware level or at the software level [128], which we discuss in more detail now.

2.4.1 Hardware-Level Fault Injection

Hardware-level fault injection is performed by using an external device to inject faults in the target hardware, for example, a processor or memory. The injection can be performed at the pin level [10, 11, 181]; another approach is to expose the target processor to radiation [64, 73, 156, 157] or electromagnetic interference [100]. Exposure to radiation, through for example a neutron beam, is helpful in understanding the vulnerability of an existing processor chip. Additionally, the entire vulnerability stack shown in Figure 2.2 is exposed to the radiation. Therefore, hardware-level fault injection provides the most accurate reliability estimation. However, the chip must be designed before performing the injection, and this is already too late in the design cycle of a processor chip to impact its design. However, the feedback from the hardware-level fault injection may be extremely helpful for improving the reliability of future designs. The cost associated with the experimental setup to inject faults is typically very high and the hardware-level fault injection can also damage the chip under test.

2.4.2 Software-Level Fault Injection

Software-level fault injection is widely used to assess the vulnerability of programs running on a hardware. The program under test can be a user-level application or system software such as the OS. As the name suggests, the faults are injected in the software itself, either at compile-time or at runtime. At compile time, the code can be modified to inject faults at specific locations; the fault gets triggered when the particular location is executed. At runtime, a fault injector can be invoked when the program under test meets certain conditions — for example, when accessing a particular memory location or writing to a particular architectural register. Fault injection campaigns can be launched at the architecture level, microarchitecture level, or Register-Transfer Level (RTL) model (or simulator) of a processor. The feedback from software-level fault injection can be included in improving the robustness of the chip, and therefore, unlike hardware-level fault injection, software-level fault injection is

helpful in the early reliability assessment of a hardware structure. Several tools have been proposed to estimate the robustness using software-level fault injection [34, 36, 61, 78, 99, 117, 176, 204, 218].

A massive injection campaign needs to be performed before the vulnerability of a program can be quantified with a certain degree of statistical confidence [116]. Therefore, software-based fault injection is a slow process, and the time to perform fault injection increases with increasing details of the underlying model [37]. For example, injecting faults in an RTL-level simulator requires an order of magnitude more time than an architecture-level simulator [43]. Several researchers have proposed techniques to speed up the fault injection campaign without significantly degrading its accuracy [98, 167]. Contrary to the hardware-level fault injection that can inject faults even at the level of silicon, the lowest accessible hardware structures by software-level fault injection are architectural registers, memory addresses, and data values. The process of reliability assessment using software-level fault injection is inexpensive, repeatable, and it does not damage the device under test.

2.5 General-Purpose Processor Architecture

In this section, we provide a brief overview of the functionality of superscalar in-order and out-of-order (OoO) processors. Modern processor pipelines can have up to 20 stages, however, the functionality of the pipeline can be broadly divided into five stages: fetch, decode, execute, memory access and commit. A *scalar* pipeline executes *at most* one instruction every cycle; a *superscalar* pipeline can execute more than one instruction every cycle. An *n-way* superscalar pipeline executes up to n instructions every cycle. The fetch and decode stages make up the processor *front-end*, while the other stages constitute the *back-end*. *Commit* is the last stage of the pipeline that releases core microarchitectural resources, and updates the architectural state of a program. The instructions are always fetched and committed in program order. Some pipeline stages can handle slightly more instructions than others. For example, front-end can be wider than the back-end, to ensure an uninterrupted supply of instructions to the back-end. Similarly, depending on the readiness of the available instructions, the issue stage can issue more than n instructions every cycle in an n-way superscalar processor. Both in-order and OoO processors can be superscalar.

2.5.1 Processor Front-End

The functionality of the fetch and decode stages of the pipeline is similar for both in-order and OoO cores. However, the fetch and decode stages of the in-order core are typically less complex than the OoO core. In addition, an out-of-order core also performs register renaming in the processor front-end before sending instructions to the back-end for execution.

instruction-id	instruction	data dependence
I1	add r1 ← r2, r3	
I2	mul r2 ← r3, r4	WAR I1(r2)
I3	ld r1 ← mem[x]	WAW I1(r1)
I4	add r2 ← r1, r3	RAW I3(r1), WAW I2(r2)
I5	ld r5 ← mem[y]	
I6	sub r6 ← r4, r5	RAW I5(r5)
I7	ld r6 ← mem[z]	WAW I6(r6)

Figure 2.4: An example code sequence with RAW, WAR and WAW data dependences. `RAW I(r)` means current instruction has RAW hazard with instruction `I` through architectural register `r`.

The fetch stage of the processor pipeline retrieves program instructions from the L1 instruction cache (L1 I-cache); modern processors typically fetch more than one instruction every cycle from the I-Cache [62]. Branch instructions can change the control flow based on the outcome of a branch, which is not known until the branch is executed. Therefore, branch prediction is also performed in the fetch stage of the pipeline and the next address to fetch from is the (predicted) branch target address in the program address space.

The decode stage of the pipeline is responsible for understanding the meaning of the instructions embedded in the incoming stream of bytes from the fetch unit. This stage of the pipeline also breaks instructions into micro-ops[1]. Depending on the types of instructions supported by the Instruction-Set Architecture (ISA) implemented by the processor, there can be multiple decoders of different complexity designed in the decode stage. Furthermore, both the fetch and decode stages can further be divided into multiple pipeline stages; for example, the IBM POWER8 processor has six fetch and five decode stages [183]. There can be several optimizations such as micro-op cache, loop-stream detection, and a micro-op queue implemented in the processor front-end to improve the flow of micro-ops to the processor back-end [201].

2.5.2 Pipeline Hazards

A *control dependence* happens when the flow of execution (or PC) changes due to execution of certain instructions like branches or exceptions. A *data dependence* occurs when there is a data dependency between two instructions, either through (architectural) registers or through memory. There are three types of data dependences: *read after write (RAW)*, *write after read (WAR)* and *write after write (WAW)*. Figure 2.4 shows an example of these data dependences. RAW is the only true dependence as WAR and WAW can be eliminated through register renaming, as discussed in Section 2.5.4. A pipeline *hazard* is

[1]We use the terms instruction and micro-op interchangeably in this work and the meaning should be clear from the context.

a situation that prevents the next instruction in the instruction stream from executing during its designated clock cycle [84]. Pipeline hazards cause performance bottlenecks. A *structural hazard* occurs when the appropriate hardware resource, for example an execution unit, to process an instruction is not available in a given cycle.

2.5.3 In-order Cores

In an in-order core, the instructions are issued to the execution units in program order. A *scoreboard* keeps track of the data dependences among instructions, and the availability of resources to execute the instructions. An instruction is issued only when all its operands are ready and the required functional unit is available. The dependent instruction must wait in case of a RAW hazard; for example, in Figure 2.4, instruction `I4` cannot be issued until `I3` finishes execution and updates register `r1`. Since instructions are issued in order, there is no WAR hazard because the previous instruction finishes reading the register value before the next instruction writes to it. As an example, `I2` writes to `r2` only *after* `I1` has read register `r2`. In case of a WAW hazard, as with instructions `I2` and I4 on register `r2`, `I4` cannot write to `r2` until `I2` has finished writing to `r2`. Register `r2` is *busy* until instruction `I2` has written to it.

Depending on their types, instructions can take varying number of cycles to finish execution. Therefore, the number of cycles an instruction waits is decided by the latency of the instruction it depends on. For example, if the load instruction (`I3`) takes 200 cycles to read data from memory, `I4` cannot be issued for 200 cycles after `I3` was issued. Therefore, although the following instructions `I5` and `I6` do not depend on either `I3` or `I4`, `I4` *blocks* the head of the scoreboard and no instruction can be issued. The in-order issue policy is the main performance *bottleneck* of in-order cores.

2.5.4 Out-of-Order Cores

Modern OoO microarchitecture such as Intel's Skylake and IBM's POWER8 are complex with up to twenty pipeline stages [54, 183]. There are five key operations performed within an OoO core.

1. **Register renaming:** Register renaming eliminates false register dependences — that is, WAR and WAW register dependences — among instructions by mapping each architectural register to a unique physical register. Every destination architectural register of an instruction is allocated a new physical register; the source architectural registers can simply be mapped to the already renamed physical register. Figure 2.5 shows the instruction sequence from Figure 2.4 after renaming. The mapping from architectural registers to physical registers is maintained in a table commonly known as a Register Alias Table (RAT). The number of

instruction-id	instruction	data dependence
I1	add P1 ← P2, P3	
I2	mul P5 ← P3, P4	
I3	ld P6 ← mem[x]	
I4	add P7 ← P6, P3	RAW I3(P6)
I5	ld P8 ← mem[y]	
I6	sub P9 ← P4, P8	RAW I5(P8)
I7	ld P10 ← mem[z]	

Figure 2.5: True data hazards (RAW) after renaming the code sequence shown in Figure 2.4. Pi refers to the physical register i.

physical registers designed in an OoO core is significantly larger than the number of architectural registers. A RAW dependence is the only data dependence after register renaming. Therefore, although the instructions are fetched and committed in program order, the execution within the OoO core — the selection of instructions to the functional units for execution — is performed in a data-flow manner and thus possibly out of program order.

2. **In-order dispatch:** The *dispatch* stage of an OoO pipeline receives renamed instructions in program order, and allocates them back-end resources in the *same* order. Four hardware structures forming the backbone of an OoO core are the reorder buffer (ROB), issue queue, load queue, and store queue. All instructions allocate a new entry in the ROB and issue queue. A load instruction also allocates an entry in the load queue; similarly, a store instruction allocates an entry in the store queue. Allocating load and store queue entries in program order is critical for dealing with memory disambiguation, as we will discuss later.

3. **Out-of-order issue:** The OoO issue queue resolves the (in-order) issue bottleneck of the in-order core. Any instruction, irrespective of its order among other instructions in the program, can execute when its source operands are ready and the appropriate functional unit is available, leading to *out-of-order* execution. Therefore, the younger ready instructions can execute before older non-ready instructions; for example, `I5` and `I7` (see Figure 2.5) can now execute before `I4`. The execution of instructions potentially overlaps due to out-of-order issue, increasing the degree of *instruction-level parallelism (ILP)*. Increased ILP also increases *memory-level parallelism (MLP)*, which is defined as the number of outstanding requests to memory when at least one request is outstanding [45].

4. **Handling branch mispredictions and exceptions:** A younger instruction can execute before an older instruction due to out-of-order execution. In case the older instruction is a branch that is not resolved yet, the instructions younger to the branch instruction are termed as *speculative*. If the branch turns out to be mispredicted upon resolution (or

execution), all the speculative instructions need to be flushed from the pipeline. Similarly, instructions after a load instruction that causes a page-fault must be flushed from the pipeline. Overall, the *architectural state* of an executing program, which consists of the architectural register file (ARF) and memory, must not reflect updates from the speculative instructions; therefore, an instruction after a mispredicted branch or exception must not propagate its update to the ARF and memory. All branch mispredictions, exceptions and interrupts must be handled *precisely* — that is, when they commit, *all* instructions prior to them must update the architectural state of the program and *no* instruction after them updates the architectural state [185].

This sequential model of program execution where an instruction only completes after all prior instructions have completed, is realized with the help of the reorder buffer (ROB) in an OoO processor. The ROB maintains entries of dispatched instructions in program order, and the instructions are committed in the same order. An instruction updates the architectural state of the program only when it commits, and an instruction commits only when it is the oldest instruction in the ROB. Therefore, a younger instruction can never update the architectural state before an older instruction. In processors such as Intel Pentium Pro, when a branch instruction reaching the head of the ROB turns out to be mispredicted, all instructions *after* the mispredicted branch are flushed from the pipeline and the front-end is redirected to fetch from the branch target address [70]. Waiting for a (mispredicted) branch to reach the head of the ROB significantly degrades application performance. Therefore, modern processors instead maintain a log of the rename table updates as the instructions are renamed in program order. When a branch is mispredicted, the log is traversed backwards to restore the rename table to state before the branch instruction. Processors such as MIPS R10000 or Alpha 21264 maintain periodic checkpoints to reduce the traversal time through the log in the event of a branch misprediction [70].

To ensure that the instruction causing an exception is not speculative, exceptions are typically handled at the commit stage of the pipeline. For example, when a load instruction causes an exception, such as a page-fault, the instructions following the load are flushed from the pipeline and the processor begins executing the corresponding fault handler (for example, a page fault handler or TLB miss handler.)

5. **Memory disambiguation:** The load queue, store queue and store buffer are the three microarchitectural structures designed to efficiently handle memory operations in an out-of-order processor. The load queue and store queue maintain entries for load and store instructions in program order. The store queue also holds the data values for the executed store instructions. When a store instruction commits, its associated data value from the store queue is moved to the store buffer. The data in the store buffer is part of the architectural state of the program, and the data

is written to the memory hierarchy when a store instruction eventually *retires*.

In a manner similar to dependence through (architectural) registers, load and store instructions can also have RAW, WAR and WAW dependences through a common memory address. WAR and WAW dependences are automatically resolved because the load and store instructions are committed in program order. The processor must handle the RAW dependence between a load instruction and an earlier store instruction to ensure that the load instruction reads most up-to-date data value. Memory disambiguation is knowing with certainty whether the load and store instructions will access the same memory location [175]. The memory location can only be known when the address of a load or store instruction is calculated by the address generation unit; otherwise, the load or store is termed as *ambiguous*. The fundamental problem in determining the RAW dependence between a load instruction and an earlier store instruction is that the memory addresses used by load or store instructions are not known until their execution.

A naive solution to handle RAW dependences between a load and (prior) store instructions would be to execute all load and store instructions in program order. Therefore, execute a load instruction if there are no prior store instructions writing to the same memory address as the load instruction. The load instruction does not execute if there is a prior store instruction with an unresolved address. This requires checking the status of all prior store instructions in the store queue. In case of an address match, the data value has to be transferred from the store queue/buffer to the load instruction. When there is no address match, the load instruction can safely access the data value from the L1 D-cache. The in-order execution of load and store instructions ensures that a load instruction always accesses the most up-to-date data value.

However, since load instructions are typically on the critical path of a program execution, delaying the execution of a load instruction until all prior store addresses are known can adversely impact the performance of an application running on an out-of-order processor. More importantly, the majority of load instructions in a program do not depend on prior store instructions. Therefore, modern processors allow load instructions to execute out-of-order as soon as their input operands are available. A load instruction accesses the store queue, store buffer and L1 D-cache in parallel. If there is a hit in the store queue or store buffer, the data is forwarded from the youngest store instruction to the load instruction; otherwise, the data is accessed from the L1 D-cache [6]. The address and data values associated with an executed load instruction are buffered in the load queue. When a store instruction executes, it checks the load queue for a potential address conflict with a younger load instruction. In case of an address match between the store instruction and a younger load instruction, the load instruction needs to update its data value. Therefore, the load instruction, and its dependent instructions, are re-

executed. *Memory dependence prediction* is a well-known technique to limit the cost of such a re-execution in out-of-order processors.

2.6 Summary

Radiation-induced soft errors are caused either by neutron particles from cosmic rays or alpha particles present in the packaging material of a chip. These energy particles cause faults in the hardware which can be masked for various reasons. ACE analysis and fault injection are two well-established methods to evaluate the architectural vulnerability factor in modern processors. Both techniques offer different trade-offs in terms of accuracy and speed.

In-order cores are less complex and frequently stall on memory accesses while out-of-order cores employ sophisticated techniques such as register renaming and memory disambiguation to extract high degrees of instruction-level parallelism and memory-level parallelism. The concepts presented in this chapter will be helpful in understanding the detailed microarchitecture work conducted in the following chapters.

Chapter 3

Reliability-Aware Scheduling

Heterogeneous chip-multiprocessors (HCMP) [112, 113] were recently introduced to improve performance in an energy-efficient way. The presence of multiple core types, e.g., big high-performance cores and small energy-efficient cores, allows flexibility in the power-performance balance of a processor: if high performance is needed, an application can use a big core, but if energy efficiency is the main objective, it can run on a small core. An HCMP scheduler should also take into account the characteristics of the applications. Some applications show almost no performance improvement on the big core compared to the in-order core, whereas others benefit from higher performance gains. By intelligently selecting the applications that run on the big core, performance can be optimized [112, 207], potentially within a given power limit [1, 2, 138, 221].

In this chapter, we perform a detailed analysis of the performance and soft error reliability of the benchmarks from the SPEC CPU2006 suite on an HCMP. We show that the vulnerability on a core cannot be directly correlated to any application characteristic, and we observe that there is a clear opportunity for improving the overall system reliability by migrating applications between different core types without significant impact on performance. We begin in Section 3.1 by estimating reliability and performance on big and small cores, and show that there is significant potential for reliability-aware scheduling. We note the lack of a reliability metric to assess the vulnerability of a multiprogram workload on a multicore processor. Therefore, in Section 3.2, we propose the *system soft error rate (SSER)* metric for quantifying the soft error rate of multiprogram workloads. In Section 3.3, we then describe the design and implementation of our reliability-aware scheduler. The hardware overhead of the proposed scheduler is estimated in Section 3.4. A detailed evaluation of the proposed reliability-aware scheduler on reliability and performance is performed in Section 3.6.

We increase the robustness of reliability-aware scheduling by extending it in several directions. In Section 3.7, we show the impact of reliability-aware scheduling on power. We also design and implement a power-optimized scheduler and compare reliability-, performance-, and power-optimized scheduling policies in an HCMP. Section 3.8 analyzes the improvement in reliability under performance constraints. For certain systems, performance degradation below a predetermined threshold is not permitted, and therefore, optimization for reliability must be performed within the performance limit. We replace the multiprogram workloads by multithreaded workloads in Section 3.9. Multithreaded workloads have entirely different characteristics when compared to multiprogram workloads, as the threads from a multithreaded workload need synchronization among them. In Section 3.10, we extend our analysis of reliability-aware scheduling to include ACE bits exposed by an application in on-chip L1 caches in addition to the core.

3.1 Reliability in an HCMP

In this section, we first analyze the difference in vulnerability to soft errors across core types, and then show the potential for reliability-aware scheduling using an offline oracle approach. As explained in Section 2.3, all ACE bits must not encounter any soft error during a program execution, and AVF is the fraction of ACE bits to the total bits exposed by a program on a processor. We assume each bit in the processor pipeline holding state of a correct-path and non-NOP instruction to be ACE; i.e., all bits in the issue queue, load/store queue, reorder buffer, physical register file, and functional unit holding state of a correct-path, non-NOP instruction are considered ACE. Structures that improve performance but do not affect functional correctness (e.g, a branch predictor) do not contain any ACE bits.

3.1.1 Reliability versus Core Type

It is commonly known that different core types in a heterogeneous multicore processor exhibit different performance and power characteristics. However, different core types also exhibit differences in reliability, leading to an opportunity for improving reliability through scheduling.

There are basically three contributors to the reliability of an application running on a core: (1) processor design, (2) application characteristics, and (3) application performance on the core. We elaborate upon these points below.

Processor Design. The size of the core structures that hold architecture state and that are required to guarantee functional correctness affects reliability. In an in-order processor, these include register file, functional units, and scoreboard. In an out-of-order processor, the microarchitectural structures are register file, functional units, issue queue, load queue, store queue, and the

reorder buffer. The larger these structures are, the higher the probability for an error in those structures.

Application Characteristics. The application characteristics also determine its reliability — this is decided by the fraction of the architecturally relevant structures that an application occupies, i.e., AVF. Some applications can occupy a small fraction of the core structures while others may keep the pipeline mostly occupied with correct-path state. Applications occupying only a small fraction of these structures can have a lot of non-architecturally relevant instructions (NOPs, wrong-path instructions). The smaller the occupied fraction is, the smaller is the error probability.

Application Performance. The performance of an application on a particular core type decides the duration for which the application state will be exposed to soft errors. If an application executes faster, it will finish sooner, and therefore it will be less vulnerable to errors. On the contrary, if the application takes longer to finish, the exposure to soft-error increases, which leads to a higher chance for an error.

Now consider a big out-of-order core and a small in-order core in an HCMP. Obviously, the big core has larger structures than the small core. As a result, a big core is likely to expose more vulnerable state than an in-order core. However, the degree of vulnerability also depends on structure occupancy which is a function of the application and its interaction with the underlying hardware.

3.1.2 Application Sensitivity

Applications exhibit varying degrees of sensitivity to soft error vulnerability. AVF is an insightful metric to understand an application's vulnerability to soft errors. Figure 3.1 shows AVF for the SPEC CPU2006 benchmarks on a big out-of-order core as well as a small in-order cores. (See Section 3.5 for details regarding our experimental setup.) The benchmarks are sorted by their AVF on the big core: `gobmk` has the lowest AVF at 8% while `zeusmp` has the highest AVF of 41%. AVF accounts for all the ACE bits in the processor during the entire execution. In particular, if an ACE instruction occupies 64 bits in the reorder buffer (ROB) for 16 cycles, this amounts to 1024 ACE bits. This way of measuring incorporates structure size, occupancy and execution time. As expected, AVF is higher for the big out-of-order core compared to the small in-order core; this is because a big core holds more architecture state. Microarchitectural structures of the small in-order core do not hold ACE bits during execution for a long duration. For example, the physical register file consisting of 16 integer and 16 floating-point registers accounts for more than half of the total microarchitectural state of the small core (see Table 3.3). However, the physical register values are ACE only between execute and commit stages of the pipeline. Instructions occupy issue queue for a significantly longer duration than the register file due to the in order issue policy of the small core. Note however that in spite of the fact that AVF is higher on the small core than the

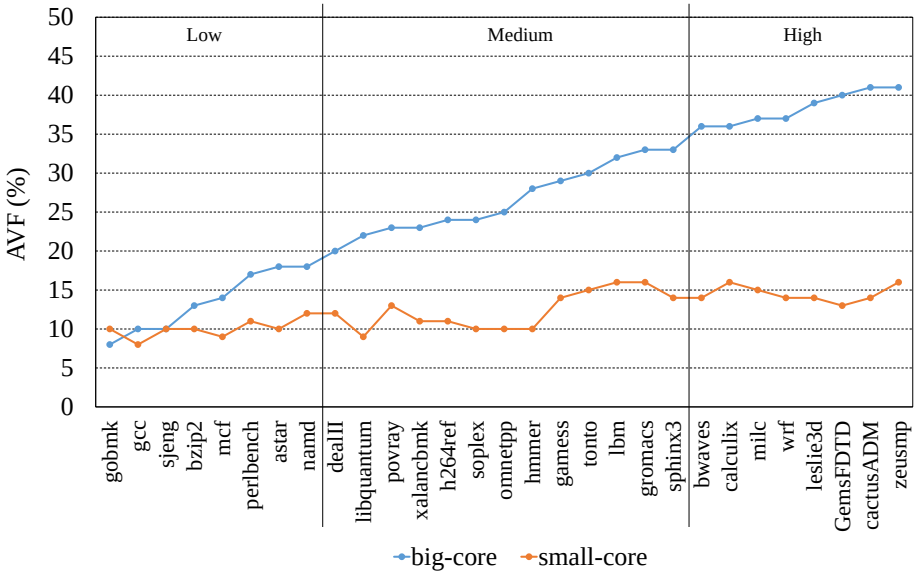

Figure 3.1: AVF for the SPEC CPU2006 benchmarks on a big out-of-order and a small in-order cores. The benchmarks are sorted by their AVF on the big core.

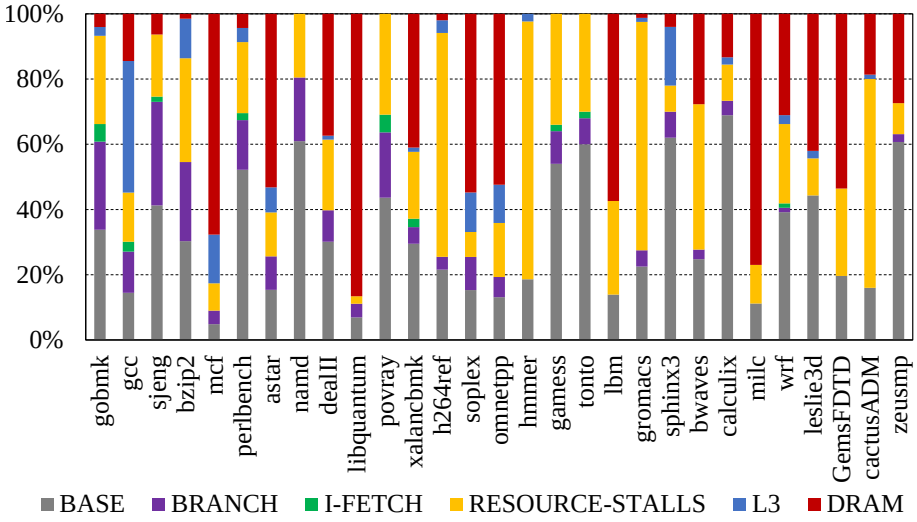

Figure 3.2: Normalized CPI stacks for the SPEC CPU2006 benchmarks on a big out-of-order core.

big core for the left-most benchmark, `gobmk`, it is still less vulnerable to soft errors on the small core because of the smaller structure size, i.e., N is smaller.

The applications appearing on the right-hand side of the graph are most sensitive to reliability-aware scheduling, i.e., when scheduled on the big core, AVF

(and thus SER) increases significantly compared to running on the small core. Applications appearing on the left-hand size are less sensitive, i.e., the increase in SER on the big core is not as high, and thus if given the choice, scheduling these applications on a big core rather than a small core will not increase overall system soft error rate as much. Figure 3.1 classifies the benchmarks into three categories based on their big-core AVF: high, medium and low. We will use this classification for analyzing the performance of our reliability-aware scheduler across workload types in the evaluation section.

It is interesting to relate the AVF graph to the normalized CPI (cycles per instruction) stacks shown in Figure 3.2. A CPI stack quantifies the fraction of cycles spent doing useful work (i.e., the base component) plus a number of adders or components to represent 'lost' cycles because of resource stalls, branch mispredictions, instruction cache misses, last-level cache (LLC) misses and main memory accesses. The 'resource-stalls' component in Figure 3.2 represents the sum of the time waiting on a full issue queue, and the time spent in accessing L1-D and L2 caches. Note that the benchmarks are ordered the same way as in Figure 3.1. The benchmarks on the left-hand side exhibit low AVF primarily because of their relatively high front-end miss components. Front-end miss events, such as branch mispredictions and instruction cache misses, cause the pipeline to be drained and hence there is relatively little vulnerable state in the processor. The benchmarks on the right-hand side on the other hand have a high AVF because they exhibit high occupancy in various back-end structures of the pipeline for a variety of reasons. Some benchmarks (e.g., `milc`) are memory-intensive: a load operation accessing main memory typically blocks the head of the reorder buffer, which causes the ROB to fill up, and which leads to significant ACE state while servicing the memory operation. Other high-AVF benchmarks (e.g., `zeusmp`) are compute-intensive: high IPC and high MLP is achieved by having high occupancy in various back-end queues. Yet other benchmarks experience resource stalls in the back-end structures because of L1 data cache misses, L2 cache misses, limited ILP (i.e., chains of dependent instructions) which cause the ROB and issue queues to fill up with instructions.

Note that there are a number of memory-intensive benchmarks (e.g., `mcf` and `libquantum`) that exhibit low AVF. This is because these benchmarks suffer from branch mispredictions which lead to a large number of un-ACE wrong-path instructions in the ROB underneath memory accesses. Although `mcf` and `libquantum` are more memory-intensive than other memory-intensive benchmarks like `leslie3d` and `milc` [79], their AVF is extremely low when compared to these memory-intensive benchmarks. Similarly, `calculix` and `povray` are the most compute-intensive benchmarks in the entire suite [79]. However, there is a large difference in their soft error reliability on a big out-of-order core.

The take-away message from this analysis is that there exists no simple workload characteristic (e.g., compute-intensive versus memory-intensive) to determine how sensitive a workload is with respect to reliability. Instead, it depends on how AVF-intensive an application is, which is a result of complex interactions among various workload characteristics and the underlying

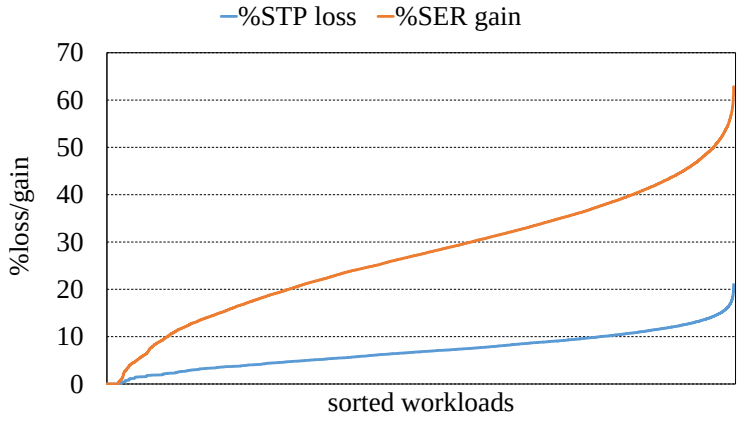

Figure 3.3: Percentage STP loss and SER gain for an oracle reliability-optimized scheduler relative to a performance-optimized scheduler for four-program workloads on an HCMP with two big cores and two small cores.

microarchitecture. This suggests that reliability-aware scheduling needs a dynamic mechanism to monitor an application's reliability on either core type in a heterogeneous multicore and adjust the schedule accordingly.

3.1.3 Oracle Reliability-Aware Scheduling

To quantify the potential of reliability-aware scheduling, we perform the following experiment. We simulate each application on both core types, and record performance and SER. We then consider all combinations of four applications on a heterogeneous multicore processor with two big and two small cores. The total number of such four-program combinations equals $\binom{29}{4} = 23,751$. For a four-program workload, there are six possible ways to schedule it on two big and two small cores: `BBSS`, `SSBB`, `BSBS`, `SBSB`, `BSSB`, `SBBS`; a `B` represents a big core and a `S` represents a small core. Of the six possible schedules, we select the one with the highest performance (expressed in system throughput (STP) [60]), and the one with the lowest total SER. (See the next section for the metric we use to quantify SER for a multiprogram workload.) We assume no interference in shared resources, and consider the performance and SER numbers from the isolated experiments. This leads to an oracle offline schedule. Figure 3.3 shows SER reduction and performance loss for the SER-optimized schedule normalized to the performance-optimized schedule. Clearly, the reduction in SER is much higher than the loss in performance, resulting in an average 27.2% reduction in SER (and up to 62.8%) while degrading performance by 7% on average. This result demonstrates the significant potential and motivates our study on reliability-aware scheduling for heterogeneous multicore processors.

	Time	ABC	SER
Big core	10	60	6
Small core	15	30	2

Table 3.1: Execution time, ABC and SER for a hypothetical benchmark on big and small cores.

3.2 Reliability Metric for Multiprogram Workloads

This section reviews existing reliability metrics and then proposes a novel soft error reliability metric for multiprogram workloads.

3.2.1 Single-Program Workloads

For a single-program workload, reliability is commonly quantified using soft error rate (SER), i.e., the number of errors per unit of time. The SER is defined as:

$$SER = \frac{ABC}{T} \times IFR, \tag{3.1}$$

with ABC defined as the total ACE bit count over the entire execution of a program. In other words, SER computes the number of ACE bits per unit of time multiplied by the intrinsic fault rate. As long as we measure SER for a single-program workload by running (a well-defined section of) the workload to completion, we can safely evaluate reliability using SER because the unit of work is constant.

3.2.2 Multiprogram Workloads

SER breaks down for multiprogram workloads. We cannot simply add up SER numbers for each of the applications in a multiprogram workload because some applications are inherently more vulnerable to soft errors than others — adding raw SER numbers would give too much weight to fast-running applications and too little weight to slow-running applications. This is similar to performance metrics for multiprogram workloads, i.e., adding plain IPC numbers gives more weight to high-IPC applications. The fundamental problem here is that SER does not take into account the impact of performance on the error rate: lower performance makes the application run longer, increasing the probability for an error during its execution.

SER falling short for multiprogram workloads is illustrated with an example in Table 3.1. The execution time and ABC are given for a workload running on big and small cores. For this hypothetical example, the ratio of the SER on big core versus small core is 3, which suggests that the big core is three times

less reliable compared to the small core; we assume same IFR on the big and small cores. However, if we look at the ratio of the ABC on the big and small cores — which truly reflects the amount of correct-path state exposed by the application on the two core types — the big core is twice as less reliable than the small core. Hence, SER favors small cores (more than what they actually should be.) Therefore, it is crucial to include the slowdown caused by different core types while estimating reliability on a heterogeneous multicore processor. In the following section, we propose a novel metric that includes the impact of performance on the reliability assessment on a (heterogeneous) multicore processor.

3.2.3 System Soft Error Rate

To include the impact of performance on SER, we weight per-application SER with the slowdown incurred because of multiprogram execution. Application slowdown is defined as the execution time of an application on the (heterogeneous) multicore divided by its execution time on a reference machine (e.g., an isolated big core). A slowdown of 1 means that the application executes equally fast as on the reference machine; a slowdown of 2 means that the application takes twice as long under multiprogram execution compared to isolated execution. We then define *weighted SER (wSER)* of an application in a multipropgram workload as follows:

$$wSER = \frac{ABC}{T} \cdot \frac{T}{T_{ref}} \cdot IFR = \frac{ABC}{T_{ref}} \cdot IFR, \tag{3.2}$$

with ABC and T the ABC and execution time of the application in the multiprogram workload, respectively; and T_{ref} the execution time of the application on an isolated reference core (e.g., a big core in a heterogeneous multicore). In other words, wSER weights the application's SER during multiprogram execution with its slowdown compared to isolated execution. This is to account for the fact that if the application runs longer during multiprogram execution (which is what you would expect because of interference in shared resources), it gets exposed to soft errors for a longer duration.

Summing the weighted SER values for the individual applications in a multiprogram workload then yields our novel *system soft error rate (SSER)* metric:

$$SSER = \sum_{i=1}^{n} wSER_i = \sum_{i=1}^{n} \frac{ABC_i}{T_{i,ref}} \cdot IFR, \tag{3.3}$$

which quantifies the total weighted SER across all the applications in the multiprogram workload. SSER gives a higher weight to slow-running applications in the multiprogram workload mix, and a smaller weight to fast-running applications. This is to account for the fact that slow-running applications will be exposed to soft errors for a longer duration, hence we scale their per-application SER proportionally with their relative slowdown.

(a) homogeneous multicore: SSER=2			
	SER	*slowdown*	*wSER*
benchmark A on big	1	1	1
benchmark B on big	1	1	1
(b) homogeneous multicore: SSER=3			
	SER	*slowdown*	*wSER*
benchmark A on big	1	2	2
benchmark B on big	1	1	1
(c) heterogeneous multicore: SSER=1.5			
	SER	*slowdown*	*wSER*
benchmark A on small	1/8	4	0.5
benchmark B on big	1	1	1

Table 3.2: Examples illustrating the SSER metric.

3.2.4 Illustrative Examples

We now illustrate the intuitive and system-level meaning of SSER using a couple examples, see also Table 3.2. Consider a homogeneous multicore with two big cores, and assume that the two co-running applications do not interfere with each other, i.e., they both run equally fast on the homogeneous multicore compared to isolated core execution — example (a) in Table 3.2. Assume further that per-application SER is not affected by multiprogram execution. SSER equals 2 in this case, which makes perfect sense: the system's vulnerability is twice as high on the homogeneous multicore compared to isolated execution because we now have two co-running applications.

Assume now that one application slows down by a factor of 2 (e.g., because of hardware interference) and the other application is not affected at all — example (b) in Table 3.2. In this case, SSER equals 3, i.e., a weighted SER of 1 for the application that does not slow down, plus a weighted SER of 2 for the application that slows down by a factor two. This makes intuitive sense because it takes two times as long for the slow application to get the same amount of work done, and therefore the slow application is twice as vulnerable.

Consider now a heterogeneous multicore — example (c) in Table 3.2. The application that runs on the small core experiences a slowdown of 4 while its SER reduces by a factor of 8 compared to running on the big core. As a result, its weighted SER equals 0.5, i.e., the application is slowed down by a factor of 4 but it is 8 times less vulnerable to soft errors per unit of time, hence it is only half as vulnerable for getting the work done. SSER thus equals 1.5. Note that SSER in example (c) is smaller than for the homogeneous multicore examples (a) and (b); this is due to the fact that even though the benchmark running on the small core slows down substantially, it exposes way fewer ACE bits, which leads to a net reduction in overall system vulnerability.

3.3 Reliability-Aware Scheduling

Having demonstrated the potential for reliability-aware scheduling and having derived the SSER metric for quantifying system-level reliability, we now describe our sampling-based reliability-aware scheduler for heterogeneous multicores. Figure 3.4 shows the sequence of steps followed by the scheduler. We assume that we can measure the performance of each application on each core (e.g., the number of instructions executed during the last scheduler quantum), and the number of ACE bits in each structure (i.e., ACE bit counter or ABC over the past quantum), which we both need to compute SSER. In this section, we explain the working of the scheduling algorithm; we quantify the hardware overhead for measuring ABC later in the Section 3.4.

3.3.1 Scheduling Algorithm

The scheduler starts with an initial sampling phase to collect performance and ABC information for each application on each core type. We assume that the number of applications equals the total number of cores, leading to each application running on a core for every quantum of the execution. If the number of big cores equals the number of small cores, this requires two sampling quanta: putting half of the applications on a big core and half on a small core in the first sampling quantum, and inverting this schedule in the next sampling quantum, i.e., the applications running on a big core are moved to a small core, and vice versa. If the number of big cores differs from the number of small cores, e.g., one big core and three small cores, more quanta are needed to sample each application on each core type (4 sampling quanta in this example). After this initial sampling phase, the scheduler follows the algorithm described in Algorithm 1.

The algorithm first verifies whether the sampled data is recent. If an application has run for 10 consecutive scheduler quanta on the same core type, a sampling phase is triggered: the application is scheduled on the other core type by switching it (during a short sampling quantum) with the application that is running for the most consecutive quanta on the other core type. Like this, the scheduler ensures that the sample data is up-to-date, adapting to potential execution phase changes.

If all applications have recently sampled data for both core types, the scheduler calculates the weighted SER (wSER) for each application if we were to schedule them on the other core type than they are currently scheduled on. It then selects the application with the highest wSER reduction and the application with the smallest wSER increase, and checks whether switching the two applications leads to a net overall SSER reduction. If so, the applications are switched, and the next couple is checked. If no global SSER reduction can be obtained, the current schedule is maintained for the next scheduler quantum. After finishing a quantum, the sample data is automatically updated.

Algorithm 1 Sampling-based reliability-aware scheduler (n is the number of applications.)

```
 1: sampleRequired = false
 2: quantumNumber = 0
 3: lastSampledAt = 0
 4: while true do
 5:    if quantumNumber ≤ 2
       or (quantumNumber − lastSampledAt) == 10 then
 6:       sampleRequired = true
 7:    end if
 8:    if sampleRequired == true then
 9:       startSamplingPhase()
10:       lastSampledAt = quantumNumber
11:       sampleRequired = false
12:       continue
13:    end if
14:    for i = 1 to n do
15:       reduction[i] = getWeightedSERReduction(i)
16:       coreAssigned[i] = false
17:       for j = i + 1 to n do
18:          maxReduction[i,j]= 0
19:       end for
20:    end for
21:    for i = 1 to n do
22:       for j = i + 1 to n do
23:          if coreType[i] == coreType[j] then
24:             continue
25:          end if
26:          if (reduction[i] − reduction[j]) > maxReduction[i,j] then
27:             maxReduction[i,j] =
                reduction[i] - reduction[j]
28:          end if
29:       end for
30:    end for
       {sortedReductions contains an array of n maxReductions[i,j] in their decreasing order}
31:    sortedReductions[n] = sortMaxReductions()
32:    for k = 1 to n do
33:       currReduction[i,j] = sortedReductions[k]
34:       if currReduction[i,j] > 0
          and coreAssigned[i] == false
          and coreAssigned[j] == false then
35:          switchCoreTypes(i,j)
36:          coreAssigned[i]= true
37:          coreAssigned[j]= true
38:       end if
39:    end for
40:    for i = 1 to n do
41:       ABC[i] = getCurrentQuantumABC(i)
42:       IPC[i] = getCurrentQuantumIPC(i)
43:    end for
44:    quantumNumber++
45: end while
```

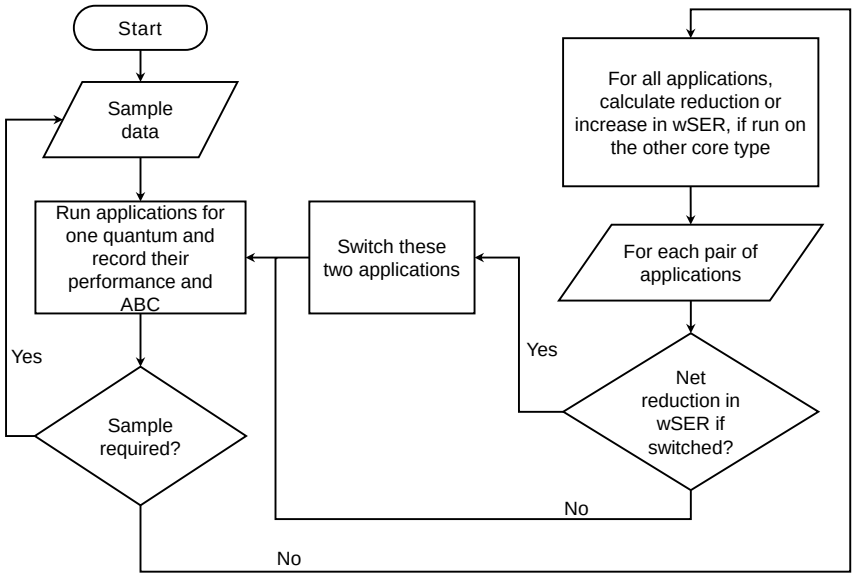

Figure 3.4: Flow chart representing the sequence of steps followed by the reliability-aware scheduler for improving system reliability on a heterogeneous multicore processor.

We need to sample both performance and ABC, because the SSER metric needs both. Sampling ABC requires hardware support to compute occupancy in all relevant processor structures, as we will describe in the next section. Sampling performance can be done by counting the number of instructions executed per quantum – we sample at fixed time quanta (1 ms in our setup). This involves a basic performance counter as is implemented in most recent processors. To compute an application's slowdown, we take the big core as the reference core. Because we have no reference performance data of an isolated big core execution, we assume that the sampled big core performance is a good proxy for reference core performance. Note that the sampled value is subject to interference in the shared resources (e.g., shared cache and memory) because other programs are co-running while sampling.

It is important for a sampling-based scheduler to limit sampling overhead. On the other hand, we need to sample for a sufficiently long period of time in order to obtain stable sampling information. This is why we make a distinction between a sampling quantum and a scheduler quantum. We set the scheduler quantum to 1 ms in all of our experiments, and the sampling quantum to one tenth the scheduler quantum or 0.1 ms. All results in the evaluation section include sampling overhead.

3.3.2 A Two-Program Workload Example

We illustrate the working of reliability-aware scheduler with a two-program workload consisting of `calculix` and `povray`; the time varying reaction of the

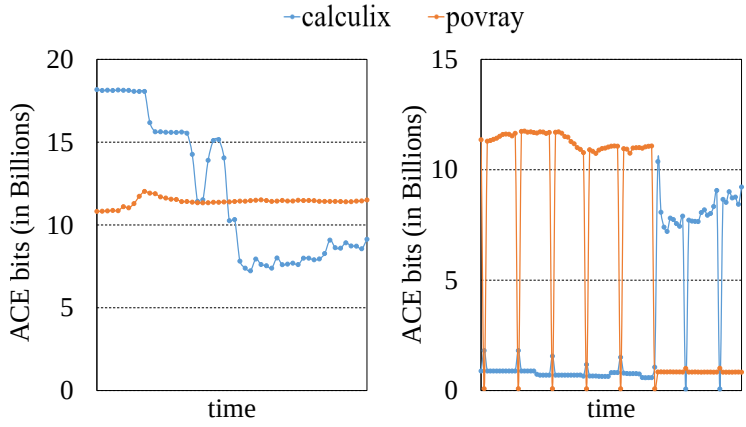

Figure 3.5: ABC over time for `calculix` and `povray` when executed in isolation on a big core (on the left) and as a two-program workload on one big core and one small core under reliability-aware scheduling (on the right).

scheduler for this two-program workload is shown in Figure 3.5. Each dot represents ABC per 1 ms. The left graph shows ABC over time for `calculix` and `povray` when executed in isolation on a big core; the right graph shows ABC when executed concurrently on an HCMP with one big and one small core. When run in isolation, `povray` experiences almost constant ABC; `calculix` on the other hand experiences a big drop in ABC towards the end of its execution. When co-executed on the HCMP, `calculix` is scheduled on the small core initially due to its high big-core ABC compared to `povray`. Upon the phase change in `calculix`, the scheduler reacts by migrating the two applications. The two-program workload case also illustrates sampling overhead: sampling is initiated once every 10 scheduler quanta for only one tenth of the quantum, so we sample one percent of the time. Sampling incurs the drops and spikes in the ABC curves.

3.4 Hardware Overhead

As mentioned in the previous section, computing ABC in support of our reliability-aware scheduler requires hardware support. For an out-of-order core, we need counters for the five major structures, including the ROB, issue queue (IQ), load/store queue, register file (RF), and functional units (FU). Furthermore, we also need to factor out wrong-path and NOP instructions. We propose the following hardware additions. Per ROB entry, we keep two extra counters: one for recording the dispatch time of an instruction (i.e., the time it is inserted into the ROB), and one for recording the issue time (i.e., the time the instruction starts executing). These counters should be large enough to cover the maximum number of cycles an instruction resides in the ROB; we set the size of the counter to be 12 bits (maximum of 4096 cycles). At the time the instruction commits — which ensures that it is a correct-path instruction —

we can deduce the time this instruction spent in each of the relevant structures as follows:

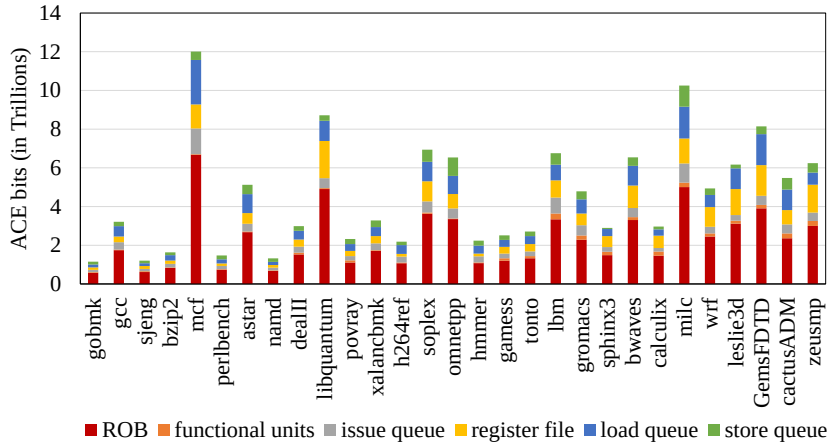

Figure 3.6: ABC stacks for the out-of-order core.

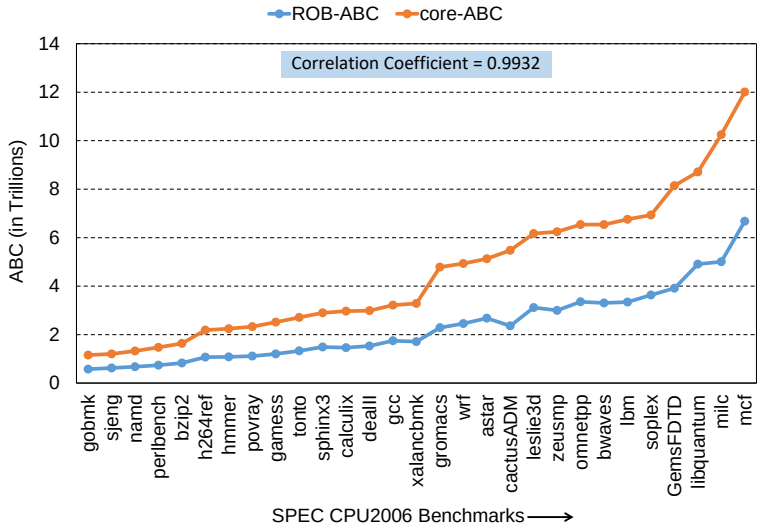

Figure 3.7: Correlation between core-ABC and ROB-ABC for the SPEC CPU2006 benchmarks on the big out-of-order core. Benchmarks are sorted in the increasing order of their core-ABC. *ROB-ABC is strongly correlated with core-ABC and using ROB-ABC for scheduling reduces hardware overhead from 906 bytes to 296 bytes.*

- The time spent in the ROB is the commit time minus the dispatch time.
- The time spent in the issue queue is the issue time minus the dispatch time.

- For a load or store instruction, the time spent in the load/store queue is the commit time minus the dispatch time — we model an architecture where load/store queue entries are allocated at dispatch time.
- The time the physical output register of an instruction is ACE is the commit time minus the finish time (which is the issue time plus its latency). Note that all architectural registers are ACE all of the time.
- The time spent in a functional unit is the functional unit's execution latency.

At the commit stage, where we keep one counter for each of the five structures, we add the per-instruction occupation time in each of the five structures to the respective overall counters. By doing so, the counters keep track of the accumulated occupancy in the respective structures. At the end of a quantum, total ABC is calculated as the accumulated occupancy times the number of bits per entry — the multiplication is done by the scheduler in software.

The total hardware overhead amounts to:

- Two 12-bit counters per ROB entry, which amounts to 3072 bits for an 128-entry ROB.
- One 32-bit counter per profiled structure, which amounts to 160 bits for 5 counters (with one counter per structure). 32 bits is sufficient for the quantum size in our setup (2.6 million cycles times at 2.6 GHz, and at most 128 entries per structure).
- Additional functional units for calculating occupancy and adding them to the counter. We need 5 adders per instruction in the data path (one per structure), and since up to 4 instructions can commit per cycle, this requires 20 adders in total.

Total hardware overhead thus equals 3,232 bits plus 20 adders. Extrapolation from [206] suggests that a 32-bit adder consumes about 1,200 transistors. One SRAM cell contains 6 transistors, so a rough equivalence relation is 200 SRAM bits for one 32-bit adder. So, in total the hardware overhead of this baseline implementation equals 7,232 bits or 904 bytes. To reduce the hardware overhead for the big core, the scheduler can use ACE bit information of the ROB only. We choose the ROB, because it is a central structure, containing a lot of useful state, and all other structures contain a subset of the instructions in the ROB. This is confirmed by the ACE bit counter (ABC) stacks shown in Figure 3.6 for the one-billion instruction workloads considered in this study. ABC stacks represent the breakdown of the total occupancy of a core in its microarchitecture structures. As Figure 3.7 shows, ROB ABC correlates very well with overall core ABC (correlation coefficient of 0.99), and contributes to almost half of the total occupancy of the core across all benchmarks. In other words, ROB ABC can serve as a proxy for the overall core ABC, which allows for correct scheduling decisions to be made using relative ABC numbers

across applications. For this implementation, we only need the dispatch time per ROB entry (12 bits times 128 entries equals 1,536 bits), one ROB ACE counter (32 bit) and 4 adders, resulting in a total of 2,368 bit equivalents or 296 bytes in total for this area-optimized implementation.

For the small in-order core, we only keep track of the fetch time. Because all instructions need to go through all stages, and each stage has a similar buffer for each instruction, we calculate the time between fetch and writeback of each instruction as a way to account for the number of ACE bits in the pipeline buffers. In addition, we add the functional unit ACE bits by multiplying the latency of the operation by the size of the functional unit. This requires 10 fetch time counters (5 stages times 2 instructions per stage) at 10 bits per counter (the time an instruction spends in the in-order core is usually less than in an out-of-order core), and one 32-bit total ACE counter (132 bits and two adders in total, resulting in 532 bit equivalents or 67 bytes).

3.5 Methodology

3.5.1 Experimental Setup

Because there is no way of evaluating architectural vulnerability on real hardware, we evaluate our scheduler using simulation. We use Sniper 6.0 [32] using its most detailed cycle-level core model. Sniper supports the x86 ISA (including x86-64 and SSE2) and runs both multiprogram workloads as well as multithreaded applications. This simulator has been validated against real hardware with a reported error that is similar to other academic simulators [32]. Note that the proposed reliability-aware scheduler is a generic solution that is applicable to any ISA. It is not geared towards a specific ISA, and can be deployed to any specific (micro)architecture.

We augment Sniper with ACE bit counters to count the number of ACE bits in the different structures. For the big out-of-order core, we count ACE bits in the ROB, issue queue, load/store queue, register file and functional units. Similarly, for the small in-order core, we count ACE bits in the fetch, decode, register read, execute and write-back stages. NOPs and wrong-path instructions are assumed un-ACE. Table 3.3 shows the configurations of the big out-of-order and the small in-order core types, as well as the bit counts per entry in each structure (taken from Nair et al. [145]). The big core includes an 8-stage *front-end pipeline*, i.e., we assume 8 stages between fetch and dispatch — the total number of pipeline stages is larger. We assume that the architectural state of a program is always protected. We consider a merged register file organization, similar to Intel Pentium 4, MIPS R12000 and Alpha 21264 processors [70, 86], where the same register file keeps speculative and (committed) architectural state of the program. We do not include the cache in the ACE calculation, because the cache configuration is the same for both core types, and caches typically include error detection and correction mechanisms, making them less vulnerable to soft errors. Our default configuration assumes

	Big core	*Small core*
Frequency	2.66 GHz	2.66 GHz
Type	out-of-order	in-order
ROB size	128, 76 bit/entry	-
Issue queue size	64, 32 bit/entry	4, 32 bit/entry
Load queue size	64, 80 bit/entry	-
Store queue size	64, 144 bit/entry	10, 144 bit/entry
Pipeline width	4	2
Pipeline depth	8 stages	5 stages
	(front-end only)	2 × 76 bit/stage
Functional units	3 int add (1 cyc)	2 int add (1 cyc)
	1 int mult (3 cyc)	1 int mult (3 cyc)
	1 int div (18 cyc)	1 int div (18 cyc)
	1 fp add (3 cyc)	1 fp add (3 cyc)
	1 fp mult (5 cyc)	1 fp mult (5 cyc)
	1 fp div (6 cyc)	1 fp div (6 cyc)
Register file	120 int (64 bit)	16 int (64 bit)
	96 fp (128 bit)	16 fp (128 bit)
L1 I-cache	32 KB, assoc 4, 2 cyc	32 KB, assoc 4, 2 cyc
L1 D-cache	32 KB, assoc 8, 4 cyc	32 KB, assoc 8, 4 cyc
Private L2 cache	256 KB, assoc 8, 8 cyc	256 KB, assoc 8, 8 cyc
Shared L3 cache	8 MB, assoc 16, lat 30 cyc	
Memory	BW 25.6 GB/s, lat 45 ns	

Table 3.3: Big and small core configurations.

the same frequency for both core types, but we also evaluate the impact of having a lower frequency for the small core than the big core.

3.5.2 Workloads

We create multiprogram workloads from the SPEC CPU2006 benchmarks. We construct 1 billion instruction SimPoints [180] for each benchmark. We categorize benchmarks into three groups, based on their sensitivity to reliability-aware scheduling, see also Figure 3.1. The eight benchmarks with the highest AVF are classified in the high sensitivity group (H); the eight benchmarks with the lowest AVF are classified as low sensitivity (L); and the 13 remaining benchmarks have medium sensitivity (M). For the two-program combinations, we make 6 categories of mixes: HH, HM, HL, MM, ML and LL. We randomly generate 6 workloads in each category — but we also make sure that each benchmark occurs at least once — leading to 36 evaluated workloads. For the four-program combinations, we take the same 6 mix categories by doubling the benchmark categories: HHHH, HHMM, HHLL, MMMM, MMLL and LLLL, and again generate 6 workloads in each category. We do not duplicate individual benchmarks, i.e., HHHH contains four different benchmarks. We do another doubling round for the eight-program combinations. The benchmarks making the four-program workloads are shown below:

1. `bwaves-milc-cactusADM-sphinx3`
2. `bzip2-perlbench-wrf-cactusADM`

3. gamess-calculix-cactusADM-sphinx3
4. GemsFDTD-gromacs-tonto-namd
5. GemsFDTD-gromacs-wrf-lbm
6. gobmk-zeusmp-sjeng-bzip2
7. gobmk-zeusmp-sjeng-tonto
8. h264ref-mcf-bwaves-milc
9. h264ref-mcf-gamess-calculix
10. hmmer-povray-cactusADM-sphinx3
11. hmmer-povray-gamess-calculix
12. lbm-h264ref-mcf-bwaves
13. leslie3d-bzip2-perlbench-wrf
14. leslie3d-dealII-bwaves-milc
15. leslie3d-dealII-cactusADM-sphinx3
16. libquantum-gcc-gamess-calculix
17. libquantum-gcc-hmmer-povray
18. libquantum-gcc-leslie3d-dealII
19. libquantum-gcc-wrf-lbm
20. libquantum-gcc-xalancbmk-soplex
21. milc-gobmk-zeusmp-sjeng
22. omnetpp-astar-cactusADM-sphinx3
23. omnetpp-astar-GemsFDTD-gromacs
24. omnetpp-astar-leslie3d-dealII
25. omnetpp-astar-libquantum-gcc
26. omnetpp-astar-tonto-namd
27. omnetpp-astar-xalancbmk-soplex
28. omnetpp-gobmk-zeusmp-sjeng
29. perlbench-GemsFDTD-gromacs-wrf
30. tonto-namd-cactusADM-sphinx3
31. tonto-namd-gamess-calculix
32. tonto-namd-hmmer-povray
33. wrf-lbm-hmmer-povray
34. xalancbmk-soplex-gamess-calculix
35. xalancbmk-soplex-h264ref-mcf
36. xalancbmk-soplex-leslie3d-dealII

We evaluate the two-program workloads on an HCMP consisting of 1 big and 1 small core (denoted 1B1S). The four-program workloads are evaluated on both symmetric and asymmetric HCMPs. In a *symmetric* HCMP, the number of big cores equals the number of small cores, while in an *asymmetric* HCMP, the number of big cores differs from the number of small cores. We evaluate our four-program workloads on a symmetric HCMP configuration consisting of 2 big and 2 small cores (2B2S), and also on asymmetric HCMP configurations: 1 big, 3 small cores (1B3S) and 3 big, 1 small cores (3B1S). The eight-program combinations are evaluated on a symmetric HCMP with 4 big and 4 small cores (4B4S). The standard quantum time is 1 ms. For each experiment, the longest running application executes its full 1 billion instruction SimPoint, and the faster running applications are restarted until the end of the experiment. For the applications that restart, we record performance and wSER across all rep-

etitions of that application. The reason is that the longer running application could enter a new phase near the end of its execution, causing the schedule to change, which in its turn impacts the other co-running application(s). Taking results from the first execution only for the repeating application(s) would not cover these changes in the schedule.

3.5.3 Migration Overhead

The overhead for saving and restoring microarchitectural state to support core migration plus the overhead of weighted speedup/SER calculation is (conservatively) modeled as 20 μs. The impact of cache warming (including cache-to-cache transfer latency) is modeled faithfully in the simulator. This overhead has a negligible impact on the final results: less than 1% for a random scheduler that switches every quantum, and less than 0.5% for both the performance- and reliability-optimized schedulers.

3.6 Evaluation

We evaluate the following three schedulers:

- The *random scheduler*, for each time slice, randomly selects the applications to run on the big core(s).
- The *reliability-optimized scheduler* optimizes SSER using the algorithm from Section 3.3.
- The *performance-optimized scheduler* optimizes system throughput (STP) [60] or weighted speedup, using the same sampling-based scheduling algorithm optimizing for STP rather than SSER.

We first analyze the results for the 2B2S configuration. Next, we show how our scheduler performs for different core and application counts, as well as for asymmetric HCMP configurations. We also show the impact of using only ROB ACE bits to steer scheduling, the impact of the sampling period, and the impact of reducing the frequency of the small core.

3.6.1 2B2S Results

Figure 3.8 evaluates system soft error rate (SSER) and system throughput (STP) for the reliability- and performance-optimized schedulers, normalized to the random scheduler, for four-program workloads running on a 2B2S HCMP. SSER is a *lower-is-better* metric, while STP is a *higher-is-better* metric. Each dot represents a workload; the workloads are sorted by SSER and STP, respectively.

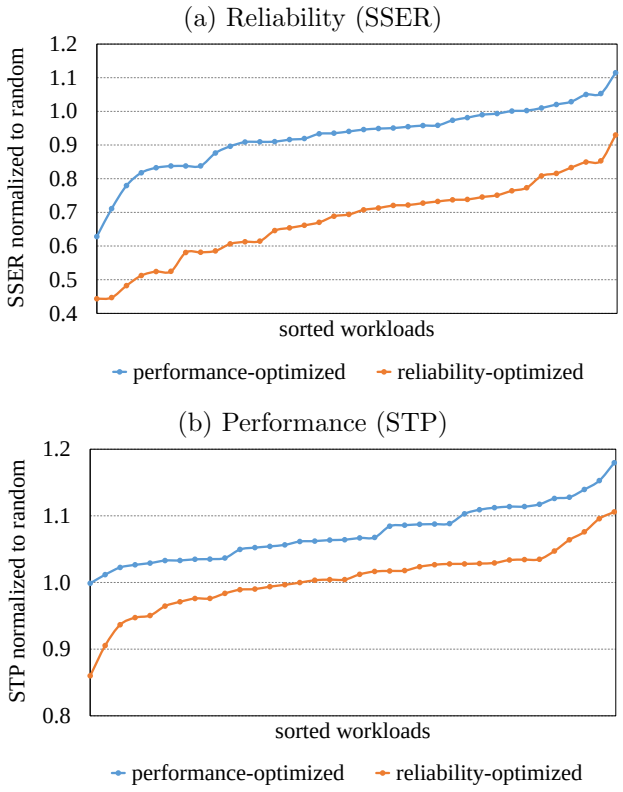

Figure 3.8: System soft error rate (a) and system throughput (b) for reliability- and performance-optimized scheduling normalized to random scheduling for all four-program workloads on an HCMP with 2 big cores and 2 small cores. *The reliability-optimized scheduler significantly and consistently improves reliability; overall, reliability improves by 25.4% compared to the performance-optimized scheduler.*

The reliability-optimized scheduler significantly and consistently improves reliability, i.e., SSER reduces by 32% on average and up to 55.6% compared to the random scheduler; and by 25.4% on average and by up to 60.2% compared to the performance-optimized scheduler. Reliability-aware scheduling effectively determines which applications are most vulnerable to soft errors and puts those applications on the small cores to improve overall system reliability.

The performance-optimized scheduler also reduces SSER over the random scheduler (by 7.3% on average). This improvement is substantially smaller and, moreover, it is not consistent, i.e., reliability decreases for a number of workloads. The reason for the (average) improved reliability is the apparent correlation between performance and reliability. For example, `sjeng` and `povray` are compute-intensive benchmarks. The performance-optimized scheduler will run both `sjeng` and `povray` on the big cores. However, as Figure 3.1 shows, these benchmarks do not expose a large vulnerable state on the big core relative to

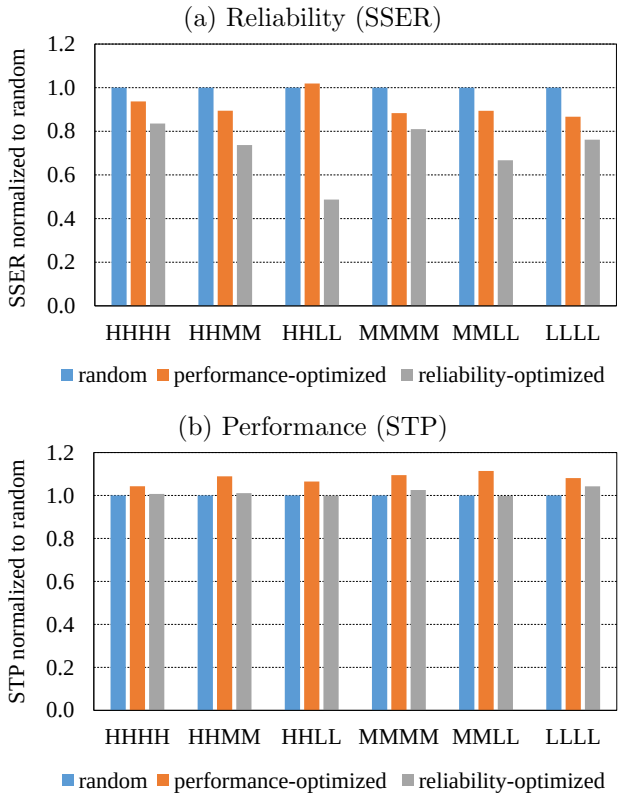

Figure 3.9: SSER (a) and STP (b) on a 2B2S system per workload category. *Workload categories with more divergent AVF among benchmarks experience the highest improvement in system reliability.*

the small core. Scheduling these benchmarks on the big cores improves both performance and reliability.

In terms of performance, the reliability-optimized scheduler yields similar performance to the random scheduler (half of the workloads are worse, half are better, resulting in an average near 0% difference), and degrades performance by only 6.3% on average (and by 18.7% at most) compared to the performance-optimized scheduler. The performance improvement of performance-optimized scheduling over random scheduling is in line with prior work [207].

3.6.2 Analysis by Workload Category

Figure 3.9 shows the same results as Figure 3.8 but now groups the results per workload category, with the categories defined based on big-core AVF, see Section 3.1.2. The largest improvement in system reliability is observed for the workload category that includes high-AVF applications and low-AVF applications (see 'HHLL'). This does not come as a surprise: the high-AVF applications are scheduled on the small cores to reduce overall system reliability,

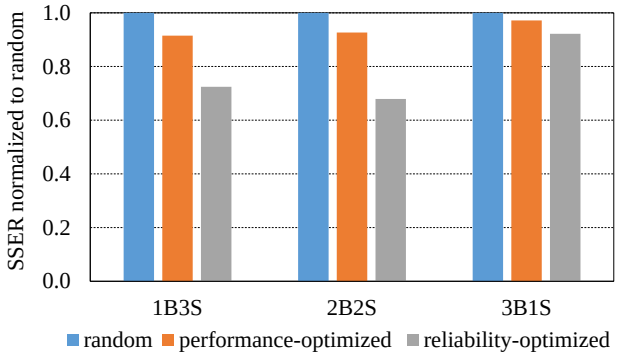

Figure 3.10: SSER across asymmetric HCMPs with 4 cores in total. *Higher improvements in system reliability are obtained for symmetric HCMPs than asymmetric HCMPs.*

while scheduling the low-AVF applications on the big cores. The workload categories with less divergent application behavior ('HHMM' and 'MMLL') also show substantial improvements in reliability, though not as high as for the 'HHLL' category. Here, again, reliability-aware scheduling is able to schedule the applications with high AVF (relative to the other applications in the mix) on the small cores and vice versa. For the workload categories with similarly AVF-sensitive applications (all 'H', 'M' or 'L' applications), we observe modest improvement in reliability. The reliability-aware scheduler makes the correct scheduling decisions in terms of AVF, i.e., it schedules applications with the highest AVF on the small cores and the applications with the lowest AVF on the big cores. Nevertheless, this leads to a small improvement in system reliability because of the lower system performance compared to performance-optimized scheduling, which tempers the improvement in soft error rate — remember that SSER weights relative per-application slowdown.

3.6.3 Asymmetric HCMPs

The results in the previous sections assume symmetric HCMPs, i.e., the number of big cores equals the number of small cores. We now evaluate asymmetric HCMP configurations for four-program workloads: 1 big and 3 small cores (1B3S), and 3 big and 1 small cores (3B1S), see Figure 3.10. The most noteworthy observation from this graph is that the highest reduction in SSER is obtained for the symmetric HCMP configuration; this is due to the fact that there are more scheduling opportunities on the symmetric HCMP than on the asymmetric HCMPs, i.e., 2 out of 4 applications need to be selected to run on a small core on the symmetric HCMP (2 combinations out of 4 leads to 6 possibilities) as opposed to one application to run on the big or small core in the asymmetric HCMPs (1 combination out of 4 leads to only 4 possibilities). The reduction in SSER on the 3B1S system (7.8%) is smaller than on the 1B3S system (27.5%) because there is only one small core available in the

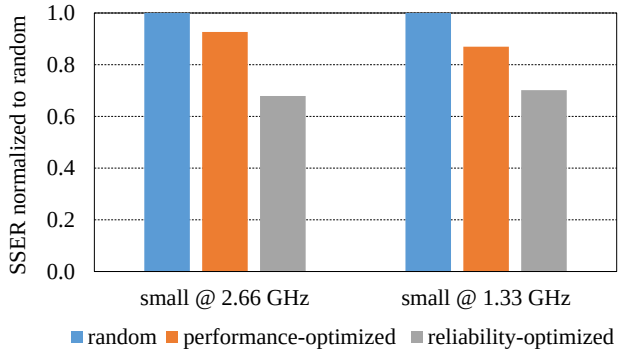

Figure 3.11: SSER for the 2B2S system with the small cores running at different frequency settings. *Reliability-aware scheduling is robust with respect to small-core frequency setting.*

former system; this limits the scheduling opportunities to reduce soft error rate by migrating one of the co-running applications to the small core.

3.6.4 Lowering Small Core Frequency

So far, we assumed that the big and small cores run at the same frequency. We now evaluate the robustness of the reliability-aware scheduler with respect to frequency setting, see Figure 3.11. To this end, we set small-core frequency to 1.33 GHz while running the big core at 2.66 GHz. The bottom line is that reliability-aware scheduling is robust with respect to frequency setting: system reliability improves by 29.8% compared to random scheduling for the low-frequency small core. This improvement is slightly smaller compared to the high-frequency small core case because lowering the small core's frequency also lowers its performance, which increases its weighted SER because of the larger slowdown compared to big-core performance. This reduces the opportunity for reliability-aware scheduling to improve reliability compared to the high-frequency small core case. The performance-optimized scheduler on the other hand improves reliability more for the low-frequency small core than for the high-frequency small core (13% versus 7.3%) compared to random scheduling. This is a side-effect of the wider gap between big and small core performance. Performance-optimized scheduling improves overall system performance compared to random scheduling (by 10% on average), which decreases an application's weighted SER and improves overall system reliability more than random scheduling.

3.6.5 Changing Core Count

Figure 3.12 evaluates SSER across two-, four- and eight-program combinations on symmetric HCMPs (1B1S, 2B2S, and 4B4S). The results are consistent across core counts: the reliability-optimized scheduler significantly im-

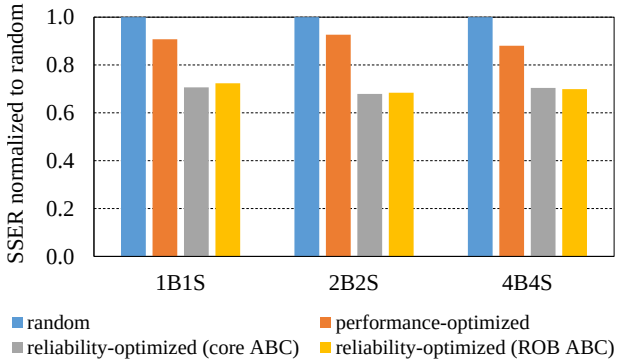

Figure 3.12: SSER as a function of core count, assuming symmetric HCMPs and considering ROB ABC in addition to core ABC.

proves system soft error rate compared to random scheduling by 29.3%, 32% and 29.8% on average for 1B1S, 2B2S and 4B4S, respectively, while yielding comparable performance to the random scheduler, and only slightly worse performance compared to the performance-optimized scheduler (within 6.3% on average). This result shows that our scheduler scales well with core count and the number of co-running applications.

3.6.6 ROB ACE Bit Counter

Up to now, we assumed an ACE bit counter for all structures. To reduce hardware overhead by a factor of 3, as previously described in Section 3.4, the area-optimized implementation counts ACE bit information in the ROB only. Figure 3.12 shows SSER for reliability-aware scheduling using core ABC versus ROB ABC. The relative difference is negligible (31.6% reduction in SSER for ROB ABC versus 32% for core ABC for the 2B2S system), which justifies the reduction in hardware cost by only tracking ROB ABC.

3.6.7 Sample Rate

Figure 3.13 evaluates the impact on system reliability and performance while varying the sample rate to keep the big and small core performance and soft error rates up to date, see also Section 3.3.1. Our default sampling period SP is set to 10, i.e., we initiate a sampling phase every 10 scheduler quanta, and we sample for a sampling quantum of 0.1 milliseconds. Two interesting observations are to be made here. First, reliability improves for smaller sampling quanta as a result of reduced sampling overhead. Second, reliability improves as we increase the sampling period, i.e., as we sample less frequently. This suggests that our workloads show relatively stable time-varying execution behavior. However, some workloads clearly benefit from having a high sample frequency. For example, the workload consisting of xalancbmk, soplex, leslie3d

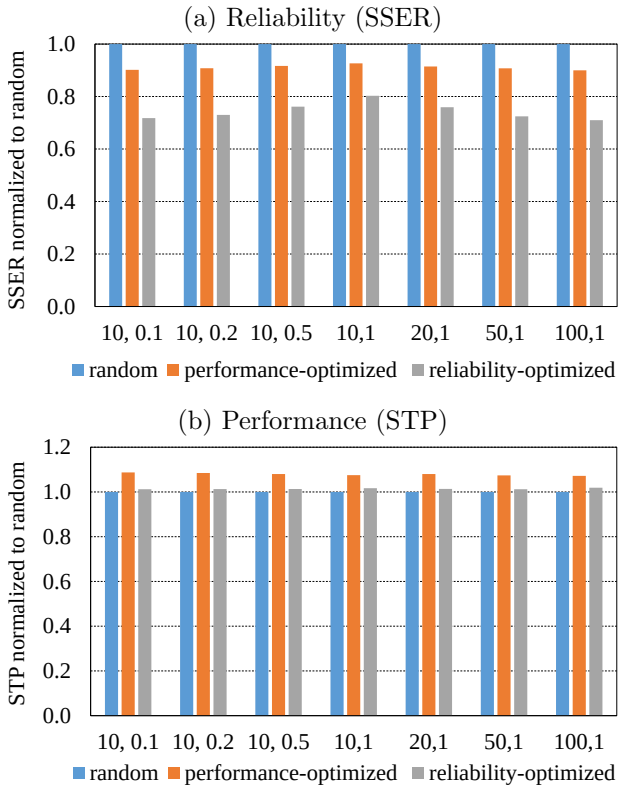

Figure 3.13: SSER (a) and STP (b) for a 2B2S system while varying the sampling parameters (r, s), i.e., sampling every r quanta for s milliseconds (i.e., the sampling quantum).

and dealII has a 18.4% reduction in SSER for a sampling period of 10, whereas SSER reduces by only 10% for a sample period of 100.

3.7 Power Analysis

There is an important trade-off between performance, power, and reliability, as corroborated by a recent study [199]. In the previous section, we focused on performance and reliability. However, changing the workload schedule on a heterogeneous multicore also affects power consumption. Therefore, in this section, we first evaluate how reliability-aware scheduling affects power consumption. We then subsequently explore the trade-off between scheduling for performance, power and reliability.

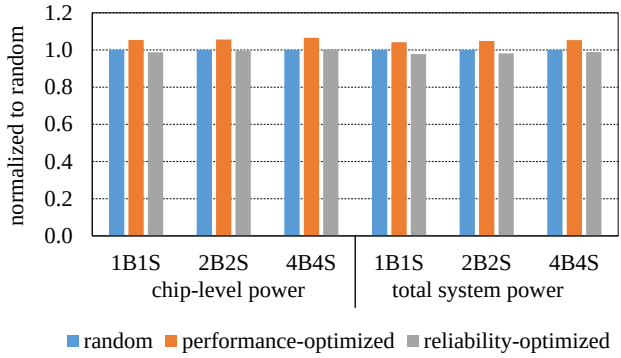

Figure 3.14: Impact on chip-level and total system power consumption.

3.7.1 Impact of Reliability-Aware Scheduling on Power

Figure 3.14 quantifies the impact on chip-level power (including L3) and total system power (processor plus DRAM) with increasing core count. We use McPAT [118] to quantify power consumption. The bottom line is that reliability-optimized scheduling reduces chip-level and system power by 6% and 6.2% on average, respectively, relative to performance-optimized scheduling. The reason is that performance-optimized scheduling puts applications on a big core for performance reasons although this may increase power consumption. For example, a memory-intensive application with high degrees of MLP will be scheduled on the big core to improve performance [207]; this will lead to an increase in power consumption. The reliability-aware scheduler on the other hand schedules this workload on the small core to reduce soft error vulnerability, also reducing power.

3.7.2 Trade-Offs in Performance-, Power- and Reliability-Optimized Scheduling

We implemented a *power-optimized scheduler* to evaluate the impact of optimizing for power on reliability. The power-optimized scheduler, alike our reliability- and performance-optimized schedulers, is a sampling-based scheduler. The scheduling decision is based on the energy consumed by each core per quantum. Note that an ideal power- or reliability-optimized schedule can be achieved by scheduling workloads on small cores in a sequential manner. However, all our scheduling policies maintain the fundamental assumption that no core remains idle while an HCMP is executing a multiprogram workload. Figure 3.15 shows the relationship among power, performance and reliability when we execute four-program workloads on 2 big and 2 small cores. Optimizing power always leads to performance degradation, and also leads to an overall improvement in reliability (by 12.2% on average) compared to random scheduling. Benchmarks such as `cactusADM` and `hmmer` expose a large state inside the big core, leading to high soft error vulnerability and high power consumption. Such benchmarks are scheduled on a small core to improve both power and

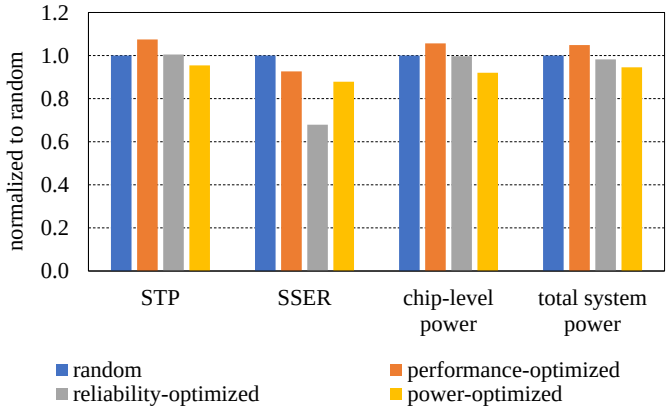

Figure 3.15: Comparing performance-, reliability- and power-optimized schedulers for all four-program workloads on an HCMP with 2 big cores and 2 small cores. All results are normalized to the random scheduler.

reliability. For such benchmarks, optimizing for power also leads to an improvement in reliability, and vice versa, optimizing reliability also improves power.

There are several workloads for which a reliability-optimized schedule is different from a power-optimized schedule. For example, `milc` and `sjeng` run on different core types for reliability and power when they co-run. Compared to `milc`, `sjeng` incurs much higher power on the big core compared to the small core. Therefore, for power, it is always scheduled on the small core. On the other hand, `milc` is a memory-intensive benchmark that maintains a large vulnerable state inside the core while waiting for long-latency memory requests to complete. Since the difference in SER for `sjeng` is small between the big and small cores, the reliability-optimized scheduler runs `milc` on the small core and `sjeng` on the big core.

The key take-away from the results reported in Figure 3.15 is that there is a trade-off between performance-, power- and reliability-optimized scheduling. Performance-optimized scheduling leads to high performance, but also leads to high power consumption and soft error vulnerability. Power-optimized scheduling minimizes power consumption, however this comes at a cost in performance. Reliability to soft errors slightly improves under power-optimized scheduling compared to performance-optimized scheduling. Reliability-optimized scheduling improves reliability by a significant margin while being on par with random scheduling in terms of performance and power consumption.

3.8 Reliability-Aware Scheduling under Performance Constraints

So far, we assumed that the goal is to optimize reliability while considering performance after the fact, i.e., we schedule applications to core types to optimize for reliability and we pay the cost this may incur in terms of performance. In many systems however, performance is more important than reliability, and one may not be willing to pay an average 6.3% performance degradation compared to performance-optimized scheduling, even if this improves reliability by 25.4% on average, as previously reported. Although reliability is an important concern, one may not want to incur a performance hit by more than a predefined limit, say 2%, but within this constraint one may yet want to improve reliability. In this section, we explore reliability-aware scheduling under performance constraints.

With a minimum acceptable performance level specified, we propose to augment the scheduler with a mechanism to dynamically switch between the reliability- and performance-optimized modes at runtime. The decision to choose either of the two modes depends on the requirements of the system and the workload under execution. If reliability is of utmost importance — for example, in systems working at higher altitudes in space — the goal should be to optimize for reliability. In such cases, running in the reliability-optimized mode suits the best. However, when performance is the key concern and performance is not allowed to drop below a certain performance level relative to performance-optimized scheduling, the scheduler should switch to the performance-optimized mode once the performance is about to drop below the specified level.

3.8.1 Scheduling Mechanism

To achieve performance above a specified level, we need to keep track of performance while improving reliability. At the end of every scheduling quantum, we estimate performance (i.e., STP) for all possible schedules and discard the schedules not meeting our performance criterion. Of the remaining schedules, we choose the schedule with the lowest SSER as the schedule for the next quantum. If an application continues to run on a particular core type for 10 consecutive scheduling quanta (1 ms each), a sampling phase (0.1 ms) is triggered to account for the possibility of a phase change in the application's execution behavior.

3.8.2 Evaluation

To evaluate reliability-aware scheduling under performance constraints, we consider reliability (SSER) and performance (STP) for four-program workloads on a 2B2S system under various performance constraints, see Figure 3.16. We start with the performance-optimized scheduler (shown on the left) and gradu-

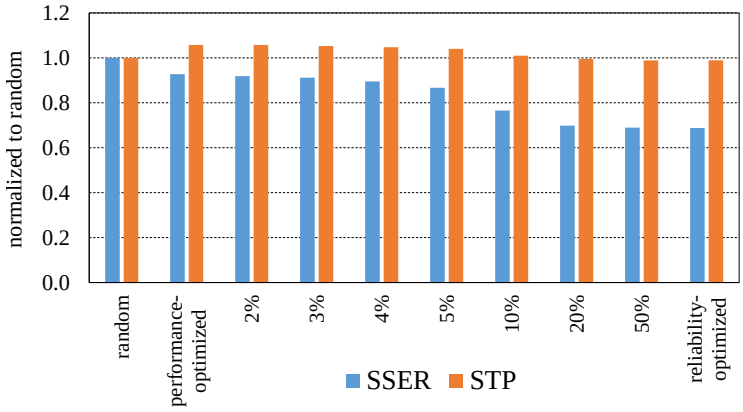

Figure 3.16: Average reliability (SSER) and performance (STP) relative to the random scheduler for reliability-aware scheduling under performance constraints, for four-program workloads on a 2B2S system.

ally increase the allowable performance degradation. Eventually, when there is no performance constraint, we end up with the reliability-optimized scheduler (shown on the right). When the performance limit is set at x%, the STP of the four-program workload must never be degraded by more than x% at any point during the execution compared to the performance-optimized schedule. For example, when the performance limit is set at 4%, and the highest possible STP of a four-program workload in a scheduler quantum is 3.0 on a 2B2S system, then the scheduler should map applications in such a manner that the STP does not degrade below 2.88 (4% degradation of 3.0). This constraint is met every quantum and ensures that the workload will not experience an overall performance degradation by more than 4%; in fact, the average performance degradation is typically smaller than the performance limit that was set.

It is important to note that a performance constraint can also be imposed for a longer interval than a per-quantum basis. However, in this section, we aim to improve reliability without delaying the response of the system beyond a specified threshold at *all* times. One such scenario occurs for interactive applications. If the performance of an application degrades beyond the human perception threshold, the user may notice a difference in the response time. However, as long as the performance is maintained within this threshold, the OS can run in the reliability-optimized mode to minimize the number of soft errors.

The results in Figure 3.16 indicate a clear trend — increasing the allowable performance limit increases the performance degradation while at the same time improving reliability, as expected. At small performance constraints, the improvement in reliability is limited and so is the impact on performance. The reason for the small impact is the limited number of opportunities for choosing an alternative schedule. In particular, there are six possible mappings for a four-program workload on two big and two small cores: `BBSS`, `SSBB`, `BSBS`, `SBSB`, `BSSB` and `SBBS`; where a `B` and `S` represents the respective application

running on the big versus small core, respectively. One of these schedules is the performance-optimized schedule. The scheduler has limited opportunity to choose a schedule other than the one that optimizes performance while remaining within 2% of the performance-optimal schedule. However, it may still successfully pick such a schedule in very few cases.

Increasing the performance constraint provides more flexibility to the scheduler and the improvement in reliability is also higher. In particular, the average gain in SSER for a performance constraint of 5% and 10% equals 13.5% and 23.5%, respectively. Note that performance is still better than the random scheduler for these performance levels. As the limit is further increased, the scheduler starts to choose schedules that are more similar to the ones chosen by the reliability-optimized scheduler. For the 20% and 50% performance limits, the numbers are very close to the reliability-optimized scheduler — an average improvement in reliability of 32% at the cost of a 1% performance degradation compared to the random scheduler. Overall, we conclude that the improvement in reliability is always much higher than the degradation in performance. The higher the allowable performance degradation, the higher is the improvement in reliability; the actual performance constraint, however, can be adjusted by the system administrator or end user based on the requirements.

3.9 Multi-Threaded Workloads

So far, we considered multiprogram workloads composed out of single-threaded programs, for which we observed the highest improvements in reliability for workload mixes consisting of diverse applications, i.e., high-AVF applications running concurrently with low-AVF applications; the smallest improvements are observed for workload mixes composed out of applications with similar AVF characteristics. We now consider multi-threaded workloads. Most multi-threaded workloads are data-parallel in which all threads execute the same code on different portions of the data. As a result, all threads exhibit similar execution behavior. We refer to these workloads as *homogeneous* workloads. Some multi-threaded workloads however expose pipelined parallelism, i.e., the outcome produced by one thread is the input for another thread. These workloads are *heterogeneous*, i.e., different threads execute different code. Based on the results obtained for the multi-program workloads, we expect limited improvement for the multithreaded workloads that are homogeneous, but we expect a higher improvement for the heterogeneous workloads.

3.9.1 Metrics

For multiprogram workloads, the necessity for metrics such as STP for performance and SSER for reliability arises from the fact that co-executing programs affect each other's performance. However, for multithreaded workloads, execution time or *start-to-finish* time correctly measures performance, i.e., this is the time it takes to get a unit of work done. Similarly, since the

amount of work performed by a multi-threaded program is fixed, SER is an appropriate metric to quantify the vulnerability of a multi-threaded program to soft errors. ABC of a multithreaded program is the sum of the ABC values for all threads. Once we know the overall ABC, SER can be calculated as described in Section 2.3.

3.9.2 Performance-Optimized Scheduling

Identifying bottlenecks and improving performance of multithreaded workloads on multicore hardware is a challenging task, especially on heterogeneous multicore processors, and a number of prior works have focused on this problem, see for example [20, 55, 94]. The challenge when executing multi-threaded workloads on multicore hardware is to make sure that all threads make equal progress, i.e., all threads need to reach the end of the execution or the next barrier at roughly the same time, or in other words, the execution needs to be balanced. This may be complicated because of negative interference in shared resources, e.g., one thread may evict another thread's data from the shared cache. Heterogeneous multicore processors further complicate this, i.e., the thread(s) running on the big core(s) make much faster progress than the one(s) running on the small core(s). One solution is to make sure all threads get an equal share of the big core cycles, i.e., by allowing all threads to run on a big core alternately. This leads to a balanced execution, improving overall application performance. This is typically a viable solution for homogeneous multi-threaded workloads, however, heterogeneous workloads need a more involved solution, i.e., we need to make sure all threads make equal progress, as described by Van Craeynest et al. [208]. There is a subtle but important difference between *equal share* and *equal progress*. Equal share guarantees the same number of big core cycles for all threads; equal progress on the other hand guarantees that all threads benefit equally from running on the big cores, e.g., if one thread benefits twice as much from running on the big core, it will receive only half as many cycles. Equal progress enables balanced execution even for heterogeneous multi-threaded workloads. Van Craeynest et al. [208] find that equal-progress scheduling is the best performing performance-optimized scheduler, which we adopt accordingly in this section.

3.9.3 Results

For our evaluation, we compare the reliability- and performance-optimized schedulers on an HCMP with two big and two small cores (2B2S). The two schedulers minimize SER and total execution time, respectively. We also compare against a random scheduler that randomly selects threads to run on the big cores.

Table 3.4: Multithreaded benchmarks from PARSEC and Rodinia.

Suite	Benchmark	Input size
Rodinia	backprop	large
	bfs	large
	cfd	large
	hotspot	large
	kmeans	large
	srad	large
PARSEC	bodytrack	large
	canneal	large
	dedup	medium
	ferret	medium
	fluidanimate	large
	swaptions	large

3.9.3.1 Methodology

We need to consider a few subtle changes in the experimental methodology for the multi-threaded workloads in comparison to the multiprogram workloads. The scheduling quantum can be fixed for the multiprogram workloads (e.g., 1 ms). However, this is not appropriate for multi-threaded programs for which the number of running threads may vary dynamically at runtime because of sequential code sections and synchronization activity. Therefore, a scheduler should only take into account the threads performing useful work. When the number of active threads does not change during the course of a 1 ms time interval, we fix the scheduling quantum to 1 ms. In addition, a quantum starts (or ends) when a thread changes from running to waiting and vice versa. In such cases, the size of a quantum will be less than 1 ms. This flexibility in quantum size is required to consider all running threads for scheduling. Sampling is performed in a manner similar to what is done for the multiprogram workloads — when a thread continues to run for ten consecutive quanta on one core type, we trigger the sampling phase for a period of 0.1 ms.

Another difference is that the number of active threads at runtime can be less than the number of cores available in an HCMP. This may lead to certain cores remaining idle for some time during program execution. When there is a possibility of a core remaining idle during a scheduler quantum, we utilize as many big cores as possible to take advantage of their high performance. For example, in a 2B2S system, if there are only three active threads during a quantum, the two big cores will always be running two threads and one small core will run the third thread leading to one small core remaining idle.

We use benchmarks from the Rodinia [38] and PARSEC [21] suites to evaluate our reliability-aware scheduler for multithreaded workloads, see Table 3.4. We simulate the benchmarks that we were able to successfully run on our simulator. We consider the parallel portion of the benchmarks in the evaluation; the sequential phases are run on the big core for highest performance. All bench-

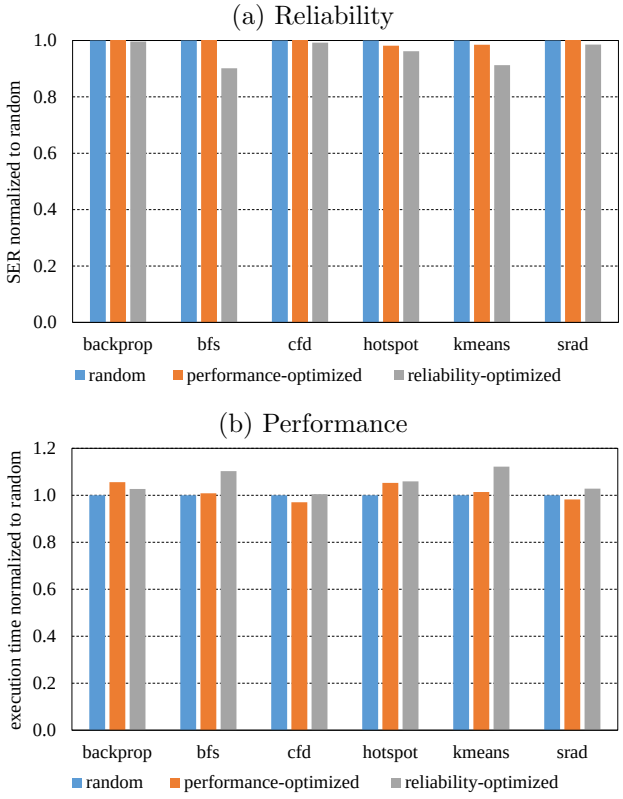

Figure 3.17: Reliability (a) and performance (b) for the Rodinia benchmarks on a 2B2S system.

marks except for `ferret` were executed on a 2B2S system. `ferret` requires at least six threads (cores) for execution and therefore we simulate six threads for `ferret` on an HCMP with three big and three small cores (3B3S).

3.9.3.2 Rodinia

Figure 3.17 shows reliability and performance for the Rodinia benchmarks for reliability-aware scheduling compared to random and performance-optimized scheduling. The highest improvement in soft error rate compared to both random and performance-optimized scheduling is achieved for `bfs` (10%), followed by `kmeans` (8.8%). The improvement is less significant for the other benchmarks. Looking at performance, we observe that reliability-aware scheduling is either performance neutral or degrades performance. We note a one-to-one trade-off between reliability and performance for most benchmarks. For example, for `bfs`, reliability-aware scheduling improves reliability by 10% while at the same time degrading performance by 10%. The reason is that the Rodinia benchmarks are homogeneous data-parallel workloads, and hence there is limited opportunity to improve reliability, as expected and argued

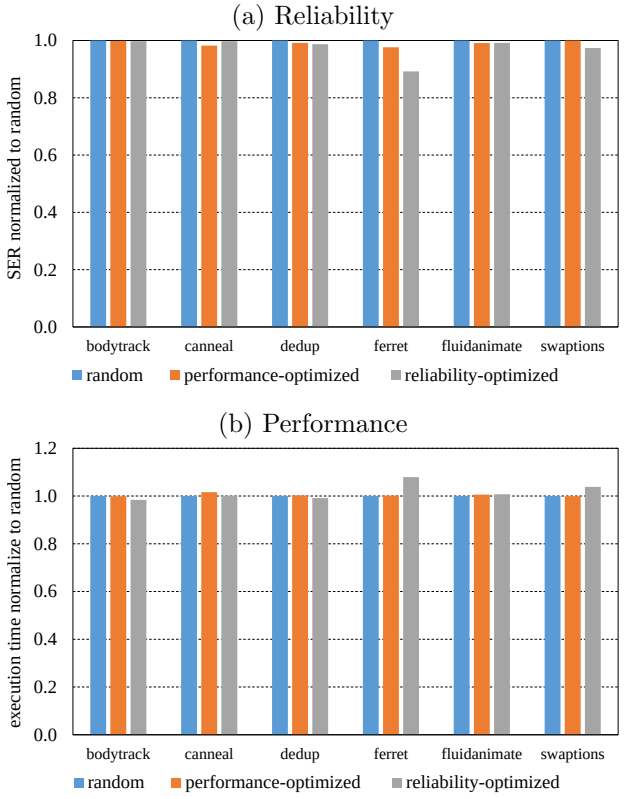

Figure 3.18: Reliability (a) and performance (b) for the PARSEC benchmarks.

above. The data-parallel nature of the workloads is also the reason for similar performance between the random and performance-optimized scheduling.

3.9.3.3 PARSEC

Figure 3.18 shows the results for the PARSEC benchmarks. Two of the PARSEC benchmarks are heterogeneous workloads, namely `ferret` and `dedup`; all other benchmarks are homogeneous data-parallel workloads. We observe similar results for the homogeneous PARSEC benchmarks as for the Rodinia benchmarks: the improvement in reliability through reliability-aware scheduling leads to an almost equally high degradation in performance (and both are small). For one of the heterogeneous workloads, namely `ferret`, we do observe an interesting result: the improvement in reliability (11%) is higher than the degradation in performance (7%), which is in line with the results and conclusion obtained for the multiprogram workloads. Unfortunately, we do not observe a similar result for `dedup`, the other heterogeneous benchmark. Through detailed analysis using *bottle graphs* [56] of `dedup`'s execution behavior, we observe that there is a very high degree of parallel imbalance among the

threads. One critical thread runs for a longer time than all other threads put together. This leads to the other non-critical threads remaining idle for most of the execution. Since our scheduling policy never leaves a big core idle, the critical thread is always running on the big core for all three schedulers, thus leading to similar reliability and performance figures for all of them.

The overarching conclusion from this section is that reliability-aware scheduling has limited benefit for multi-threaded workloads. The primary reason is that different threads typically execute the same code and hence there is limited opportunity to exploit diversity in AVF characteristics across the different threads. We typically observe a one-to-one trade-off between reliability and performance. Only in a limited number of cases, i.e., heterogeneous workloads with different threads that execute different code and that exhibit different AVF characteristics, do we observe an opportunity to improve reliability at the expense of a relatively small performance degradation.

3.10 Incorporating Unprotected L1 Caches

Protection techniques based on Error Detecting Codes (EDC) and Error Correcting Codes (ECC) incur chip area, power and possibly latency overheads, and are typically applied to the cache levels beyond the L1 caches [189]. Several prior works estimate and mitigate soft errors of on-chip caches in general, and L1 caches in particular, see for example [12, 22, 24, 135, 189, 213]. Recent work also focuses on dynamically reconfiguring last-level caches and improving reliability across the cache hierarchy in the presence of multibit soft errors [110, 111, 197]. Reliability-aware scheduling as proposed in this chapter works for the case in which the L1 caches are protected (which is what we assumed so far) as well as for the case in which the L1 caches are not protected (which is the subject of this section). In order for reliability-aware scheduling to be able to incorporate L1 cache soft error vulnerability, we need to also estimate and measure L1 cache soft error vulnerability. In this section, we first explain our methodology to dynamically compute the ABC for the L1 caches and then evaluate how well reliability-aware scheduling performs taking into account reliability of the core *and* the L1 caches.

3.10.1 Estimating Cache Soft Error Vulnerability

Our methodology for calculating ACE Bit Count in the data and tag arrays is based on the work done by Biswas et al. [22]. A cacheline is ACE if its correctness is required for the correct execution of a program. (Note we assume both the L1 D-cache and L1 I-cache to be write-back caches.) For the data array, ABC can be estimated as follows. There are four time intervals during which a cacheline is ACE: *fill-to-read*, *read-to-read*, *write-to-read* and *write-to-evict*. For the tag array, the correctness of a program is affected only by the *false-positive* case, when an incorrect cacheline is returned due to an error in the tag bits. Therefore, to estimate ABC in this case, we implement the *hamming-*

distance-one analysis and conservatively assume that all (tag) entries of a set are at a hamming-distance of 1 from the tag bits of the requested memory address. According to hamming-distance-one analysis, wrong data will be returned to the core when there is a difference of only a single bit between the incoming address and the tag bits of a cacheline, and this particular tag bit flips due to soft error, causing a cache hit. Therefore, such a tag array bit must be ACE for each cacheline in a set; the maximum number of such bits for a set is equal to the associativity of the cache. For example, each tag array access contributes 8 ACE bits for the 8-way set-associative L1 D-cache with one cycle tag access time in our setup.

3.10.2 Hardware Overhead

The hardware cost for computing ABC for the L1 caches is limited. There are 512 64 B cachelines in a 32 KB L1 cache. Assuming a quantum size of 1 ms, this amounts to 375,940 cycles when running at 2.66 GHz. This is also the maximum number of cycles a cacheline can be ACE. Accounting for this many cycles requires 18.5 bits; we assume 20 bits per cacheline. For each cacheline, we keep track of the last access time. This amounts to 20 bits per cacheline ABC counter, or a total of 1280 bytes.

Whenever a cacheline is read/written/evicted, we update one global cache-wide ABC counter. In the 'worst' case, the entire cache can be ACE for one quantum, which implies that this cache-wide ABC counter requires 36 bits. When a cacheline is read or evicted, we add the difference between the current cycle count (since the beginning of the scheduling quantum) and the cacheline ABC counter to the global counter, and we replace the cache ABC counter value with the current cycle count upon a read, eviction and write. A 36-bit adder is equivalent to 250 bits, similar to what is described in Section 3.4. The addition of ABC counters across quanta can be done in software. The overall hardware cost for an L1 cache amounts to 1314 bytes.

3.10.3 Impact of Caches on Soft Error Vulnerability

The impact L1 caches have on soft error vulnerability is quantified in Figure 3.19: cache-AVF and total-AVF (that is, AVF for the core and the L1 caches; total-ABC is defined similarly) are shown for the SPEC CPU2006 benchmarks; the benchmarks are sorted in the same order as in Figure 3.1. (Note that the reported AVF values are much smaller in Figure 3.19 compared to Figure 3.1; this is because the L1 caches are now included in the total structure size.) It is clear from the figure that there is a strong correlation between total-AVF and cache-AVF. This is primarily because total-ABC is dominated by the L1 caches. The size of, or more precisely the architecture state contained in, the L1 caches is ten times higher than the out-of-order core — 64 KB versus almost 6 KB. In spite of the strong correlation between cache-AVF and total-AVF, we observe that the gap between both curves widens going from left to

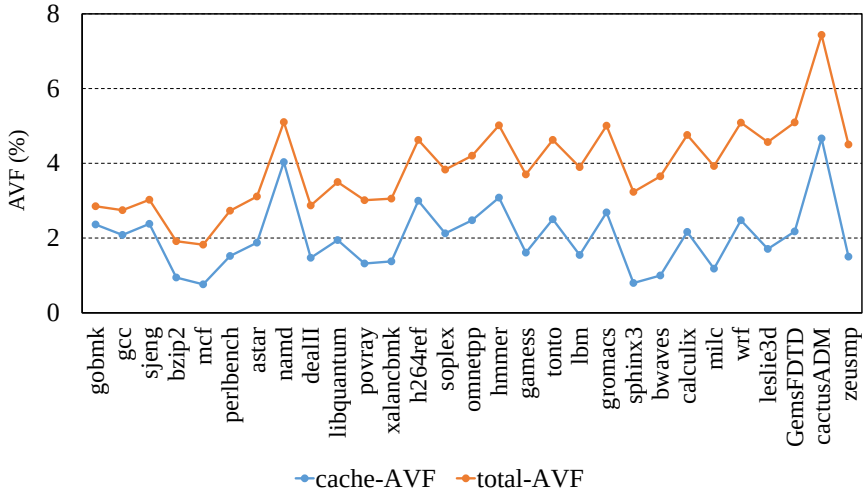

Figure 3.19: Cache-AVF and total-AVF for the SPEC CPU2006 benchmarks on a big out-of-order core.

right in Figure 3.19. This is because the benchmarks on the right-hand side of the graph have higher core-AVF, as previously reported.

3.10.4 Results

Figure 3.20 shows results for reliability-aware scheduling when ABC for the L1 caches is also taken into account. That is, total-ABC is used in Algorithm 1 as well as in Equation 3.3 for estimating SSER. We evaluate three cases while varying the size of the L1 caches for the small core. In the first case, the L1 caches for both the big and small cores are equal in size. In the other two cases, we reduce the size of the L1 caches for the small core by a factor of 2 and 4, respectively. When the L1 cache size is equal between the big and small core, the impact on reliability and performance is small. The reason is twofold: (i) the total amount of vulnerable state is dominated by the L1 caches, as described above, and (ii) execution time on the small core takes longer and as a result cachelines get exposed to soft errors for a longer duration, further narrowing the difference in vulnerable state between the big and small cores. Reducing the size of the L1 caches in the small core, the difference in vulnerable state increases between the big and small cores, which leads to significant average improvements in SSER by 5% and 11% for half the cache size and a quarter the cache size for the small cores, respectively. Note that performance is largely unaffected compared to random scheduling. These results demonstrate that reliability-aware scheduling is beneficial even if the L1 caches are unprotected and need to be taken into account as part of the scheduling policy. In addition, reliability-aware scheduling is more effective as cache size differs between the big and small cores.

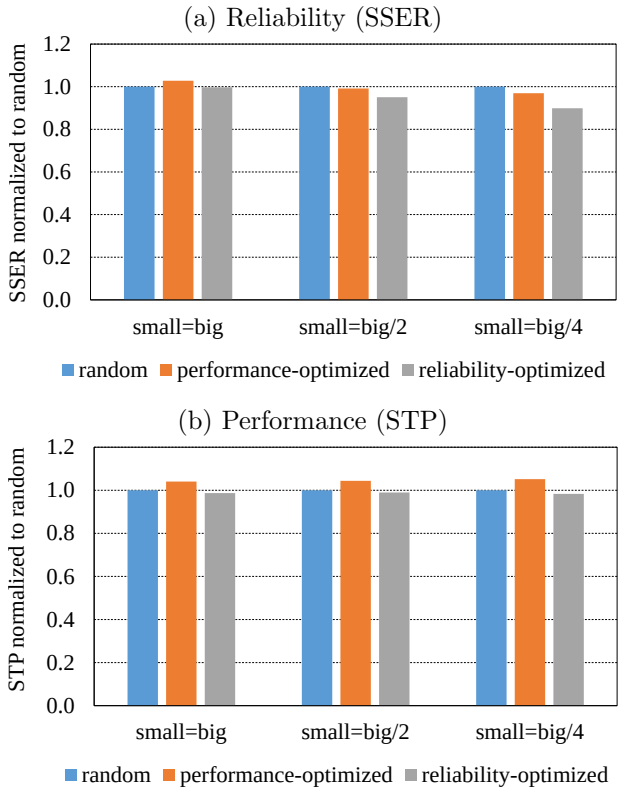

Figure 3.20: SSER (a) and STP (b) of four-program workloads on a 2B2S system with decreasing L1 cache size for the small cores.

3.11 Related Work

We now discuss related work in processor reliability, as well as recent work in scheduling for HCMPs.

3.11.1 Monitoring, Modeling and Improving Reliability

Processor reliability is a growing concern, and a significant body of prior work targets decreasing the occurrence of soft errors, either through radiation-hardened circuit design [30], error detection and correction mechanisms [153], or architectural techniques [190, 214]. Our scheduling technique is orthogonal to these approaches, and provides additional reliability improvements.

Other researchers have studied monitoring and modeling reliability for processor design (e.g., where to add error detection) and online reliability estimation (e.g., to find out when to enable an architectural error reduction technique that may also incur a performance hit). One way to evaluate soft error reliability is through fault injection, and to monitor what fraction of faults

lead to incorrect program executions [120]. Mukherjee et al. [137] propose ACE bit analysis as an alternative to fault injection to evaluate the reliability in architecture studies. They also introduce the concept of AVF. Biswas et al. [22] show how to measure AVF for address-based structures. Sridharan and Kaeli [191] propose to split AVF into PVF (program vulnerability factor) and HVF (hardware vulnerability factor), which can be determined independently. Other prior work models AVF through regression on performance counters [121, 210], or through analytical mechanistic modeling [145]. Nair et al. [144] develop a methodology for creating AVF-stressing benchmarks, providing a processor AVF upper bound.

No prior work has studied reliability characteristics of HCMPs, or has considered HCMP scheduling as a way to improve reliability. This dissertation is also the first to propose a system-level reliability metric for multiprogram workloads.

3.11.2 Scheduling Heterogeneous Multicores

Kumar et al. [112, 113] advocate single-ISA heterogeneous multicores to improve energy and power efficiency. Many proposals advocate scheduling compute-intensive applications on the big cores, because they show the highest performance improvement [39, 109, 179]. Van Craeynest et al. [207] show that memory-intensive applications can also show important performance gains on big cores if they are able to exploit more memory-level parallelism. Other proposals focus on optimizing energy efficiency [124] or power efficiency [138, 221]. This dissertation is the first to improve reliability on HCMPs through scheduling.

3.12 Summary

Applications exhibit different soft error reliability characteristics on big, out-of-order cores versus small, in-order cores. This provides considerable opportunity to improve system reliability through scheduling on HCMPs. An oracle offline analysis considering an HCMP with 2 small and 2 big cores shows that reliability-aware scheduling improves system reliability by 27.2% on average and up to 62.8%, while degrading performance by at little as 7% on average compared to performance-optimized scheduling.

In this chapter, we propose a reliability-aware scheduler that samples the reliability characteristics of running applications on either core type, and dynamically schedules applications on big versus small cores to improve overall system reliability. We propose a novel system-level reliability metric, system soft error rate (SSER), that weights per-application SER with its relative slowdown to account for the difference between small and big core performance. The proposed scheduler leverages a low-overhead (296 bytes per core) counter architecture to track hardware occupancy.

Reliability-aware scheduling improves system reliability by 25.4% on average and up to 60.2% compared to performance-optimized scheduling, while degrading performance by 6.3% only. The proposed scheduler is robust across core count, number of big versus small cores, and frequency settings. Moreover, as a side effect, reliability-aware scheduling reduces power consumption by 6.2% on average compared to performance-optimized scheduling. We also evaluate a power-optimized scheduler, in addition to the performance- and reliability-optimized schedulers. Compared to a random scheduler, optimizing for power leads to a degradation in performance and an improvement in reliability. For applications that cannot tolerate performance degradation below a certain threshold, we achieve the best system-level reliability within the threshold, and as the performance threshold is relaxed, the improvement in reliability also increases. Multithreaded workloads, due to their data-parallel nature, do not experience large improvements in reliability relative to the degradation in their performance. The vulnerable state in the on-chip L1 caches is almost $10\times$ more than the out-of-order core. Consequently, L1 caches dictate the improvement in reliability when ACE bits exposed by L1 caches are included as part of the reliability-aware scheduling policy.

Chapter 4

Dispatch Halting

Processor performance has increased exponentially over the years not only due to the continuous technology scaling but also because of the microarchitectural enhancements to extract more parallelism from application. To increase the degree of instruction-level parallelism (ILP), the core microarchitecture structures have increased in size with every technology generation. For example, the reorder buffer (ROB) and issue queue have increased from 128 and 36 entries in Intel's 2008 Nehalem microarchitecture (45 nm technology node), to 224 and 97 entries in the current Skylake microarchitecture (14 nm technology node), respectively [54, 90]. Larger structures contain more architectural state and therefore increase the vulnerability to soft errors. The memory-intensive workloads further exacerbate the soft error vulnerability of the core microarchitecture because a memory access typically stalls the processor for (at least) a couple hundreds of processor cycles. In particular, the processor back-end is occupied by a large microarchitectural state for a long time window after a long-latency load miss. Therefore, devising techniques that reduce the vulnerability to soft errors for memory-intensive workloads is thus of critical importance. Needless to say that such techniques should incur marginal overhead in terms of performance, power and chip area.

In this chapter, we propose *dispatch halting*, a microarchitectural technique to minimize soft error vulnerability in out-of-order processors. Dispatch halting stalls dispatch from the pipeline front-end into the back-end after a long-latency load miss to prevent instructions following the load from allocating back-end resources. These subsequent instructions are buffered temporarily in an extended micro-op queue (EMQ). Dispatch is resumed when the long-latency load is about to return. We consider two variants, proactive and reactive dispatch halting. Proactive dispatch halting predicts the load miss and, to preserve performance on par with a conventional out-of-order core, selectively copies loads and branches, along with their producer instructions, from the EMQ to the back-end for speculative pre-execution to expose memory-level parallelism (MLP) and resolve mispredicted branches that are independent of the long-latency load. In reactive dispatch halting, dispatch is halted when

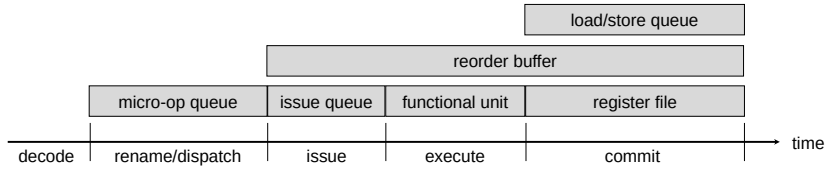

Figure 4.1: Timeline representing when entries allocated in back-end resources are ACE for committed instructions in a superscalar out-of-order core.

a long-latency load miss blocks commit at the ROB head for a given number of cycles. The instructions in the ROB turn into speculative execution mode to preserve MLP, and are flushed when the load is about to return. When exiting dispatch halting, the instructions buffered in the EMQ are dispatched into the back-end for normal execution. Overall, dispatch halting reduces the amount of vulnerable state upon long-latency load misses by holding instructions in a smaller structure while preserving out-of-order performance through (invulnerable) speculation.

Section 4.1 characterizes the vulnerability of an instruction as it occupies resources within different microarchitectural structures. Section 4.2 quantifies the vulnerability, calculated as ABC, for all benchmarks, and demonstrates that memory-intensive benchmarks are highly vulnerable to soft errors; it quantifies the potential for improving reliability and motivates our work on dispatch halting. Section 4.3 details the working of proactive dispatch halting on a pipeline in a stage-by-stage fashion. Section 4.4 explains reactive dispatch halting, which marks the processor back-end as speculative after a long-latency load instruction blocks commit for a certain number of cycles. The methodology is explained in Section 4.5, followed by the results in Section 4.6. We elaborate on the related work in Section 4.7 and summarize the chapter in Section 4.8.

4.1 OoO Core Soft Error Vulnerability

In an OoO core, instructions are fetched, decoded, and stored in a buffer called the micro-op queue (MQ) [201]. From the micro-op queue, the instructions are delivered to the back-end for execution. The process of sending instructions from the micro-op queue to the back-end of an OoO processor is known as *dispatch*. An instruction can only be dispatched to the back-end when there are enough back-end resources available to hold the instruction in the issue queue (IQ) and reorder buffer (ROB). A memory access instruction also allocates an entry in the load queue (LQ) or store queue (SQ). When an instruction's operands are ready and a functional unit (FU) is available, the instruction is issued: the instruction is removed from the IQ and executed on the FU. Once the instruction finishes its execution, its result is stored in the physical register file (RF), dependent instructions in the IQ are woken up and the reorder buffer is updated to reflect that the instruction is ready to commit. An instruction is committed when all instructions before it in program order

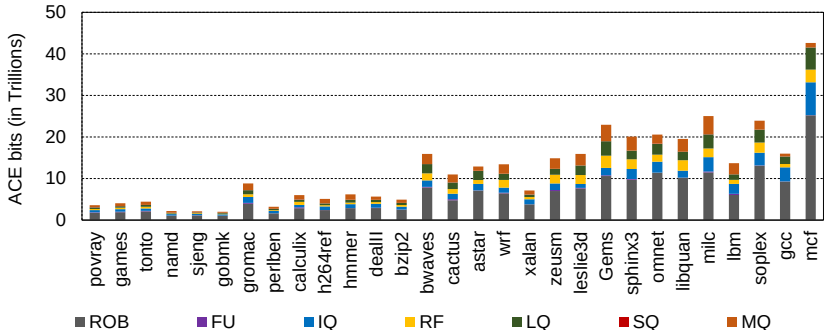

Figure 4.2: ABC stacks for an out-of-order core. *Memory-intensive workloads (on the right-hand side) lead to high ABC because of high occupancy in ROB, IQ, LQ and RF.*

have been committed. Upon commit, all back-end resources allocated to an instruction are released.

Figure 4.1 shows the duration when entries in back-end structures are ACE for correct-path instructions. For a correct-path instruction, all ACE bits exposed during the execution through an OoO core must be intact. Therefore, all the entries allocated for correct-path instructions in the micro-op queue, issue queue, ROB, register file and load/store queue must be correct from allocation to release. We therefore consider the time from dispatch to commit as ACE for the ROB entries. The address and data values in the load/store queue are considered ACE between issue and commit. An issue queue entry is considered ACE from dispatch to issue. Architecture registers are assumed ACE throughout the entire execution; a physical register is considered ACE between execute and commit, assuming a physical register transitions into an architecture register upon commit. The number of ACE bits exposed during execution on a functional unit is the bit width of the unit times the number of execution cycles per instruction.

4.2 Potential for Reducing Vulnerability

Soft error vulnerability is a function of the interaction between the application and the core's microarchitecture, i.e., the number of ACE bits (or ABC) on a particular microarchitecture depends on the application's execution characteristics. Figure 4.2 shows the ABC stacks for 1 B instruction traces for the SPEC CPU2006 benchmarks on an OoO core; the benchmarks are sorted from left to right by the number of last-level cache (LLC) misses per thousand instructions (MPKI), hence, memory-intensive benchmarks appear on the right hand side in the graph. (See Section 4.5 for our experimental setup.) An ABC stack breaks down the ABC of a core into its different microarchitectural structures. The higher ABC, the higher the total number of errors an application encounters. One important point to deduce from this figure is that the

memory-intensive applications lead to large occupancy inside the core. This is particularly the case for the reorder buffer, but also the issue queue, load queue and register file. As a result, the total number of errors encountered by memory-intensive applications is higher than for compute-intensive applications.

Figure 4.3 helps us understand the reason behind the high number of soft errors encountered by the memory-intensive applications. The 'OoO core' bars in Figure 4.3 are identical to the top height of the ABC stacks in Figure 4.2. Memory accesses as a result of LLC load misses frequently lead to full-ROB stalls, i.e., when a load miss blocks commit, new instructions are dispatched into the ROB until the ROB fills up, at which point new instructions can no longer be dispatched. To assess the impact of full-ROB stalls on reliability, we perform the following experiment. When a load instruction that misses in the LLC causes a full-ROB stall, we start counting ACE bits exposed in all structures of the pipeline by all in-flight instructions. When the load returns, we sum up the ACE bits exposed in all structures, and stop the counters. This is the reliability overhead caused by the LLC miss between the full-ROB stall and the return of the memory access. We repeat this process for every LLC load miss that results in a full-ROB stall. When we add the number of ACE bits exposed by all such loads, we obtain the reliability overhead caused by all full-ROB stalls. This reliability overhead is represented by the 'full-ROB stall' bar in Figure 4.3. Clearly, full-ROB stalls are responsible for a good fraction of soft errors encountered by the memory-intensive applications. For example, for benchmarks such as `omnetpp` and `libquantum`, more than 60% of the ACE bits are exposed during full-ROB stalls due to memory accesses. However, for applications such as `gcc` and `mcf`, full-ROB stalls lead to only 20% of the ACE bits. A large percentage of ACE bits in these applications are not exposed during the full-ROB stalls.

To understand the gap in reliability between full-ROB stalls and normal OoO execution, we perform another (but similar) experiment. When an LLC load miss reaches the head of the ROB and blocks commit, we start counting ACE bits exposed in all pipeline structures by all in-flight instructions. Unlike the previous experiment, we start counting ACE bits as soon as the LLC load miss blocks commit — we do not wait for the ROB to fill up as in the previous experiment. When the LLC load miss returns, we sum the ACE bits exposed in all structures, and stop the counters. This is the reliability overhead caused by the load instruction between blocking and unblocking commit. We repeat this process for every LLC load miss that blocks the ROB head. When we add the number of ACE bits exposed by all such loads, we obtain the reliability overhead caused by all memory accesses between ROB blocking and unblocking. This reliability overhead is represented by the 'ROB head blocked' bar in Figure 4.3. Obviously, the ACE bits exposed in this experiment also include the ACE bits exposed during full-ROB stalls, as a number of LLC load misses will result in full-ROB stalls. From Figure 4.3, we observe that most ACE bits for all memory-intensive benchmarks, including `gcc` and `mcf`, are exposed when the ROB head is blocked by LLC load misses (by 67% on average, and up to

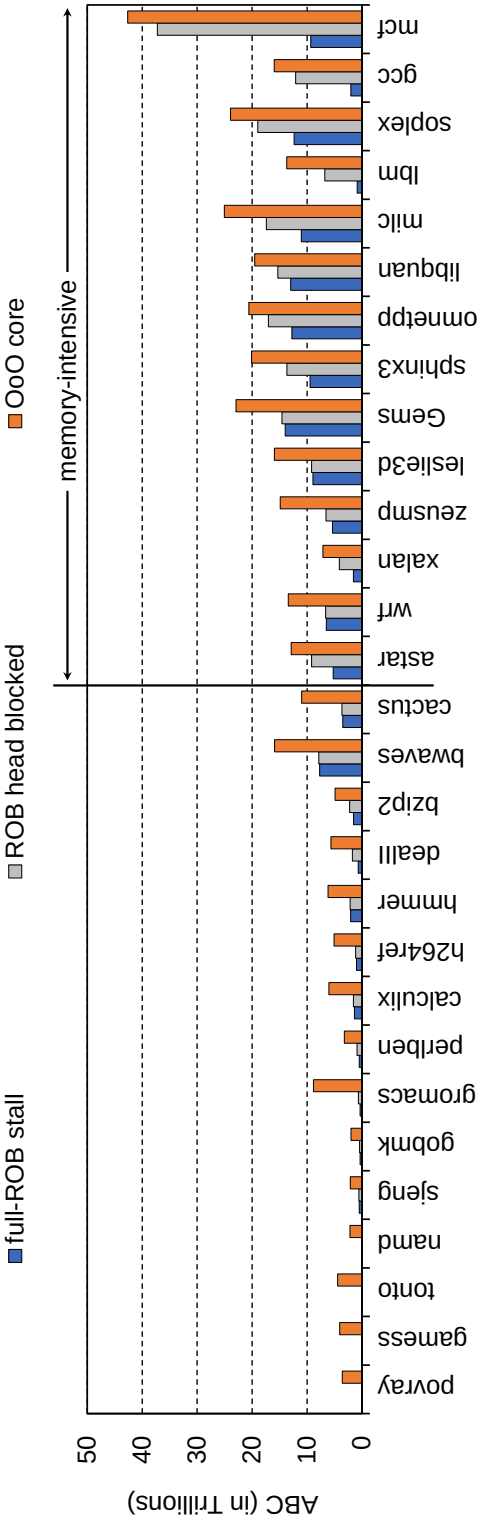

Figure 4.3: Impact of memory accesses on ACE bit count (ABC) on an out-of-order core. *A significant fraction of the total ACE bit count results from LLC load misses filling up the ROB and blocking the head of the ROB.*

87%). The fine subtlety between the two experiments is important for exploiting the full potential for improving reliability. A load instruction that blocks the head of the ROB may not necessarily lead to a full-ROB stall. However, a large number of ACE bits are still exposed to soft errors. We encounter such situations when an LLC miss is followed by a branch misprediction, front-end miss or a full issue queue. Benchmarks such as `gcc` and `mcf` have a large number of branch mispredictions in the shadow of long-latency load misses [145]; `lbm` is stalled on a full issue queue for about 20% of the time; `soplex` and `astar` also encounter branch mispredictions and some other resource stalls in the pipeline [147].

Therefore, to minimize the vulnerable microarchitecture state while a memory access is serviced, it is crucial to devise a mechanism that reacts as soon as an LLC load miss reaches the head of the ROB. However, it may take a couple tens of cycles (if not more) before an LLC load miss is detected and is the oldest instruction in the ROB. In the meantime, instructions fetched after the LLC load miss may have allocated OoO core back-end resources. We thus need a mechanism that acts much sooner, as soon as the LLC load miss blocks commit, or even sooner, as soon as the LLC load miss is dispatched. Ideally, we would only allow the LLC load miss to occupy back-end structures and prevent subsequent instructions from allocating OoO resources to minimize the amount of vulnerable state. This requirement motivates our work on proactive dispatch halting which we describe next. Alternatively, it is possible to mark an LLC missing load instruction and the instructions following it as speculative when the load instruction reaches the head of the ROB. A copy of these instructions can be buffered in the front-end and replayed when the blocking load instruction returns. Such a policy will re-execute all instructions beyond a blocking load. Reacting to an LLC miss by buffering and replaying instructions motivates our work on the reactive dispatch halting. Reactive dispatch halting is discussed in detail in Section 4.4.

4.3 Proactive Dispatch Halting

Proactive dispatch halting (P-DH) halts dispatch after a predicted LLC load miss and prevents subsequent instructions from allocating entries in the ROB, issue queue, load/store queue and register file, thereby reducing the amount of exposed vulnerable state. Obviously, we foresee countermeasures to preserve OoO performance. P-DH requires modifications to the microarchitecture. We add two new hardware structures between the decode stage and the dispatch stage: a *Load Miss Predictor (LMP)* and a *Producer Instruction Table (PIT)*. The LMP predicts whether a load instruction will incur an LLC miss, which will trigger halting dispatch. The PIT stores the addresses (PCs) of the instructions that a load or branch instruction depends upon through real register dependences. The overall microarchitecture of the core is shown in Figure 4.4. In this section, we explain the necessary hardware modifications required to ensure the flow of instructions through the pipeline in a reliable manner while

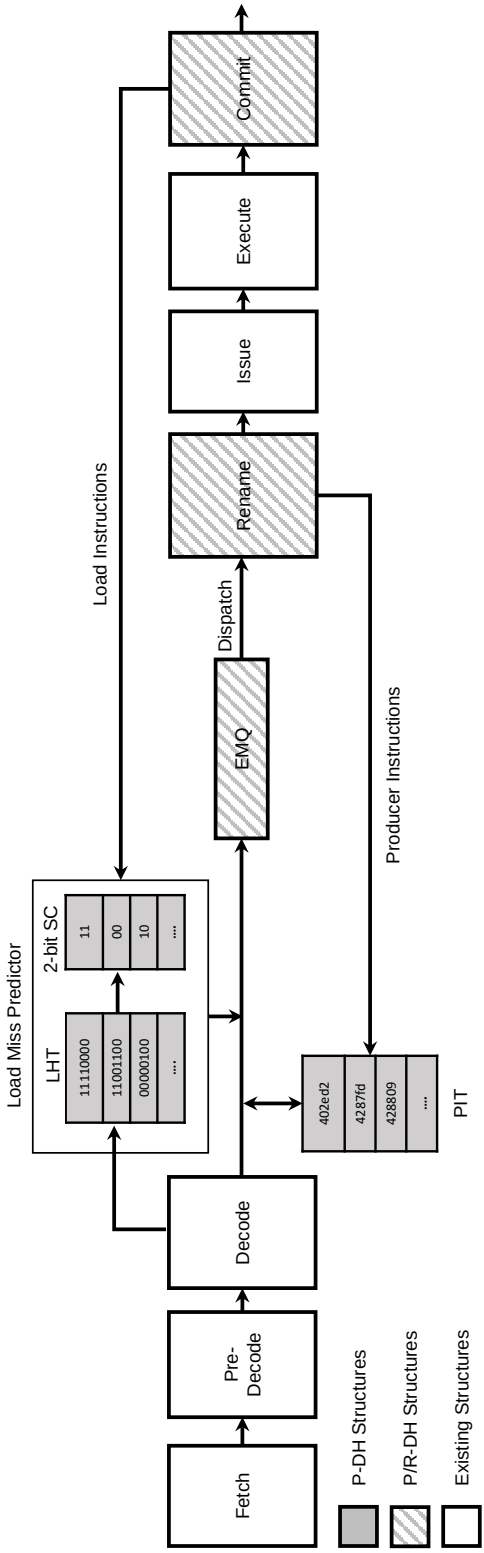

Figure 4.4: Core microarchitecture support for dispatch halting: the solid gray boxes are newly added structures under P-DH; the dashed gray boxes are modified structures required for both P-DH and R-DH; the white boxes are existing structures.

preserving OoO performance. As in a normal pipeline, instructions are fetched and (pre-)decoded. The decoder stage breaks up instructions in micro-ops which are then directed to a buffer called the Micro-op Queue (MQ).

4.3.1 Load Miss Prediction

A load instruction that misses in the LLC is most likely to also miss in the LLC in the near future. We track load LLC misses and their hit/miss history in hardware, and predict their future behavior using a Load Miss Predictor (LMP). The LMP in this work is a two-level predictor. Although other load miss predictors have been proposed, see for example [161, 196], we find a two-level predictor to be effective for our purpose. The first level, the Load History Table (LHT), indexed using 8 bits from the load PC, maintains the local hit/miss history of the past 8 instances of the load. The second level uses these history bits from the LHT as an index to a table with two-bit saturating counters. When a load instruction commits, its hit/miss status is updated in the LMP. The LMP is accessed after the decode stage and dispatch is halted when a load instruction is predicted to miss in the LLC. We refer to a load instruction that halts dispatch as a *halting load*.

In addition to the LMP, we also implement a halting-load counter that counts the number of halting loads that have passed the dispatch stage, i.e., the counter is incremented when a halting load is dispatched and is decremented when normal dispatch is resumed. The processor remains in dispatch halting mode as long as there is at least one halting load in-flight.

4.3.2 Halting Dispatch

Dispatching a halting load, i.e., a load instruction that is predicted to miss in the LCC, engages dispatch halting. Instructions following the halting load in the dynamic stream are buffered in the Extended Micro-op Queue (EMQ) — the EMQ is described in the following section. The process of preventing further instructions from being dispatched is known as a *halt*. Every halt has a corresponding *resume*, when we resume dispatching instructions from the EMQ into the back-end. We assume that dispatch can be halted at the granularity of a processor cycle. Therefore, depending on the width of the processor pipeline, all instructions selected for dispatch in a cycle are dispatched. If a halting load is one of them, the dispatch is halted from the following cycle onward.

4.3.3 Extended Micro-op Queue Operation

The micro-op queue streamlines the flow of micro-ops between the front-end and back-end of an OoO core by hiding potential bubbles created by different sources of micro-op generation in the front-end. Intel's core microarchitecture also introduced a structure called the Loop Stream Detector (LSD) to store small repeating loops from the micro-op queue. The micro-op queue is a circular

FIFO buffer present in most contemporary microarchitectures. For example, Intel's Haswell and Skylake microarchitectures [54] have a micro-op queue of 28 and 64 entries, respectively.

We exploit this already existing structure by extending it to store more instructions, and therefore call it the *Extended Micro-op Queue (EMQ)*. When a halting load is dispatched, and dispatch is halted, instructions start filling the EMQ. Once the EMQ is full, the front-end stalls. The instructions that were dispatched before the halting load execute and commit from the back-end of the pipeline and the only instruction occupying the back-end structures is the halting load, waiting for its data to return from memory. In a normal pipeline, there is a large number of instructions in the back-end waiting to be issued or committed, depending on the execution of their predecessors. When the data is about to return from main memory, dispatch is resumed and the instructions from the EMQ start allocating the out-of-order core back-end structures.

There are two performance issues with this (naive) proposal. First, a normal out-of-order core benefits from memory-level parallelism (MLP) by servicing multiple independent memory accesses while a long-latency load blocks the ROB head. Halting instructions beyond a halting load in the EMQ prevents the processor from exploiting MLP, thereby significantly degrading performance. Second, dispatch halting also delays branch resolution, which can also degrade performance if there are mispredicted branches in the EMQ that are independent of the long-latency load miss.

To maintain OoO performance, we propose to speculatively pre-execute all loads and branch instructions and their backward slices beyond a halting load, and resume normal execution when the long-latency load returns. More specifically, instructions beyond a halting load are stored in the EMQ up until the EMQ fills up. Loads and branches are copied to the back-end for speculative pre-execution; this happens in program order as they are inserted in the EMQ. In addition, we also copy the backward slices of the loads and branches to the back-end. A backward slice contains all the instructions that produce a register value that leads (directly or indirectly) to the load or branch. We keep track of backward slices in a hardware table called the Producer Instruction Table (PIT) as we describe in the next section. In other words, loads, branches and instructions whose PCs appear in the PIT — these are backward slice instructions — are copied to the back-end for speculative pre-execution. The instructions are *not* copied non-consecutively from the EMQ to the back-end. A micro-op is inserted in the EMQ after decode in program order. If it hits in the PIT, or if it is a load or a branch micro-op, the micro-op is also sent to the back-end. Note that speculatively pre-executed instructions in dispatch halting mode do not allocate ROB entries, hence they are never committed. Their purpose is only to improve MLP by generating prefetches and to resolve independent mispredicted branches as early as possible. When normal dispatch resumes, all instructions after the halting load are dispatched from the EMQ in the ROB and back-end of the processor for normal (non-speculative) execution, i.e., they are renamed, dispatched, issued, executed and committed in a typical out-of-order fashion.

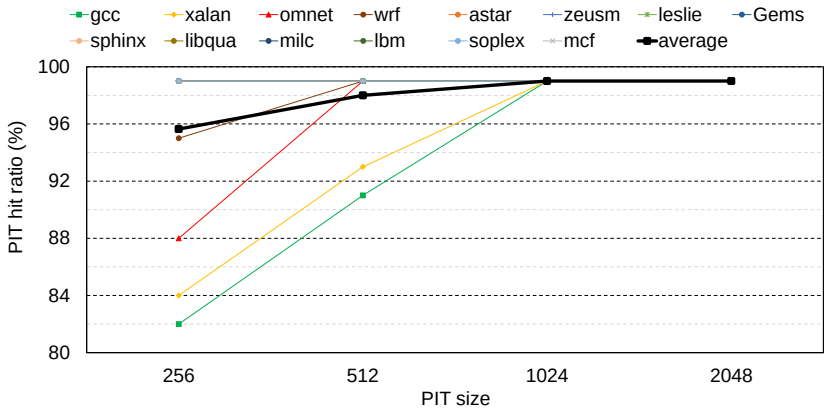

Figure 4.5: PIT hit rate as a function of its size for the memory-intensive benchmarks. *A 256-entry PIT leads to an average 95.6% hit rate; a 1K-entry PIT leads to a 99% hit rate for all benchmarks.*

It is worth noting that the amount of MLP generated and the number of resolved mispredicted branches when dispatch is halted depends on the size of the EMQ. If the size of the EMQ is as big as the ROB, we expect the performance of our reliability-aware microarchitecture to be close to a conventional out-of-order core (as we will confirm in the results section). Note that in some pathological cases, dispatch halting may lead to more MLP being exposed than in a normal OoO core. Consider a load instruction that is predicted to be an LLC hit, turns out to be an LLC miss. This load will block commit. Now, assume that a load further down the instruction stream is a halting load. This load will halt dispatch and will copy loads, branches and their backward slices to the processor back-end for speculative execution. The amount of MLP that can be exploited by the processor originates from the independent load misses between the blocking load and the halting load, plus the independent load misses in the EMQ beyond the halting load. Because the EMQ is assumed to be the same size as the ROB in our baseline experimental setup, this may lead to cases where the effective instruction window of in-flight instructions is larger than the ROB. As a result, dispatch halting may expose more MLP than a conventional OoO core, which leads to ultimately higher performance. We will quantify this second-order effect in Section 4.6.

4.3.4 Computing Backward Slices

Computing the backward slices for loads and branches is done iteratively across multiple executions of these instructions (e.g., across different iterations of a loop) by relying on register renaming logic, similar to what is proposed by Carlson et al. [33]. The instructions appearing in backward slices are kept track of in the *Producer Instruction Table (PIT)*. The Register Alias Table (RAT)[1]

[1]We use RAT to refer to front-end RAT; retirement RAT is only used in the reactive dispatch halting proposed in Section 4.4.

in an OoO core eliminates false register dependences by mapping Architectural Registers (AR) to Physical Registers (PR). We extend each entry of the RAT to also store the PC of the instruction that produces the latest value for a given AR. An additional mark bit is set if the instruction that writes to that AR is a load. When renaming an AR source register for a load or branch, the PC of the producer instruction is available through the RAT. This producer PC is added to the PIT unless the mark bit is set (in which case the producer instruction itself is a load). When decoding a producer instruction (i.e., PC appears in PIT) during the next execution of the instruction (e.g., next iteration of the loop), the RAT entries for its source operands will also have the PCs of their own producers. We then add these PCs to the PIT as well (again, unless the mark bit is set). This process iteratively builds up the backward slices for loads and branches across subsequent executions of these instructions, and stops when the backward slice hits another load. The PCs of the instructions in the backward slices are stored in the PIT. Over time, e.g., a couple iterations of a loop, the PIT contains the PCs of the instructions along the load and branch backward slices.

A key observation here is that the number of producer instructions is relatively small. Figure 4.5 quantifies the PIT hit rate as a function of its size. A 256-entry fully-associative cache is sufficient to capture 95.6% of the backward slice instructions for the memory-intensive workloads. For most applications, a 256-entry PIT leads to a 99% hit rate.

4.3.5 Resuming Dispatch

When resuming normal dispatch, instructions buffered in the EMQ are dispatched to the back-end. This happens when the halting load is about to return. Since memory access latency depends on several organizational parameters and run-time contention in the main memory subsystem, we do not know in advance the number of cycles required by the halting load to complete the memory access — average memory access time varies between 183 cycles (xalan) and 331 cycles (mcf). We therefore compute the Average Memory Access Time (AMAT) as a (simple) moving average across the last N LLC load misses (with $N = 64$ in our experiments)[2]. When a halting load is dispatched, dispatch is halted for a number of cycles equal to AMAT minus a constant proportional to the number of cycles required to fill the ROB. In the absence of stalls when dispatch can be maintained at the rate of the processor width, we assume that 40 cycles are sufficient to move all instructions from the EMQ to the ROB. For example, if the current AMAT equals 200 cycles, we resume dispatch after 160 cycles.

It is possible that a load instruction labeled as a halting load actually hits in the LLC, or L1/L2 caches. This leads to an incorrect halt, which likely hurts performance. We therefore resume dispatch when the load/store unit detects a tag hit at any of the cache levels. We squash the speculative instructions in the

[2]We rely on an efficient hardware implementation for computing a moving average.

IQ, LQ and SQ. The performance penalty of an incorrect halt is the time to access the cache tags at the different cache levels in a sequential manner, plus the time to send the signal back to the dispatch stage to resume. We assume a 2-cycle penalty for resuming dispatch after a tag hit. Therefore, in case of an L2 hit, the overall performance penalty will be the sum of the tag access times for L1 and L2, plus 2 cycles.

4.3.6 Flush Semantics

The RAT is periodically checkpointed for a fast recovery from branch mispredictions and exceptions. There can be several checkpoints maintained for quickly identifying and flushing wrong-path instructions from the pipeline [3, 133, 172]. An identifier (typically ROB index) associates each instruction in the pipeline to a checkpoint. In dispatch halting, we also checkpoint the RAT at every halt. Now, the RAT renames speculatively copied instructions; these instructions are then dispatched to the back-end for execution. As described in Section 4.3.3, there can be a mispredicted branch in the EMQ. In speculative mode, the checkpointing on a branch instruction is performed in a typical out-of-order fashion. The only difference is that the checkpoint holds the identifier of an instruction in the EMQ, instead of the ROB, and the branch predictor tables are updated with speculative state. If a branch instruction that is independent of a halting load is mispredicted, all instructions following the mispredicted branch instruction are squashed from the EMQ and the other front-end structures, and the fetch unit is redirected to the correct path. On a resume, the pipeline is restored to the checkpoint stored at the corresponding halt. Now, the execution continues in normal mode starting from the first instruction in the EMQ.

4.3.7 Microarchitecture Complexity Analysis

The total chip area overhead for implementing dispatch halting is limited to approximately 1.8 KB. Both tables of the LMP have 256 entries each, with 8 and 2 bits per entry for the first- and second-level tables, respectively, which leads to a total size of 320 bytes. For the PIT, we store the 32 least-significant bits from the address of the producer instructions, leading to a total size of 1 KB for 256 entries. The RAT amounts to 512 bytes: we assume 64 architectural registers and 8 bytes per RAT entry (9 bits for the physical register tag plus PC information of the last producer).

We model the EMQ as a circular buffer with 4 read and 4 write ports. The LMP is a two-level predictor with 2 read and 2 write ports, assuming at most two loads per cycle. We model the PIT as a fully-associative cache assuming 2 search ports and 1 write port. (Two search ports are enough because we only need a PIT access for producer instructions, which is a subset of the arithmetic instructions; we add at most one producer to the PIT per cycle, hence one

write port.) The RAT is modeled as an ECC-protected direct-mapped RAM structure with 12 read and 4 write ports [133, 162].

Using CACTI 6.5 [119], we estimate cycle time for the EMQ, LMP, PIT and RAT to be 0.216 ns, 0.179 ns, 0.362 ns, and 0.165 ns, respectively. This is below the processor cycle time (0.376 ns), hence there is no impact on processor timing.

We want to stress that none of the added structures are on the processor's critical path. If needed, accesses to the LMP and PIT can be pipelined. Moreover, we do not necessarily need to store the full PC per RAT entry. A few bits may be sufficient to distinguish the 256 entries in the PIT. Furthermore, the PIT is a predictive structure, hence we can tolerate some inaccuracy. Finally, updating the PC information in the RAT is not on the critical path; it can be spread out over multiple cycles if needed. The overall conclusion is that dispatch halting does not affect a processor's cycle time.

4.4 Reactive Dispatch Halting

P-DH, as described in the previous section, proactively halts dispatch after a predicted long-latency load miss and speculatively executes only select future instructions to generate MLP. P-DH's effectiveness hinges on the accuracy of the load miss predictor and incurs a modest hardware cost (i.e., LMP, PIT and extensions to the RAT). We therefore consider an alternative design option, namely *reactive dispatch halting (R-DH)*, which halts dispatch when a long-latency load miss is observed to block commit from the ROB for N cycles (with $N = 15$ in our setup). R-DH has the advantage to be more precise than P-DH in the sense that it targets all long-latency load misses, not just the ones that are correctly predicted. Moreover, R-DH does not incur any hardware cost. On the flip side, R-DH stalls dispatch relatively late — it is reactive — and a number of instructions have been dispatched into the processor back-end by the time dispatch is halted.

To address the latter issue and eliminate the amount of vulnerable state in the back-end under R-DH, we employ the following mechanism. When the load miss is observed to block commit for N cycles, we mark the processor back-end as speculative, i.e., instructions already present in the ROB continue to execute speculatively and generate MLP, however these instructions are never committed; these instructions hence do not expose any vulnerable state. Note that because we wait for the long-latency load to reach the ROB head *and* then wait for another N cycles before stalling dispatch, the ROB is expected to be partially (largely) filled with instructions, which enables the core to expose significant MLP, comparable to normal out-of-order execution. Triggering a pipeline flush when a long-latency load is detected, as proposed by Weaver et al. [214], leads to a significant performance drop (by on average 12% according to our experiments) because the core is unable to expose MLP.

Dispatch halting ends and normal execution resumes by flushing the instructions in the ROB and re-dispatching (and thus re-executing) instructions from the long-latency load. When to return to normal execution is determined using the same AMAT-based mechanism as P-DH, see Section 4.3.5. In order not to re-fetch and re-decode instructions after halting dispatch (to save power), we keep track of the instructions from the long-latency load in the EMQ. This requires a slightly different operation of the EMQ under R-DH compared to P-DH. For P-DH, instructions leave the EMQ as soon as they are dispatched. For R-DH on the other hand, we keep the instructions in the EMQ until they are committed. The EMQ thus contains all instructions in the ROB plus additional instructions that are yet to be dispatched. The circular EMQ keeps track of the ROB head, the ROB tail (equals the EMQ head, i.e., next instruction to dispatch) and the EMQ tail. Resuming normal dispatch is then as simple as assigning the EMQ head pointer to point to the ROB head.

The flushing mechanism in R-DH also works differently from the P-DH. In P-DH, we take a checkpoint at every halt, and the pipeline is restored to this checkpoint upon resume. In R-DH, since we cannot know in advance whether a load instruction will block the ROB head for a long duration, we cannot precisely checkpoint the state of the front-end RAT. When the back-end is marked as speculative, all the instructions in the ROB have already passed the rename stage of the pipeline, and they have updated the front-end RAT. For example, some of the branch instructions might insert a new checkpoint in the front-end RAT for fast recovery from branch mispredictions. Checkpointing upon every load instruction will incur a large overhead of maintaining the front-end RAT in R-DH. Therefore, to restore the pipeline to the point where the halting load is the next instruction to be dispatched, we copy the content of the retirement RAT to the front-end RAT upon resume. Retirement RAT maps each architectural register to a physical register that stores the committed value of instructions older to the halting load. When the mappings of the front-end and retirement RATs are same, the pipeline returns to the normal mode, and the instructions starting from the halting load are dispatched again from the EMQ to the back-end.

It is clear from the above discussion that R-DH presents a different trade-off than P-DH. Through experimental evaluation (see Section 4.6), we find that R-DH leads to higher reliability than P-DH because it targets all long-latency load misses. Moreover, it does not incur the hardware cost for predicting load misses and tracking backward slices for loads and branches. On the other hand, R-DH incurs a modest power cost because of re-executing *all* instructions from a long-latency load. In addition, R-DH also incurs a small performance degradation because of not exposing as much as MLP as under P-DH.

4.5 Methodology

In this section, we provide the details of our experimental setup to evaluate dispatch halting.

Table 4.1: Simulated baseline OoO core configuration.

Frequency	2.66 GHz
Type	out-of-order
ROB size	128, 120 bits/entry
Issue queue size	64, 80 bits/entry
Load queue size	64, 120 bits/entry
Store queue size	64, 184 bits/entry
Pipeline width	4
Pipeline depth	8 front-end stages
Functional units	3 int add (1 cyc)
	1 int mult (3 cyc)
	1 int div (18 cyc)
	1 fp add (3 cyc)
	1 fp mult (5 cyc)
	1 fp div (6 cyc)
Register file	120 int (64 bit)
	96 fp (128 bit)
EMQ	128 entry, 44 bits/entry, 4r, 4w
PIT	256 entry, fully assoc, 2r, 1w, 2s
LMP	2 × 256 entry, direct-mapped, 2r, 2w
L1 I-cache	32 KB, 4-way assoc, 2 cycles
L1 D-cache	32 KB, 8-way assoc, 4 cycles
Private L2 cache	256 KB, 8-way assoc, 8 cycles
Shared L3 cache	1 MB, 16-way assoc, 30 cycles
Memory	DDR3-1600, 800 MHz, 7.6 GB/s
	ranks: 4, banks: 32, page size: 4 KB
	bus: 64 bits, `tRP-tCL-tRCD`: 11-11-11

4.5.1 Experimental Setup

We use the most accurate, hardware-validated core model in Sniper 6.0 [32]. We augment Sniper to compute ACE bit counts in the micro-op queue, reorder buffer, issue queue, load/store queue, functional units and register file. NOPs and wrong-path instructions are considered un-ACE. As discussed in Section 4.1, we model a processor where physical register file entries are ACE from execute to commit; architectural registers are ACE throughout the execution of a program. We model a register file where a physical register transitions into an architectural register upon commit. This is similar to merged register file organization of the Intel Pentium 4, MIPS R12000 and Alpha 21264 processors [70, 86], where the same register file keeps speculative and (committed) architectural state of an application. There are two Register Alias Tables (RAT), we call them front-end RAT and retirement RAT, for mapping architectural registers to physical registers. The ROB, issue queue and load/store queue entries are allocated at dispatch. All occupied entries are freed at commit. The detailed configuration of the simulated (baseline) out-of-order core is provided in Table 4.1. We do not assume a hardware prefetcher in our baseline setup, however, we evaluate the impact of hardware prefetching in the evaluation.

Table 4.2: Per-entry details for the various pipeline structures in our baseline out-of-order core.

Structure	*Details*	*Bits/entry*
ROB	PC index: 12 bits; mapping: arch(7), phy(7), oldphy(7), 2 src, 1 dest, total = 21 × 3 = 63 bits; LQ and SQ index: 14 bits; ld, st, int, fp completion status, exception bits, marker bits; other control info	120
Issue queue	2 src, 1 dest reg tags: 21 bits; LQ and SQ index for address generation: 14 bits; micro-op: 32 bits; other control info	80
Load queue	VA and PA for memory-ordering violations: 96 bits; ROB ID: 7 bits; SQ index: 7 bits; fault bits; other control info	120
Store queue	Everything in load queue plus 64-bit data	184
EMQ	PC index: 12 bits; micro-op: 32 bits	44

4.5.2 Microarchitecture State

To quantify the impact on reliability, we need to know the sizes of all the structures in the core microarchitecture. We assume the sizes given in Table 4.2, which provides a justified balance among the various pipeline structures. We assume that the instruction fetch unit maintains a table that tracks all in-flight instructions between fetch and commit. This table maintains the PCs of all in-flight instructions, which incurs less hardware cost and state than propagating PC information throughout the pipeline. PC information is needed to index the branch predictor and to guarantee precise exceptions. Each ROB entry holds a 12-bit index to this PC table. The micro-op queue also stores this index and passes it to the ROB upon dispatch. The ROB also holds register tags (7 bits each) for mapping, completion statuses for different instruction types, exception bits, load/store queue index, and status/ready bits. We assume at most two source registers and one destination register. An issue queue entry holds opcodes, source and destination register tags, ROB ID, and load/store queue tags for sending address information. The load queue needs virtual and physical addresses (48 bits each) for handling TLB misses and memory ordering violations, the corresponding ROB ID, and different fault bits. In addition to the details required in the load queue, the store queue also stores data values (64 bits).

Note that the amount of state needed per EMQ entry (44 bits) is much smaller than the cumulative state per instruction once dispatched. An instruction that gets dispatched gets allocated an entry in the ROB (120 bits) and issue queue (80 bits), and in case of a load or store instruction, an entry is also allocated in the load queue (120 bits) or store queue (184 bits). In total, a non-memory instruction in the back-end occupies 200 bits, a load instruction occupies 320 bits and a store occupies 384 bits. The discrepancy between the amount of state per EMQ entry (44 bits) versus an instruction in the back-end (200–384 bits) is the primary reason why dispatch halting reduces the

vulnerability to soft errors. We account for the vulnerable state in all the microarchitecture structures including the EMQ, ROB, IQ, RF, LQ and SQ. Note also that we do not need to take the extra bits into account for the added hardware structures (LMP and PIT) to support dispatch halting because dispatch halting is a speculative technique. For the same reason, instructions executed speculatively under dispatch halting do not incur vulnerable state either.

4.5.3 Workloads

We consider all 29 benchmarks from SPEC CPU2006 for which we create representative 1B instruction SimPoints [180]. We report results for all benchmarks but specifically focus on the memory-intensive benchmarks. We define a workload to be memory-intensive if the number of LLC misses per kilo instructions (MPKI) exceeds 8. In all the graphs, the benchmarks are sorted by increasing order of memory intensity, with `astar` and `mcf` being the least and the most memory-intensive benchmarks, respectively, see Figure 4.3.

4.6 Results

We primarily focus on the memory-intensive workloads because that is where dispatch halting is most beneficial; nevertheless, we also report the impact on compute-intensive workloads. We use *MTTF* and *IPC* to quantify reliability and performance, respectively. We compare the following three configurations:

- OoO: Our baseline out-of-order core from Table 4.1.
- NO-SPEC: A naive solution to improve reliability is to simply halt dispatch upon a predicted long-latency load. Hence, there is no speculation and no MLP gets exposed.
- ONLY-LOADS: When dispatch is halted proactively, *only* load instructions and their producers available in the PIT are passed on to the back-end for speculative execution. Branch instructions and their producer instructions are not copied to the back-end. Therefore, mispredicted branches in the EMQ are not resolved.
- P-DH: Dispatch is halted proactively by predicting long-latency load misses, after which load and branch instructions and their producers are copied to the back-end for speculative execution.
- R-DH: Dispatch is halted reactively when a long-latency load blocks commit for more than $N = 15$ cycles. Instructions already in the ROB execute speculatively and are flushed when returning to normal execution.

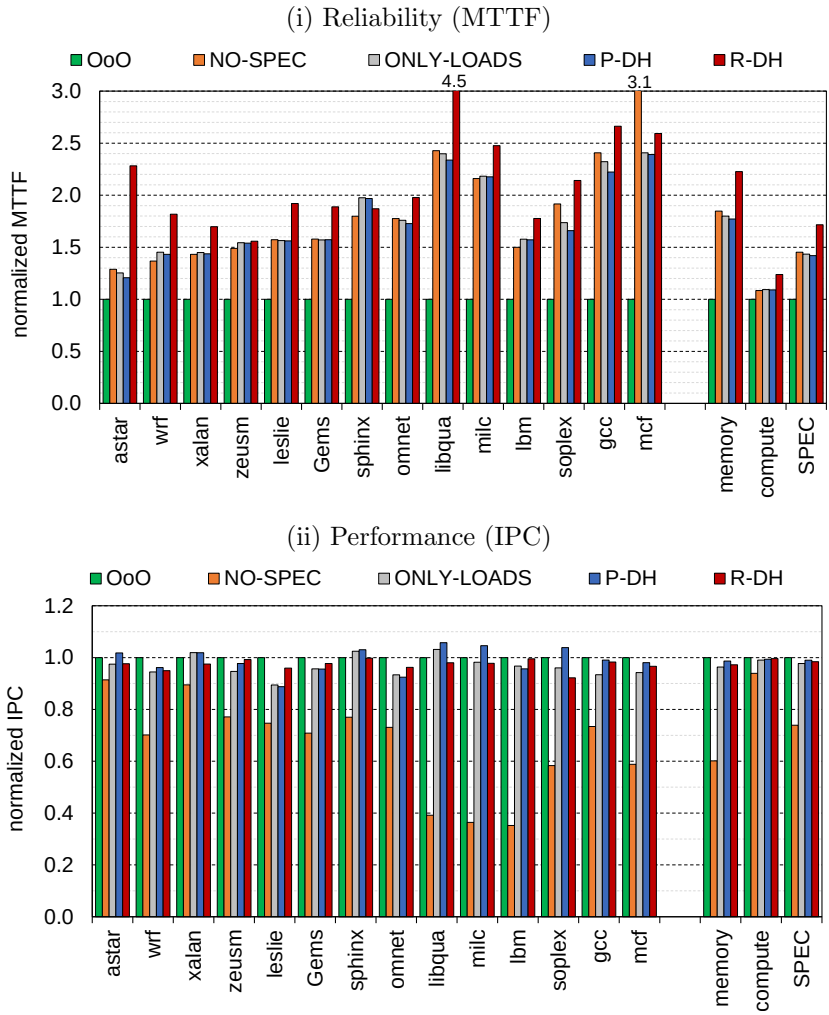

Figure 4.6: Effect of dispatch halting on (a) reliability (MTTF) and (b) performance (IPC). *Dispatch halting significantly improves reliability while maintaining performance compared to an OoO core.*

4.6.1 Reliability and Performance

Figure 4.6 evaluates the impact of dispatch halting on reliability (MTTF) and performance (IPC), normalized to the baseline out-of-order core. The key take-away message is that dispatch halting significantly improves reliability with a minor impact on performance, and R-DH improves reliability more than P-DH while degrading performance only slightly. More specifically, R-DH and P-DH improve reliability by on average 2.23× and 1.77×, respectively, for the memory-intensive benchmarks, while degrading performance by 2.8% and 1.3%. For the compute-intensive workloads, R-DH and P-DH improve

reliability by 1.23× and 1.09×, respectively. As expected, the improvement in reliability is much larger for the memory-intensive workloads, since dispatch halting, by design, aims at reducing the architecture state vulnerable to soft errors in the event of long-latency load misses. Across all benchmarks, R-DH and P-DH improve reliability by 1.72× and 1.42×, respectively. The higher improvement through R-DH compared to P-DH is because R-DH reduces the vulnerable state upon all long-latency load misses, not only the ones that are correctly predicted under P-DH.

Note that no-speculation is an unfavorable design point. Although reliability improves by 1.85× on average for the memory-intensive benchmarks, this comes at a cost of a performance hit by 34% on average. No-speculation minimizes the amount of vulnerable state but does not expose any MLP which has a detrimental impact on performance.

In the next two subsections, we analyze the impact on reliability and performance in more detail.

4.6.2 Reliability Analysis

Dispatch halting significantly reduces the number of exposed ACE bits in the OoO core back-end for memory-intensive workloads. We report substantial MTTF improvements in the 2× to 4.5× range for `libquantum`, `milc`, `gcc` and `mcf`, see Figure 4.6(a). `libquantum` and `milc` are benchmarks for which a load instruction that blocks the head of the ROB frequently leads to a full-ROB stall; this also explains why R-DH outperforms P-DH by a significant margin, especially for `libquantum`. For `mcf` and `gcc`, it is crucial to not wait for the full-ROB stall, but to halt immediately after dispatching a (predicted) long-latency load instruction (see also Section 4.2). For `lbm`, which frequently blocks on the issue queue before the ROB is full after an LLC miss, dispatch halting improves MTTF by 1.57× (P-DH) and 1.78× (R-DH).

For the moderately memory-intensive workloads, see the benchmarks `wrf` (second to the left) till `omnet` in Figure 4.6(a), the improvements in reliability range between 1.4× to 2×. The `astar` benchmark (left-most benchmark) leads to a modest reliability improvement of 1.2× under P-DH. Aliasing in the load miss predictor for `astar` causes several load PCs to map to the same entry in the first-level table, increasing the number of LMP mispredictions. Up to five load instructions are mapped to the same LHT entry for `astar` and this leads to incorrect halts when an LHT entry is shared by a load with a high LLC miss probability and a load with a high LLC hit probability. As we describe in Section 4.6.6, there is significant potential for improving reliability for `astar` using a better load miss predictor. R-DH does not suffer from limited load miss predictions and improves reliability by 2.28× for `astar`.

4.6.3 Performance Analysis

Dispatch halting leads to an average IPC degradation of 0.9% (P-DH) and 1.6% (R-DH) compared to an out-of-order core across all benchmarks. The degradation for R-DH is consistent across all benchmarks (and at most 7.8%) because R-DH exploits less MLP. For P-DH, we observe some variability across workloads, ranging from an improvement of 5.7% for `libquantum` to a degradation of 11.2% for `leslie`. We now explore the impact of proactive dispatch halting on performance in more detail.

P-DH may lead to a performance improvement in case of an LLC load miss followed by a halting load, as described in Section 4.3.3. Assume we incorrectly predict the first load to be a hit. The instructions between these two loads are in the ROB and commit blocks on the first LLC load miss. However, since dispatch is halted by the second load instruction, we start sending loads, branches and their producers to the issue queue for speculative execution. When the EMQ is full, the front-end pipeline stalls. This situation leads to a situation in which the processor can exploit more parallelism in the memory hierarchy than a normal OoO core can. A number of benchmarks experience such an improvement, most notably `sphinx3`, `libquantum`, `milc` and `soplex`, see Figure 4.6(b).

The proactive dispatch halting may also degrade performance for a couple reasons. First, performance may degrade if we predict a load to be an LLC miss which turns out to be a hit. The incurred penalty is the number of cycles from dispatch to issue for the halting load, plus the number of cycles to detect the cache hit. This is the case for a couple benchmarks including `mcf`, `soplex` and `xalancbmk`. Second, when dispatch is halted, the number of speculatively executed load instructions is limited by the size of the PIT. A load instruction cannot execute speculatively if its producer(s) are not found in the PIT, thus affecting performance. This is the primary reason for the degradation in IPC for `omnetpp` (88% hit rate in the PIT, see also Figure 4.5). `gcc`, `leslie3d` and `xalancbmk` also experience some performance degradation due to a larger set of producer instructions than what the PIT can accommodate.

Both P-DH and R-DH resume dispatch based on the average memory access latency (AMAT) as previously explained[3]. However, the effective memory access latency varies depending on the amount of contention in the memory subsystem. Whenever the effective memory access latency is smaller than the average, dispatch halting may degrade performance. We find this to be the case for primarily `leslie3d`, but also `zeusmp`, `wrf` and `GemsFDTD` in case of P-DH.

[3]Using per-load average memory access latency did not have significant impact on our results.

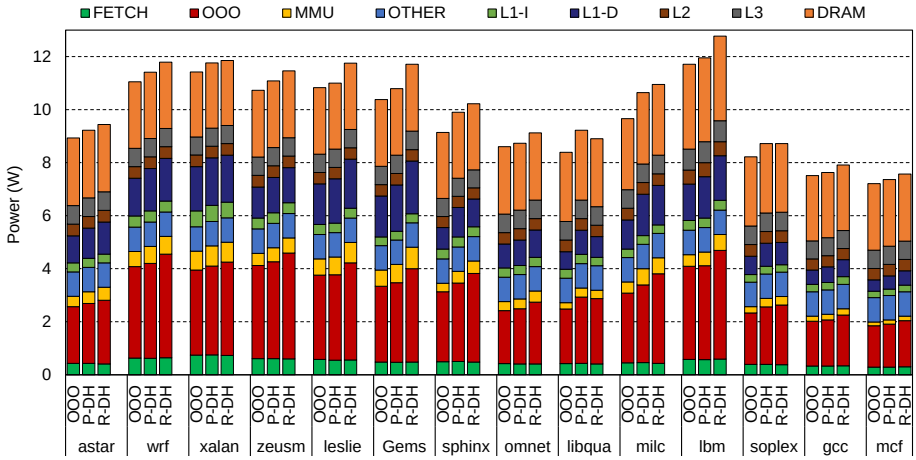

Figure 4.7: Impact of dispatch halting on power consumption. *P-DH and R-DH increase system power by on average 2.7% and 6.2%, respectively.*

4.6.4 Mispredicted Branches

Before halting the dispatch, the R-DH has already dispatched several instructions to the back-end. This not only helps in generating MLP but also resolves mispredicted branches in the ROB. However, in proactive dispatch halting, it is important to speculatively execute both loads *and* branches (and their producer instructions) upon halting dispatch. Reliability is not affected by whether branches are executed speculatively or not, see Figure 4.6(a). However, we note a non-negligible performance degradation for a number of benchmarks (by 3.6% on average for the memory-intensive benchmarks) if we do not execute branches speculatively under halted dispatch, see Figure 4.6(b). Speculatively executing branches allows the processor to resolve mispredicted branches earlier (if they are independent of the outstanding load misses), which enables the processor to re-direct fetch earlier towards the correct path. Speculatively executing branches under P-DH improves performance by 7.8% for `soplex`, 6.4% for `milc` and 5.7% for `gcc`; other benchmarks such as `astar`, `zeusmp` and `mcf` also benefit from this optimization.

4.6.5 Power Consumption

Dispatch halting increases power consumption because (select) instructions are executed twice upon a dispatch halt. These instructions are first executed speculatively and then re-executed. The purpose of the speculative execution is to expose MHP and resolve mispredicted branches. In addition, the extra structures for P-DH, i.e., the PIT and LMP, also consume power. In this section, we now quantify the increase in power consumption due to dispatch halting. We use McPAT [118] and its integration with Sniper [82] to quantify chip-level and total system power assuming a 22 nm chip technology. We model the PIT and LMP to support P-DH, and we assume that the EMQ is part of

the baseline OoO core. We estimate power (and chip area) for these three structures using CACTI 6.5 [119] and add those numbers to the McPAT power numbers. We do account for leakage power in all of these structures.

Figure 4.7 reports power consumption for dispatch halting relative to a baseline OoO core. System power is broken up into different components including core power (fetch, OoO, MMU and other), L1, L2, LLC and DRAM power. P-DH increases total system power by 2.7% on average, and by 4.5% for the memory-intensive workloads. The PIT and LMP's contributions to total power is small, less than 1%. The increase in power consumption is thus primarily a result of the increased number of instructions executed. R-DH increases total system power by 6.2% on average, and by 7.4% for the memory-intensive benchmarks. This increase in power consumption is a result of executing instructions twice upon a dispatch halt.

It is interesting to note that the 27% and 42% increase in executed instructions under P-DH and R-DH, respectively, leads to a relatively small increase in power consumption of 4.5% and 7.4% in total system power for the memory-intensive workloads. The fundamental reason is that the speculatively executed instructions only increase activity in the back-end; they do not affect front-end processor power consumption and they do not (significantly) affect power consumption in the memory hierarchy. Instructions are fetched and decoded only once and stored in the EMQ. In addition, speculatively executed loads trigger prefetches from the next level in the memory hierarchy which would happen as regular accesses in a conventional OoO core. The increase in processor activity is a result of renaming, scheduling and executing the speculative instructions twice.

4.6.6 Potential vs. Achieved Reliability

Recall that Figure 4.3 reports that the ACE Bit Count (ABC) of an OoO core can be improved by 67% on average across the memory-intensive benchmarks assuming that we withhold from allocating OoO back-end resources upon an LLC load miss blocking commit. However, P-DH and R-DH improve ABC by on average 41.2% and 50.4%, respectively. We now investigate this gap, see also Figure 4.8. We identify three different intermediate (idealized) policies between P-DH and the potential. If we were to know the per-load memory access latency ahead of time and resume dispatch at that time (rather than the average memory access latency) minus 40 cycles — this is the PER-LOAD policy — we would be able to improve ABC by 46.1% on average. Resuming dispatch at the time the LLC load miss effectively returns (and not 40 cycles earlier) — this is the RETURN policy — improves ABC by 47.5%. Assuming a perfect load miss predictor that identifies all LLC load misses at the dispatch stage — this is the PERFECT-LMP policy — further improves ABC by 60%. It is interesting to compare P-DH and the above policies against R-DH. This analysis suggests that for most benchmarks, there is significant headroom to improve P-DH beyond R-DH by improving the load miss predictor. For two benchmarks, `astar` and `libquantum`, R-DH fundamentally outperforms P-DH

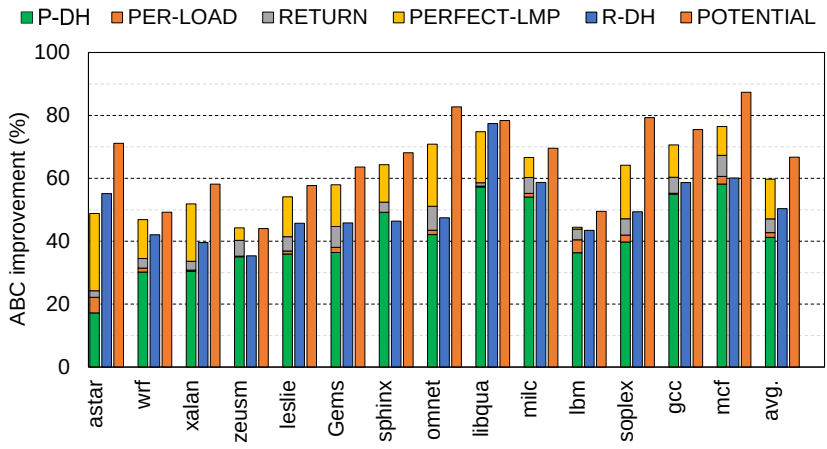

Figure 4.8: Comparing achieved versus potential reliability improvement through dispatch halting. *P-DH and R-DH yield a 41.2% and 50.4% ABC improvement, respectively, versus a potential 67% improvement.*

because P-DH is unable to correctly predict load misses at dispatch time, i.e., a load that is anticipated to be a hit at dispatch time turns out to be a miss by the time the load is executed, because of a cache eviction by another intervening memory operation. Overall, we find that improving the load miss predictor is the most promising lead to further improve reliability through proactive dispatch halting; we leave this for future work.

The remaining gap between the perfect load miss predictor and the full potential is a result of second-order effects. In particular, a load that is anticipated to be a hit at dispatch time may turn out to be a miss by the time the load is executed, because of an eviction by another memory operation. This leaves some room for further improvement.

4.6.7 Runahead Execution

Runahead execution [57, 79, 80, 139, 141, 142] is a well-known technique to exploit distant MLP beyond the ROB. The key idea is to enter speculative execution when the ROB completely fills up because of a long-latency load miss blocking commit at the ROB head. These speculatively executed instructions prefetch data into the processor's caches to speed up normal execution when the initiating load miss returns. Although runahead execution was designed to improve performance, it also improves reliability because the instructions executed underneath a long-latency load miss are speculative.

We implement the enhanced runahead execution [141] which satisfies two conditions: (1) there are no overlapping runahead intervals, and (2) runahead execution is triggered only when the long-latency load blocking the head of the ROB was issued to memory less than 250 cycles earlier. Figure 4.9 compares runahead execution (RA) against dispatch halting in terms of reliability, per-

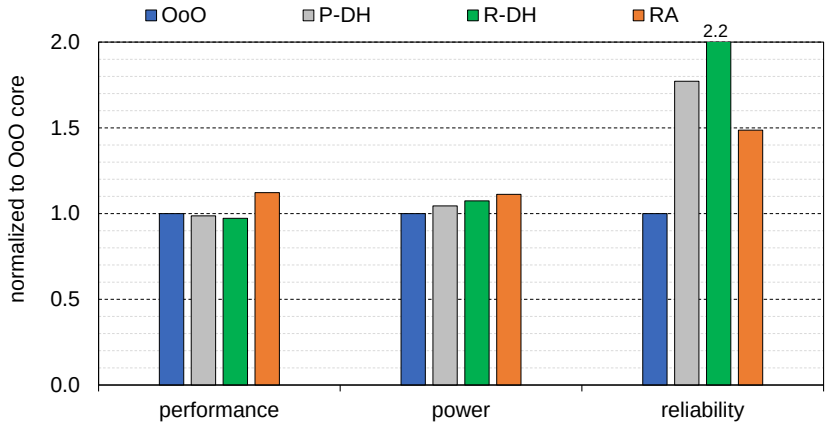

Figure 4.9: Comparing performance, power, and reliability of runahead execution versus dispatch halting for the memory-intensive benchmarks. *Runahead is less reliable than dispatch halting while consuming more power.*

formance and power. Runahead execution improves performance by 12.4% on average compared to conventional out-of-order execution, and by 13.5% and 15% compared to P-DH and R-DH, respectively. Yet, runahead is not a compelling solution in terms of reliability. Reliability is not as good as R-DH and P-DH, i.e., 1.48× improvement for the memory-intensive benchmarks versus 1.77× and 2.23×, respectively, while incurring 11.2% more system power than an out-of-order core (compared to 4.5% and 7.4% for P-DH and R-DH, respectively). The power cost for runahead is higher because (i) runahead executes more instructions speculatively than R-DH, (ii) runahead executes all instructions speculatively whereas P-DH only executes load and branch slices, and (iii) runahead re-fetches and re-decodes instructions whereas dispatch halting keeps track of the speculatively executed instructions in the EMQ. The reliability improvement is less because runahead execution (i) enters speculative execution when the ROB completely fills up upon a long-latency load miss blocking the ROB head (which is suboptimal, see also Section 4.2), and (ii) runahead execution is not triggered for short runahead intervals to limit its runtime overhead. For memory-intensive workloads, the ROB head is blocked by a long-latency load for 73% of the time, while the processor is blocked on a full ROB for only 35% of the time. In other words, the ROB head is blocked while the ROB is not completely full for 38% (= 73% − 35%) of the time. Because runahead execution (and runahead buffer [79]) is only triggered upon a full-ROB stall, this implies that runahead leaves significant opportunity on the table. Dispatch halting on the other hand enters speculative execution proactively (P-DH) or soon after the load miss blocks the ROB (R-DH), which leads to much improved reliability.

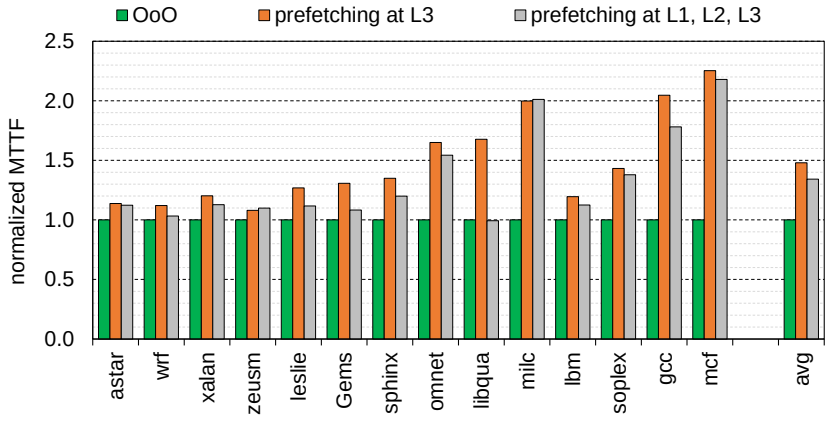

Figure 4.10: Normalized MTTF for dispatch halting relative to an OoO core with hardware prefetching enabled. *Dispatch halting significantly improves reliability on an OoO core with hardware prefetching.*

4.6.8 Sensitivity Analyses

We now perform a number of sensitivity analyses for proactive dispatch halting with respect to hardware prefetching, EMQ size and multiprogram workloads in a multicore setup.

Hardware Prefetching. We find dispatch halting to be effective even with a hardware prefetching enabled processor, see Figure 4.10. We consider two hardware prefetch scenarios: a (stream) prefetcher at the LLC versus a prefetcher at all three cache levels. We observe an average performance improvement by 40.1% and 66.6% for our memory-intensive benchmarks, respectively. Compared to an OoO core baseline with hardware prefetching, dispatch halting improves reliability by 48% and 34.2% on average, respectively. Although the improvement is less than for our baseline OoO core without prefetching (77.2% improvement), the improvement is still substantial. In terms of performance, dispatch halting improves performance by 4.1% and 1.7% for the LLC prefetcher and prefetching at all cache levels, respectively. We thus conclude that dispatch halting improves reliability for the cases where the hardware prefetcher is unable to adequately prefetch memory requests.

EMQ Size. Figure 4.11 shows the impact of EMQ size on performance and reliability. As explained in Section 4.3.3, the size of the EMQ impacts the amount of MLP that can be exploited when dispatch is halted. This has a direct effect on performance. The processor is unable to exploit as much MLP as a normal OoO core can if the EMQ is smaller than the ROB, which leads to an average performance degradation by 10% and 17% for an EMQ of size 64 and 32, respectively. A smaller EMQ however leads to improved reliability because the EMQ holds less vulnerable state. Changing the size of the EMQ thus exposes a trade-off in performance versus reliability.

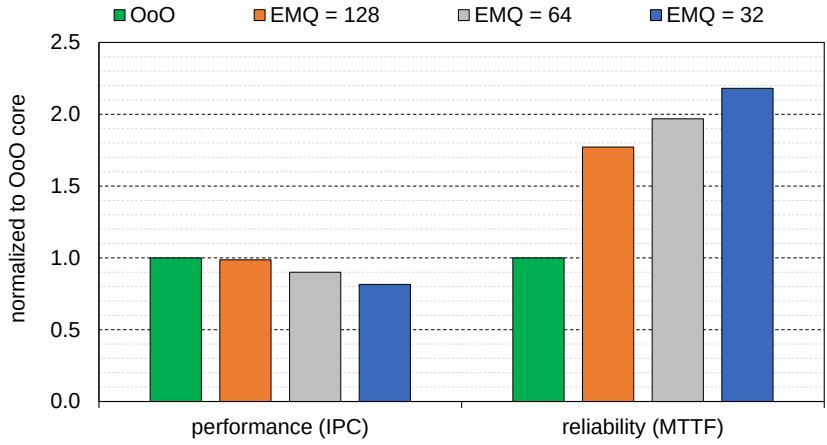

Figure 4.11: Impact of EMQ size on performance and reliability. *EMQ size exposes a trade-off in reliability versus performance, i.e., smaller EMQ size leads to improved reliability at the cost of a degradation in performance.*

Multiprogram Workloads. We now evaluate dispatch halting in a multicore processor context in which multiple independent programs co-execute. We assume randomly chosen multiprogram workload mixes with 2, 4 and 8 memory-intensive benchmarks (no benchmark replication). Because co-executing programs impact each other's performance and reliability, we use System Soft Error Rate (SSER) [147] as defined in Chapter 3 and System Throughput (STP) [60] to measure multicore reliability and performance, respectively; MTTF is computed as the inverse of SSER. The L1 and L2 caches are private to each core, and the LLC is shared by all cores. We assume a 2, 4 and 8 MB LLC for 2, 4 and 8 cores, respectively.

Figure 4.12 reports the impact of dispatch halting on MTTF and STP for the multiprogram workloads. The important observation here is that MTTF improves with increasing core count while keeping performance largely unaffected — MTTF improves by 80.1%, 90.1% and 98% with 2, 4 and 8 cores, respectively. The improvement in MTTF can be understood intuitively as increased core count leads to an increase in LLC contention and thus higher LLC miss rates. An increased number of LLC misses leads to a higher potential for improving reliability through dispatch halting.

4.7 Related Work

Researchers have proposed several methods to address the problem of soft error reliability in microprocessors over the past two decades. While some techniques focused on estimating [120, 137, 144] and modeling [22, 145, 191, 210] soft errors, others were aimed at improving soft error reliability [30, 147, 153, 190, 199, 214]. Earlier techniques used radiation-hardened circuits [30] or some form of redundancy for detection and recovery from soft errors [67, 170,

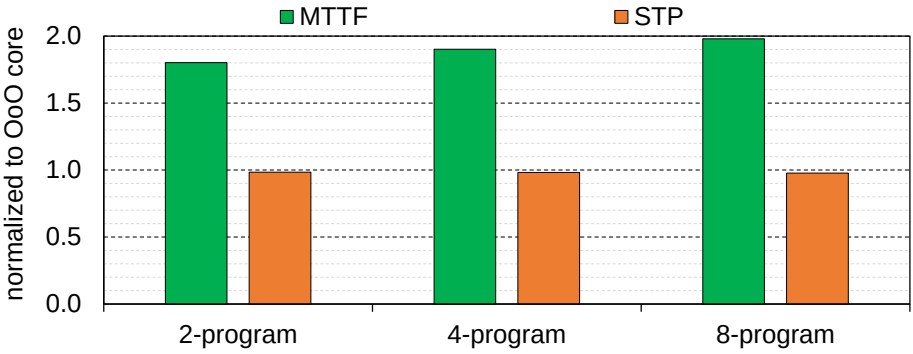

Figure 4.12: MTTF and STP for two-, four- and eight-program workloads. *Dispatch halting improves reliability with increasing core count while keeping performance largely unaffected.*

171, 186, 198, 209]. However, these techniques incur significant performance, area and power overheads [136]. To reduce these overheads, Soundararajan et al. [190] apply reliability enhancing techniques only when the reliability constraints are not met. They propose vulnerability control mechanisms for the ROB by trading off performance and reliability at runtime. Weaver et al. [214] propose fetch throttling and squashing instructions on a load miss to prevent instructions from sitting idle in the back-end structures of an in-order core. Qureshi et al. [166] detect soft errors by performing redundant execution upon a long-latency cache miss; their approach leads to a 7.1% IPC degradation for memory-intensive applications.

ECC and Parity. Address-based structures like L2/LLC and main memory are commonly protected with ECC, while L1 caches, TLBs and BTBs are protected with EDC (e.g., parity) or ECC [8, 130, 189]. For example, the L1-D, L1-I and TLBs of the Intel Xeon Phi chip are protected with parity; the L2 is protected with ECC [90]. The other structures are not as easily protected against soft errors. In particular, coding techniques such as ECC cannot be applied to latency-sensitive pipeline structures such as the ROB, IQ, etc., as it adds additional latency to each cycle [26, 40, 210]. Parity can bring significant reliability improvements, however, the area, power and energy overheads of parity are close to 14% for an OoO core, and are even higher for an in-order core [40].

Latency-Tolerant Operations. Proposals aimed at improving latency tolerance or power efficiency, also likely improve reliability [115, 127, 195, 198]. Runahead execution is a representative technique in this category and we have devoted Section 4.6.7 to evaluate how runahead execution affects soft error reliability. Waiting Instruction Buffer (WIB) [115] drains miss-dependent instructions from the issue queue to a large buffer to leverage issue queue space for executing independent instructions under an LLC miss. Continual flow pipelines [195] unblock the scheduler and register file for executing miss-independent instructions. Long-term parking [174] allocates back-end pipeline

resources as late as possible for saving power, but still allocates ROB entries for all instructions past a long-latency load miss. Fetch halting [127] requires offline profiling to improve power efficiency by reducing the occupancy in the issue queue and reorder buffer. Fetch halting requires offline profiling of load criticality and leads to an overall 6.5% degradation in IPC. Note that performance degradation causes a program to run longer, which in turn leads to increased vulnerability to soft errors. Overall, these prior proposals (still) expose a major portion of the pipeline to soft errors under an LLC miss. Dispatch halting, on the other hand, exploits speculation over normal execution when a LLC miss is detected (R-DH) or predicted (P-DH), and significantly improves reliability at low performance, power and area overheads.

4.8 Summary

Transient faults lead to a major reliability challenge in modern-day computer systems. This is particularly problematic for memory-intensive workloads as a large vulnerable state is exposed upon a long-latency load miss in an OoO core. We propose dispatch halting to address the issue of high vulnerability of memory-intensive applications on out-of-order cores. We propose proactive and reactive dispatch halting, which halt dispatch upon a predicted load miss versus upon a load miss blocking commit, respectively. Instructions are temporarily buffered in an extended micro-op queue — P-DH copies loads, branches and their producer instructions to the back-end for speculative pre-execution, whereas R-DH speculatively executes the instructions already present in the back-end. Normal execution is resumed when the long-latency load is about to return.

Dispatch halting significantly improves soft error reliability with marginal impact on performance. The key insight behind dispatch halting is that the amount of vulnerable state in the EMQ is much smaller than the cumulative state allocated in the processor back-end structures (ROB, IQ, LQ and SQ) upon a long-latency load miss *and*, in case of P-DH, the hardware structures for speculative pre-execution are all predictive, which, by construction, are not vulnerable to soft errors. P-DH and R-DH provide different trade-offs in reliability, performance, power and hardware cost. R-DH yields the highest improvement in MTTF, by 1.72× across SPEC CPU2006 (and by 2.23× for the memory-intensive benchmarks), at no additional hardware cost while incurring the highest, albeit modest, increase in power consumption (6.2% increase in total system power). P-DH on the other hand improves MTTF by 1.42× on average (1.77× for the memory-intensive benchmarks), while increasing total system power by only 2.7%, and while incurring a hardware cost of 1.8 KB. R-DH and P-DH degrade performance by only 1.6% and 0.9% on average, respectively.

Chapter 5

Precise Runahead Execution

Memory performance has increased at a much slower rate than processor over the years. A slower memory stalls the processor frequently, as memory access instructions take longer to retire from the pipeline than other instructions. Computer architects have tackled the issue of long memory access time by devising latency-tolerant techniques like out-of-order execution, deeper cache hierarchies, and sophisticated hardware prefetching. Runahead execution [57, 139, 141] is also a latency-tolerance technique that improves performance by accurately prefetching long-latency loads. The processor triggers runahead execution when a long-latency load causes the instruction window to fill up and halt the pipeline. Instead of stalling, the processor removes the blocking long-latency load and speculatively executes subsequent instructions to uncover future independent long-latency loads and expose memory-level parallelism (MLP). The processor terminates runahead execution and resumes normal operation when the stalling load returns. Because runahead execution generates memory loads by looking at the application's code ahead of time, the prefetch requests it generates are accurate, leading to significant performance benefits.

In this chapter, we take a closer look at the performance bottleneck created by long-latency loads, and understand how runahead execution and its follow-up techniques have been able to successfully improve the performance under long-latency load misses. We further sift through the microarchitectural working of runahead techniques and find that there is still a potential for further improving their performance. Our mechanism to filter out the performance bottlenecks from runahead techniques is known as *precise runahead execution (PRE)*. Unlike prior runahead techniques, precise runahead does not flush the instruction window, achieves better prefetching coverage by only executing useful instructions, and improves performance even when the instruction window is stalled for short intervals. These enhancements — aided by efficient recy-

cling of back-end resources — improve both performance and energy efficiency of runahead techniques.

In addition to substantially improving performance, runahead execution also improves reliability in out-of-order processors. Upon a full instruction window, all instructions starting from the long-latency load that blocked the instruction window are squashed from the pipeline and fetched again. The squashed instructions improve performance by generating memory prefetches, but the processor bits exposed by these instructions do not add toward the vulnerability of the application. Therefore, runahead execution and its follow-up techniques improve both performance and reliability. In Chapter 4, we thoroughly analyzed the problem of high soft error vulnerability posed by long-latency loads, and proposed dispatch halting to address the problem. Section 4.6.7 compared vulnerability reduction achieved by dispatch halting and runahead execution. This chapter extends our analysis of soft error vulnerability to runahead buffer [79] and precise runahead execution. Runahead buffer is a technique proposed as an optimization on runahead execution while precise runahead execution is a novel technique proposed in this dissertation.

This chapter is organized as follows. Section 5.1 provides background on prior runahead techniques. Section 5.2 lists the shortcomings of runahead techniques and demonstrates the potential for improving their performance. Section 5.3 provides insights underpinning PRE and describes in detail the working of the microarchitecture for PRE. The experimental setup and workloads are explained in Section 5.4. Section 5.5 compares the performance of PRE to prior runahead techniques. Section 5.6 analyses the reliability improvement incurred by all runahead techniques including PRE. Section 5.7 refreshes more related work in the areas of runahead execution, prefetching, pre-execution, and we summarize in Section 5.8.

5.1 Background

In this section, we describe the original runahead proposal and the optimizations introduced in follow-on work.

5.1.1 Full-Window Stalls

In an out-of-order core, a load instruction that misses in the last-level cache (LLC) typically takes a couple hundred cycles to bring in data from off-chip memory to the processor. Soon, the load instruction blocks commit and the core cannot make any progress. Meanwhile, the front-end continues to dispatch new instructions into the back-end. Once the ROB[1] fills up, the front-end can no longer dispatch instructions, leading to a *full-window stall*. Figure 5.1 shows that an out-of-order processor executing a set of memory-intensive SPEC CPU benchmarks spends about half of its execution time waiting for long-latency

[1]ROB and (instruction) window are used interchangeably.

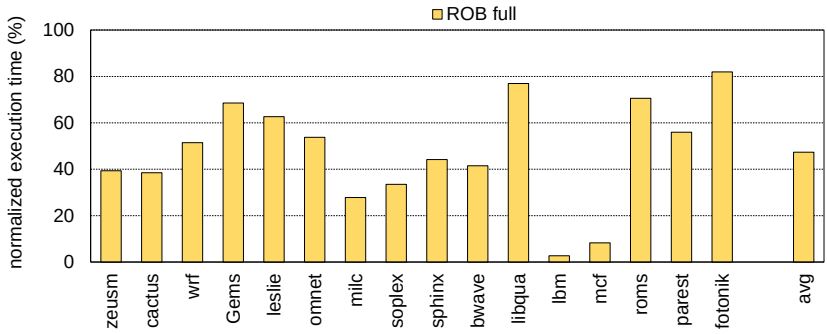

Figure 5.1: The fraction of execution time the ROB is full for memory-intensive benchmarks. *An out-of-order processor stalls on a full ROB for about half the time.*

loads blocking the ROB (see Section 5.4 for details about our experimental setup). We refer to the load instruction that causes a full-window stall as a *stalling load*, and to the backward chain of instructions that leads to a stalling load as a *stalling slice*.

5.1.2 Runahead Execution

Runahead execution [139] pre-executes an application's own code to prefetch data into the on-chip caches. Upon a full-window stall, the processor checkpoints the Program Counter (PC), Architectural Register File (ARF), the branch history register, and the return address stack. The processor enters runahead mode and marks the stalling load and its dependents as invalid. The processor pseudo-retires instructions without updating the processor architectural state to keep the execution moving forward speculatively. Once the stalling load returns, the pipeline is flushed and the checkpointed architecture state is restored. This marks the exit from runahead mode.

Runahead execution incurs a significant performance and energy overhead by flushing and refilling the pipeline when returning to normal execution mode. Mutlu et al. [141] propose enhancements to the original runahead proposal to alleviate the impact of this high overhead. Mainly, they propose invoking runahead execution only when the runahead interval is long enough to achieve high performance benefits that overshadow the overheads of runahead execution. In particular, they propose a policy to invoke runahead execution only if the stalling load was issued to memory less than a threshold number of cycles ago. They also propose another enhancement that prevents triggering runahead execution if it overlaps with an earlier runahead interval.

5.1.3 Future Thread

Future thread [15] shares the same purpose as runahead execution, while relying on two hardware threads, each with a dynamically allocated number

of physical registers. When the main thread exhausts its allocated physical registers due to a long-latency load, it stalls and the processor switches to a second hardware context (i.e., the future thread) in an attempt to prefetch future stalling loads. This technique requires hardware support for two hardware contexts. Further, it exposes less MLP than runahead because the future thread needs to share resources with the main thread, which limits how far the future thread can speculate.

5.1.4 Filtered Runahead Execution

Both the original runahead and the future-thread techniques execute all instructions coming from the processor front-end. However, many instructions are not necessary to calculate the memory addresses used in subsequent long-latency loads. Hashemi et al. [79] propose a technique to track and execute only the chain of instructions that leads to a long-latency load. Upon a full-window stall, they perform a backward data-flow walk in the ROB and store queue to find a dependency chain that leads to another instance of the same stalling load. This chain is stored in a buffer called the *runahead buffer* that is placed before the rename stage. In runahead mode, the instruction chain stored in the runahead buffer is renamed, dispatched and executed in a loop, instead of generating new instructions via the front-end. Therefore, the front-end can be clock-gated to save dynamic power consumption in runahead mode. By executing only the stalling slice, this technique exposes more MLP per runahead interval than traditional runahead.

5.2 Shortcomings of Prior Techniques

Both traditional runahead execution and runahead buffer significantly improve single-threaded performance. However, their full potential is limited by the following key factors.

Flushing and Refilling the Pipeline. Runahead execution speculatively executes and pseudo-retires instructions. At the exit of runahead execution, the processor flushes the pipeline and starts fetching instructions from the stalling load. Performing this operation for every runahead invocation incurs significant performance and energy overheads. Assuming that the ARF can be saved/restored in zero cycles, we estimate that every runahead invocation incurs a performance penalty of approximately 56 cycles assuming a 192-entry ROB: (1) refilling the front-end (8 cycles, assuming an 8-stage front-end pipeline), plus (2) refilling the ROB by re-dispatching 192 instructions with a dispatch width of 4, starting from the stalling load (48 cycles). These cycles cannot be hidden and thus directly contribute to the total execution time. Our experimental results reveal that compared to an out-of-order core, traditional runahead execution improves performance by 16% on average. However, if the instructions that occupy the ROB when the core enters runahead mode would

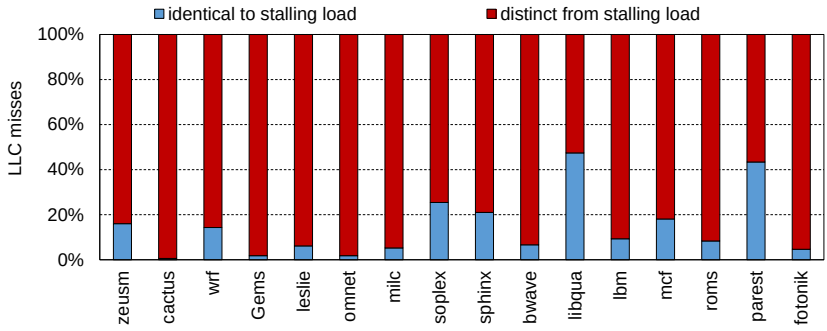

Figure 5.2: Percentage of long-latency load misses during runhead that are identical to, versus distinct from, the stalling load. *Most of the long-latency loads during runahead mode differ from the stalling load.*

not need to be re-fetched and re-processed after exiting runahead mode, the speedup has the potential to reach 22.8%.

Limited Prefetch Coverage. Traditional runahead execution has limited prefetch coverage because it executes all future instructions in runahead mode, which limits how deep in the dynamic instruction stream runahead execution can speculate. Runahead buffer filters and executes only the most dominant stalling slice per runahead interval. Runahead buffer assumes that the load that triggers runahead execution is likely to recur more than any other load within the same runahead interval. Therefore, it decides to replay only the chain of instructions that produces future instances of the same stalling load. Although runahead buffer enables runahead execution to speculate further down the instruction stream, it is limited to a single slice. Unfortunately, this does not match the characteristics of applications that access memory through a diverse set of instruction slices and multiple different load instructions.

Figure 5.2 classifies the long-latency loads (i.e., loads that miss in the last-level cache) that are encountered in a runahead interval into either identical to, or distinct from, the stalling load that initiated the runahead interval. The figure shows that most of the long-latency loads that are encountered in a runahead interval differ from the stalling load that triggered runahead execution. Relying on a single dominant stalling load per interval, as in runahead buffer, therefore neglects major prefetching opportunities. (Note further that miss-dependent misses of the same unique load that appear in the dependence chain determined by runahead buffer, cannot be prefetched — miss-dependent misses require a prediction mechanism such as address-value delta [142] or require migrating the dependency chain to the memory controller [80].)

In general, we find that memory-intensive applications access off-chip memory through multiple load slices. Figure 5.3 categorizes all runahead intervals according to the number of unique long-latency loads each interval contains. Most of the runahead intervals feature off-chip memory accesses via multiple unique load instructions.

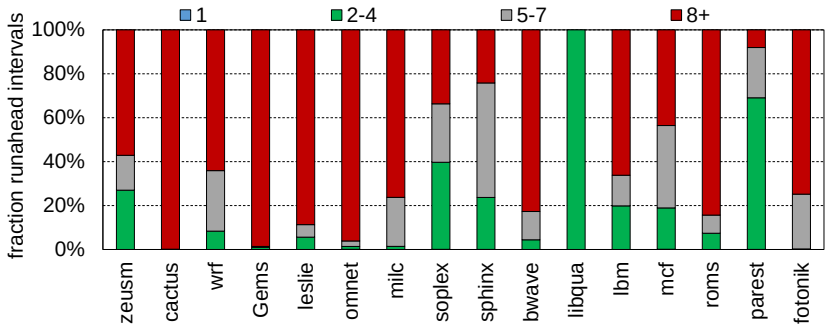

Figure 5.3: Runahead intervals categorized by the number of unique long-latency loads. *Most runahead intervals feature multiple unique long-latency load instructions.*

Short Runahead Intervals. The proposed enhancements to runahead execution prevent initiating runahead mode if the runahead interval is estimated to be short. For such cases, the overhead of invoking runahead execution outweighs its benefit [141]. However, a significant fraction of runahead intervals are short. We find that, on average for the memory-intensive benchmarks considered in this work, less than 56 cycles remain before the stalling load returns by the time the window is filled up for 40% of the runahead intervals — 56 cycles is the overhead for refilling the pipeline after a runahead interval as previously determined. Excluding short runahead intervals limits how often runahead is triggered, which wastes significant opportunity to enhance MLP.

5.3 Precise Runahead Execution

In this work, we propose *Precise Runahead Execution (PRE)* to alleviate the limitations of prior runahead proposals. PRE improves prefetch coverage over prior proposals by prefetching all stalling slices in runahead mode, unlike runahead buffer, and executing only the instruction chains leading to the loads, unlike the original runahead proposal. Moreover, PRE does not release processor state when entering runahead mode, hence it does not need to flush and refill the pipeline when resuming normal mode. This reduces the cost for invoking runahead execution.

We first describe the key insights that inspire the design of PRE, after which we describe PRE's architecture and operation in detail.

5.3.1 PRE: Key Insights

PRE builds on three key insights.

Insight #1: There are enough available physical register file (PRF) and issue queue (IQ) resources to initiate runahead execution upon a full-window stall. To execute an instruction, the processor minimally needs a physical register

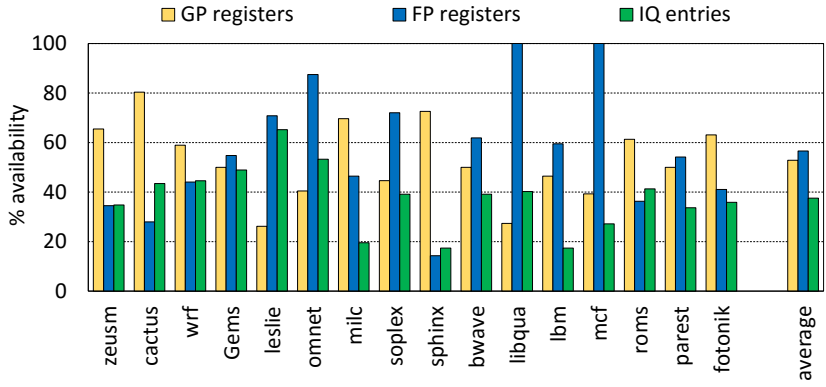

Figure 5.4: Percentage general-purpose (GP) registers, floating-point (FP) registers and issue queue (IQ) entries that are available upon a full-window stall due to a long-latency load blocking commit. *About half the issue queue and physical register file entries are available upon a full-window stall.*

to hold the instruction's destination value plus an issue queue entry for the instruction to wait until an execution unit becomes available. Figure 5.4 shows the percentage of available (i.e., unused) processor issue queue and physical register file entries at the entry of runahead mode. On average, 37% of the issue queue entries, 51% of the integer registers and 59% of the floating-point registers are free. Obviously, we can exploit these resources by increasing the ROB size. Therefore, we perform an experiment where we keep increasing the ROB size by 64 while keeping other core parameters fixed. Relative to the baseline OoO core (ROB=192), the performance improvements for ROBs of size 256, 320, 384, 448, 512, and 576, are 9.2%, 12.8%, 14.4%, 15%, 15.1%, and 15.2%, respectively, and the performance does not improve further for a larger ROB.

Although we increase the ROB size by 3×, nevertheless, the performance does not scale accordingly. This is because the majority of instructions in the ROB occupy resources without significantly boosting the performance of memory-intensive benchmarks. Therefore, the availability of resources — that is, the unused of 37% of the issue queue entries, 51% of the integer registers and 59% of the floating-point registers — at the full-window stalls is not an artifact of the unbalanced processor design. In fact, Section 5.4 provides quantitative evidence that our baseline configuration is indeed a balanced design. We thus conclude that there are enough issue queue entries and registers upon a full-window stall to initiate the speculative execution of instructions that lead to anticipated future long-latency load misses.

Insight #2: There is no need to pre-execute all instructions during runahead mode. Instead, we can speculate deeper in the dynamic instruction stream by only pre-executing stalling load slices. The majority of instructions executed during runahead execution occupy core resources (e.g., PRF, IQ, ALU) without actually contributing to generating useful prefetches. Ideally, we only need to speculatively execute instructions that lead to future long-latency load stalls,

i.e., we need to execute the producers of the long-latency loads and not their consumers. This not only reduces the core resources needed during runahead execution, it also allows for speculating deeper down the dynamic instruction stream and extract more useful prefetches. PRE achieves this by identifying and speculatively executing *stalling load slices*, i.e., backward slices of long-latency loads that lead to full-ROB stalls.

Insight #3: IQ resources are quickly recycled during runahead execution. Recycling PRF resources requires a novel mechanism that is different from conventional register renaming schemes. Stalling load slices are relatively short chains of dependent instructions. These chains of load-producing instructions occupy IQ resources for only a short time, i.e., instructions wait for their input operands for a few cycles and then execute. In contrast, the load consumers hold on to IQ resources as they wait for the load values to return from memory. In other words, PRE is able to quickly recycle IQ resources by only executing stalling load slices during runahead mode. The situation is different for the physical register file: stalling load slices hold up PRF resources if they are released using conventional register renaming. PRE therefore includes a novel register reclamation mechanism to quickly recycle physical registers in runahead mode.

Figure 5.5 depicts a schematic diagram of an out-of-order core supporting PRE. The following subsections describe its operation in detail.

5.3.2 Entering Precise Runahead Execution

As in prior techniques, PRE is invoked on a full-window stall. PRE enters runahead mode after checkpointing the Program Counter (PC) of the instruction past the full-ROB, the Register Alias Table (RAT), and the return address stack (RAS). The instructions filling the ROB can still execute as they do in normal mode. However, no instructions are committed from the ROB in runahead mode. Therefore, no updates are propagated to the ARF and the L1 D-cache. During runahead execution, PRE dynamically identifies the instructions that are part of potential stalling slices as they arrive from the decode unit (as described in the next section), and the core speculatively executes them.

5.3.3 Identifying Stalling Slices

PRE tracks the individual instructions that form a stalling slice in a new cache that we call the *Stalling Slice Table (SST)*. As Figure 5.5 shows, the SST is accessed after the decode stage. The SST is a fully-associative cache that contains only instruction addresses (i.e., PCs). If an instruction address hits in the SST, that instruction is part of a stalling slice. Whenever a stalling load blocks the ROB, we store it in the SST. To facilitate tracking the chain of instructions that leads to that load, we extend each entry in the RAT to hold the PC of the instruction that last produced that register. When the register renaming unit maps the destination architectural register of an instruction to

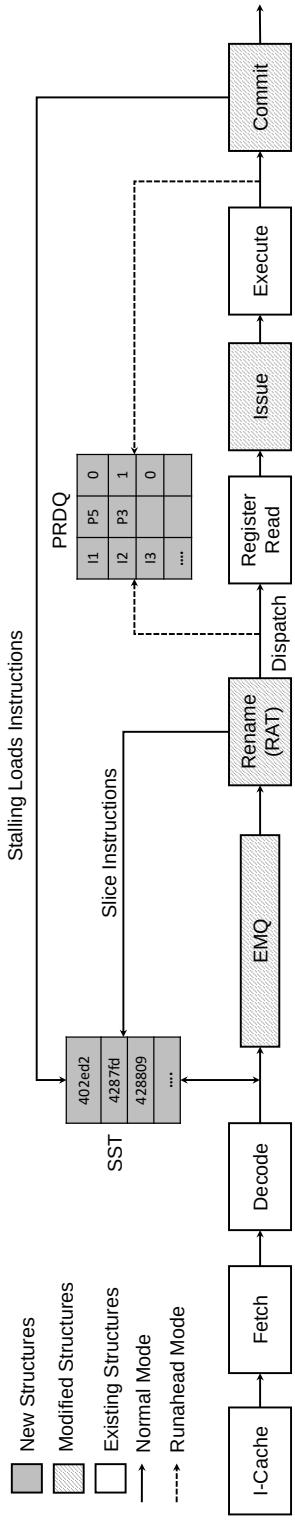

Figure 5.5: Core microarchitecture to support precise runahead execution.

a new physical register, it also updates the RAT entry corresponding to that architectural register with the PC of the instruction.

We track the stalling slices in an iterative manner. First, the stalling load is stored in the SST. When the stalling load is decoded again, e.g., in the next iteration of a loop, the PC of the stalling load hits in the SST. PRE checks the RAT entry for the load's source registers to find the PCs of the instructions that last produced those registers; these PCs are then stored in the SST. Similarly, whenever an instruction hits in the SST in the following iterations, we track the PC information of its producer instructions and add those to the SST as well. This iterative process effectively builds up the stalling slice in the SST. PRE follows this same process for all stalling loads. By tracking all stalling slices in the SST, PRE does not limit prefetch coverage to a single slice as in the runahead buffer proposal.

Branch instructions are not part of a stalling load slice because they are not involved in the load address calculation. Therefore, branch instructions are not stored in the SST. A branch instruction can modify the stalling slice by changing the producer of one instruction in the slice, potentially forming two slices that lead to the same load instruction. PRE simply identifies the new producers and adds them to the SST. In the following iterations, PRE builds the whole slice in SST similar to any other slice. In the end, SST tracks all slices that lead to stalling loads.

We find that an SST of limited size is effective at capturing stalling slices to generate useful prefetches in a runahead interval. As the application progresses to a new loop, new stalling slices are identified and stored in the SST. With an LRU replacement policy, old and unused stalling slices are automatically evicted from the SST. It may happen that a slice is not complete in the SST, e.g., while being constructed, however, the slice will soon be completed in the next few iterations. We find that an SST with 128 entries is sufficient to gain the majority of the performance benefits of runahead execution (see Section 5.3.8).

In Section 4.3.4, we exploit PIT for speculatively executing instructions when the dispatch is halted. The speculative execution of instructions under proactive dispatch halting helps in generating MLP and resolving mispredicted branches after a halting load instruction. The working of SST is similar to the PIT as both PIT and SST augment RAT for iteratively finding the past instructions that produce value of the source register of an instruction. However, the types of instructions stored in the PIT and SST are different. The PIT stores the backward slices of all load instructions and branch instructions; the load and branch instructions, however, are not stored in the PIT. SST, in contrast, stores stalling load instructions and their backward slices; branch instructions and their backward slices are not stored in the SST.

5.3.4 Execution in Runahead Mode

PRE filters and speculatively executes all stalling slices that follow the stalled window using the SST. After instruction decode, PRE executes only

Register renaming and its outcome

inst. id	instruction	dst	src1	src2	register to free
I1	add r1 ← r2, r3	P1	P2	P3	
I2	mul r2 ← r1, r4	P5	P1	P4	P2
I3	ld r1 ← mem[x]	P6			P1
I4	add r2 ← r1, r3	P7	P6	P3	P5
I5	add r2 ← r4, r5	P9	P4	P8	P7
I6	sub r1 ← r2, r6	P11	P9	P10	P6

PRDQ

inst. id	register to free	executed ?
I1		1
I2	P2	1
I3	P1	0
I4	P5	0
I5	P7	1
I6	P6	0

Figure 5.6: Recycling physical registers during precise runahead execution using the PRDQ.

the instructions that hit in the SST because they are necessary to generate future loads. PRE achieves the benefits of filtered runahead execution as with runahead buffer because it executes only the stalling slices. However, because the SST stores all stalling slices, PRE manages to executes *all* potential stalling slices, which leads to much improved prefetch coverage.

Instructions issued in runahead mode use only the free registers that are unused when runahead mode is triggered. These registers are allocated and recycled in runahead mode without affecting the physical registers allocated in normal execution. PRE properly maintains dependences among the executed instructions and manages the allocation and reclamation of registers in runahead mode as described in Section 5.3.5. At the same time, the processor continues executing the non-speculative instructions that already occupy the ROB. The results are written to the physical destination registers that were allocated before triggering runahead execution. When the processor resumes normal operation, it restores the architectural state it checkpointed upon runahead entry. Only instructions that were fetched in runahead mode need to be fetched and processed again. The physical registers that were free prior to runahead execution are reclaimed. The physical registers that hold values written by instructions in the ROB in runahead mode can properly update the architectural state and get reclaimed when their respective instructions retire in normal mode.

In runahead mode, PRE executes all the slices generated by the front-end of the processor. The front-end relies on the branch predictor to steer the flow of execution in runahead mode. PRE does not update the state and history of the branch predictor during runahead execution. However, branch instructions that reside in the ROB can be resolved in runahead mode and update the predictor as they would in normal mode. If a branch instruction in the ROB turns out to be mispredicted, the processor discards all wrong path instructions, flushes the pipeline, and resumes normal execution.

5.3.5 Runahead Register Reclamation

PRE requires sufficient issue queue entries and physical registers to run ahead. As reported in Section 5.3.1, such resources are usually available when

entering runahead mode. Stalling slices are usually short and therefore issue queue entries are quickly reclaimed and are unlikely to hinder forward progress of runahead execution. In all of our experiments, we did not observe issue queue pressure during runahead.

PRE requires special support for reclaiming physical registers during runahead execution. In an out-of-order core, a physical register can be freed only when the last consumer of the renamed architectural register commits [200]. Since instructions that are fetched in runahead mode are discarded after they finish execution, we cannot rely on the conventional renaming policy to free physical registers. Thus, we devise a new mechanism, called *Runahead Register Reclamation (RRR)*, to free physical registers in runahead mode. RRR relies on a new FIFO hardware structure, called the *Precise Register Deallocation Queue (PRDQ)* in Figure 5.5.

Figure 5.6 illustrates the PRDQ in more detail. Each entry in the PRDQ has three fields: an instruction identifier, a physical register (tag) to be freed, and an 'execute' bit that marks whether the instruction has completed execution. The figure also provides a code example to help explain the operation of the PRDQ. The instructions in the example are numbered following program order. For example, instruction `I2` precedes instruction `I4` in program order. The figure shows the instructions after the register renaming stage. In this code example, instruction `I4` reads the value of architectural register `r1` from physical register `P6`, which is written by instruction `I3`. `I4` also reads the value of architectural register `r3` from physical register `P3` written by an older instruction not shown in the code example.

PRDQ entries are allocated in program order at the PRDQ tail. Register renaming maps a free physical register to the destination architectural register of an instruction in runahead mode. We mark the old physical register mapped to the same (destination) architectural register in the PRDQ entry. A PRDQ entry is deallocated when the instruction is executed (i.e., 'execute' bit is set) and reaches the PRDQ head. PRDQ deallocation is also done in program order. The old physical register associated with the instruction is freed upon deallocation. For example, in Figure 5.6, the renaming unit maps the destination architectural register of instruction `I4` (i.e., `r2`) to physical register `P7` and marks old physical register mapped to `r2` (i.e., `P5`) to be freed when `I4` is retired and deallocated from the PRDQ.

While instructions may execute and thus mark the 'execute' bit out-of-order, in-order PRDQ deallocation guarantees that a physical register is freed only when there are no more instructions in-flight that may possibly read that register. The PRDQ is only enabled in runahead mode and its entries are discarded once the processor returns to normal mode.

5.3.6 Exiting Precise Runahead Execution

The core exits runahead mode when the stalling load returns. Upon exit, the core resumes normal execution after having restored the checkpointed PC,

RAT, and RAS. As instructions are preserved in the ROB, the core starts committing instructions right away starting from the stalling load. The front-end re-directs fetch from the first instruction after the full-window stall, i.e., the PC which was checkpointed when entering runahead mode.

5.3.7 Front-End Optimization

PRE executes future stalling slices for the entire length of a runahead interval. During this time, PRE requires the front-end of the processor to keep fetching and decoding instructions to dynamically explore the code. Therefore, the front-end has to remain active during runahead mode. The instructions fetched in runahead mode are fetched and processed again for execution in normal mode. This increases the energy overhead in the front-end of the processor for PRE compared to runahead buffer [79].

To avoid wasting the work and energy of the front-end in runahead mode, we propose the Extended Micro-Op Queue (EMQ) as shown in Figure 5.5. Superscalar out-of-order processors typically feature a dedicated micro-op queue to hold micro-ops after instruction decode. For example, Intel Skylake uses a micro-op queues of 64 entries [91]. We propose extending the number of entries of the processor's micro-op queue, hence the name (EMQ). The micro-op queue is a circular FIFO buffer and thus can be extended without impacting the complexity of the design. We augment PRE with an EMQ to store the micro-ops generated in runahead mode.

When using the EMQ, PRE stores all the decoded instructions in runahead mode, including the ones that hit in the SST. When the processor resumes normal execution, it does not need to re-fetch and re-decode all these instructions again. These instructions are directly renamed, dispatched and executed in the back-end. Note that with this optimization, the number of speculatively executed instructions in runahead mode is constrained by the size of the EMQ. When the EMQ fills up, the core stalls until the stalling load returns, at which point, the processor exits runahead mode. Alternatively, the processor can continue fetching instructions beyond the size of EMQ for the whole runahead interval. In this case, the processor only needs to re-fetch the instructions that could not be buffered in the EMQ during runahead execution. This design alternative, however, is similar to PRE's original design and does not lead to significant variation in its energy and performance profile.

Using EMQ is an optional design optimization. It is not mandatory for PRE's runahead operation. As we show in Section 5.5.3, augmenting PRE with EMQ of various sizes leads to different design points that trade off performance for energy. Designers can select a suitable design choice based on the available area and energy budgets.

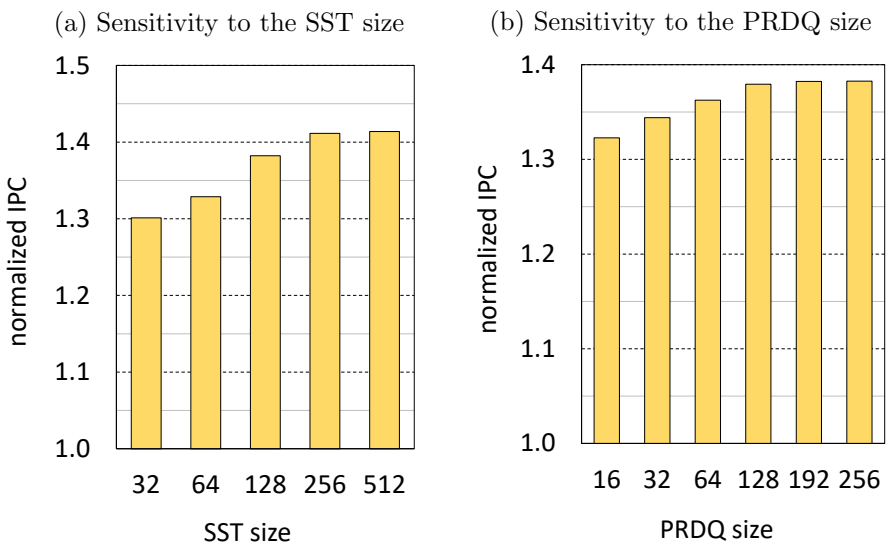

Figure 5.7: Performance impact of changing the size of the SST and PRDQ. Performance is normalized to the OoO core. *An SST size of 128 entries balances performance and hardware cost; performance saturates for PRDQ size of 192 entries.*

5.3.8 Hardware Overhead

As mentioned before, PRE relies on the newly proposed SST and PRDQ. We conduct a sensitivity analysis to empirically select their sizes. Figure 5.7 reports the impact of varying SST and PRDQ sizes on performance (normalized to baseline OoO core). To balance hardware cost and performance, we opt for an SST with 128 entries; increasing the SST size beyond 128 entries leads to a minor gain in performance while incurring a significant hardware cost. We set PRDQ size to 192 entries because it achieves the best performance and its hardware cost is small.

An SST with 128 entries each with a 4-byte tag requires 512 Bytes of storage. An entry in the PRDQ consists of a single bit to indicate that the instruction finished execution, an 8-bit tag for the physical register to free, and 12 bits (assuming a maximum of 4096 runahead instructions) to give each instruction explored in runahead mode a unique ID. This adds up for a total of 504 Bytes. Additionally, we extend each mapping of the 64-entry RAT by 4 bytes for a total of 256 Bytes. This leads to a total hardware cost of 1.24 KB. When PRE is augmented with an (optional) EMQ, the hardware overhead is increased according to the selected EMQ size, with each EMQ entry requiring 4 Bytes to hold a micro-op. In comparison, runahead buffer incurs a hardware cost of about 1.7 KB and uses expensive CAM lookups in the ROB to determine stalling slices. Overall, the hardware cost and complexity of PRE is smaller compared to the runahead buffer proposal.

Frequency	2.66 GHz
Type	out-of-order
ROB size	192
Issue queue size	92
Load queue size	64
Store queue size	64
Micro-op queue size	28
Pipeline width	4
Pipeline depth	8 stages (front-end only)
Branch predictor	8 KB TAGE-SC-L
Functional units	3 int add (1 cyc), 1 int mult (3 cyc), 1 int div (18 cyc), 1 fp add (3 cyc), 1 fp mult (5 cyc), 1 fp div (6 cyc)
Register file	168 int (64 bit) 168 fp (128 bit)
SST size	128 entry, fully assoc, LRU, 6r 4w
PRDQ size	192 entry, 4r 4w
L1 I-cache	32 KB, assoc 4, 2 cyc
L1 D-cache	32 KB, assoc 8, 4 cyc
Private L2 cache	256 KB, assoc 8, 8 cyc
Shared L3 cache	1 MB, assoc 16, lat 30 cyc
Memory	DDR3-1600, 800 MHz ranks: 4, banks: 32 page size: 4 KB, bus: 64 bits `tRP-tCL-tRCD`: 11-11-11

Table 5.1: Baseline configuration for the out-of-order core.

5.4 Methodology

Simulation Setup. We evaluate precise runahead execution using the cycle-level, hardware-validated Sniper 6.0 [32] simulator, using its most accurate core model. The configuration for our baseline out-of-order core is provided in Table 5.1. The sizes of the ROB, the physical register files, and the micro-op queue are based on the Haswell architecture [62, 76]; the size of the issue queue is set as in the runahead buffer paper [79] for fair comparison. We verify that this baseline configuration is indeed balanced, see Figure 5.8, i.e., the physical register file (PRF) and issue queue (IQ) sizes are the minimum sizes that lead to the best performance for the given ROB size. We assume that hardware prefetching is not enabled in our baseline core. However, we do evaluate the impact of hardware prefetching in Section 5.5.4. We consider an 8 KB TAGE-SC-L branch predictor as implemented for the 2016 Branch Prediction Championship [177].

Power. We use McPAT [118] to calculate power consumption assuming a 22 nm chip technology. We calculate power for the SST, EMQ and PRDQ using CACTI 6.5 [119] and add those estimates to the McPAT power numbers. We report system power (processor plus main memory).

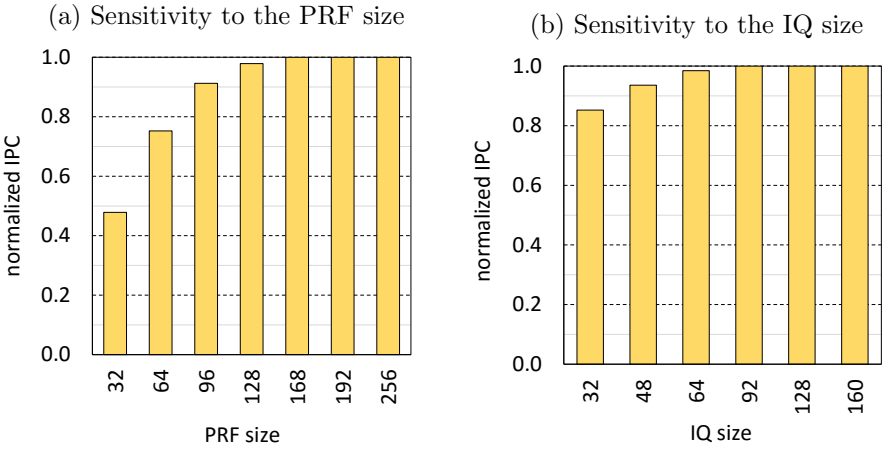

Figure 5.8: Impact of PRF and IQ sizes on performance while keeping the other configuration parameters constant. *Overall, the baseline OoO with 168 PRF entries and 92 IQ entries is a balanced configuration.*

Cycle time. We model the impact of the newly added hardware structures on processor cycle time using CACTI 6.5 [119]. We assume that the front-end can deliver up to six micro-ops per cycle to the micro-op queue. Therefore, SST has 6/2 read/write ports. In the runahead mode, we can check up to six micro-ops per cycle in the SST. PRDQ is an in-order queue with 4/4 read/write ports. The cycle time for accessing the SST and PRDQ equals 0.314 ns and 0.102 ns, respectively. Since this is below the processor cycle time (0.375 ns), we conclude that the accesses to the SST and PRDQ do not impact processor timing. (The SST can be pipelined, if needed, since it is not on the critical path.)

Workloads. We evaluate a total of 16 memory-intensive benchmarks from the SPEC CPU2006 and SPEC CPU2017 suites. From the CPU2006 suite, we select the same benchmarks as runahead buffer [79], and we maintain the same order when presenting our results. Compared to SPEC CPU2006, there are fewer memory-intensive benchmarks in the CPU2017 suite and, even though some benchmarks (e.g., `bwaves`) have multiple input data sets, their fraction of full-window stalls is similar in our setup. The three new memory-intensive benchmarks we have included from the SPEC CPU2017 suite are `roms_r_1`, `parest_r_1` and `fotonik3d_r_1`. We create 1 Billion instruction SimPoints [180] for each benchmark.

5.5 Evaluation

We compare the following four mechanisms:

- OoO: Our baseline out-of-order core from Table 5.1.

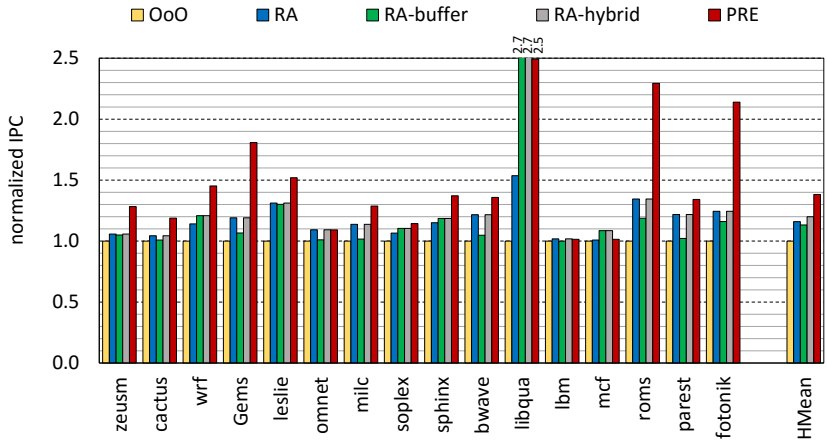

Figure 5.9: Performance (IPC) normalized to an out-of-order core for runahead execution, runahead buffer and precise runahead execution. *PRE improves performance by 38% on average compared to the baseline out-of-order core.*

- RA: The runahead execution, as explained in Section 5.1.2, with the following enhancements [79, 141]:
  - There are no overlapping runahead intervals.
  - Runahead execution is triggered only when the stalling load instruction was issued to memory less than 250 cycles earlier.
- RA-buffer: The runahead buffer mechanism explained in Section 5.1.4. In runahead mode, the front-end of the processor is clock-gated and the dominant stalling load slice for each runahead interval is executed from the runahead buffer. We assume all the chains are stored in a chain cache. Therefore, no extra overhead is required to perform backward walks in the ROB.
- RA-hybrid: The hybrid runahead approach selects the runahead technique (RA or RA-buffer) that yields the highest performance on a per-application basis.
- PRE: The precise runahead execution proposal as described in this paper.

We use *instructions per cycle (IPC)* to quantify performance. We calculate average performance across all benchmarks using the harmonic mean IPC across all benchmarks.

5.5.1 Performance

Figure 5.9 reports performance for the various runahead techniques, normalized to the baseline OoO core. While the RA and RA-buffer improve per-

formance over the OoO core by on average 16.0% and 13.3%, respectively, RA-hybrid which selects the best of both techniques improves performance by 20%. PRE on the other hand manages to improve performance by 38.2%. This is an additional 18.2% improvement over prior runahead techniques. Most of the applications gain a significant performance improvement with PRE. In general, we find that applications that spend more time waiting on a full-window stall have a higher chance to benefit from PRE. PRE achieves the highest performance improvements for `GemsFDTD`, `leslie3d`, `libquantum`, `roms` and `fotonik`. As Figure 5.1 shows, these applications spend more than 60% of their execution time on full-window stalls, providing PRE a significant opportunity to generate useful prefetches. The performance improvements for these applications range from 52% up to more than 2 ×, see `libquantum`, `roms` and `fotonik`. Other applications that spend less time waiting for long-latency loads like `zeusmp`, `wrf`, `milc`, `sphinx3`, `bwaves` and `parest` still achieve a significant performance improvement that ranges between 20% and 40%.

The significant performance improvement of PRE relative to prior runahead techniques comes from its higher prefetch coverage and the fact that it avoids flushing and re-filling the pipeline when leaving runahead mode. However, we find a few outlier cases where PRE has only a minor benefit compared to either the OoO or to prior runahead techniques. We observe that none of the runahead techniques significantly improve performance of the OoO core for `lbm`. This benchmark experiences full-window stalls for only 2.7% of the total execution time because the pipeline stalls on other resources. Therefore, the opportunity to prefetch in runahead mode is quite small. On the other hand, `omnetpp` is characterized by long stalling slices, as corroborated by [79]. The long stall slices limit PRE's opportunity to explore multiple slices per runahead interval. Therefore, PRE performs similarly to prior runahead execution for `omnetpp`.

The only benchmarks that benefit from RA-buffer more than PRE are `libquantum` and `mcf`. For `libquantum`, about 50% of the load instructions that access memory in a runahead interval are identical to the stalling load as Figure 5.2 shows. The rate at which RA-buffer executes the same stalling slice to generate prefetches exceeds that of PRE, which has to dynamically determine the slices. The benefits of the faster prefetch generation in a limited runahead interval for `libquantum` outweigh the benefits of finding all slices. On the other hand, `mcf` is characterized by its high branch misprediction rate. This means that both PRE and prior runahead techniques invoke useless runahead intervals that execute wrong-path instructions, and thus do not improve performance. Branch instructions that wait for the stalling load to be resolved benefit from RA-buffer because it prefetches only stalling load slices. RA-buffer is particularly beneficial for load-dependent branches that are mispredicted. Therefore, it manages to slightly improve performance over PRE which dynamically explores all stall slices.

We now further analyze the sources of performance improvement for PRE over prior runahead techniques.

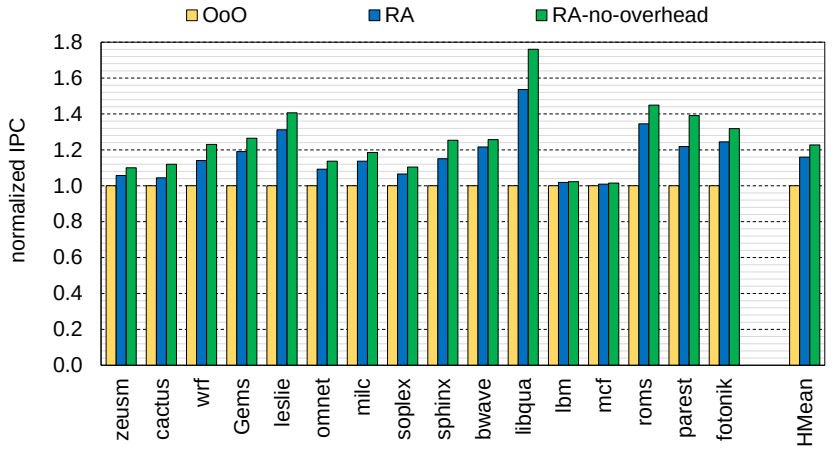

Figure 5.10: Performance impact of flushing the pipeline when leaving runahead mode and refilling it when resuming normal execution in RA. *PRE avoids this overhead as it does not need to flush the pipeline when leaving runahead mode.*

Pipeline Refill Overhead. PRE does not need to flush and refill the pipeline when resuming normal mode. This alone gives PRE a significant performance improvement over the original runahead proposal. Even with the enhancements introduced to the original runahead technique, the overhead of flushing the pipeline when leaving runahead mode and refilling it starting from the stalling load still limits its performance improvement. Figure 5.10 demonstrates the significant impact of flushing and re-filling the processor pipeline on RA's performance improvement. Every exit from the runahead mode is followed by a pipeline bubble of at least 56 cycles — 8 cycles to re-fill the front-end and 48 cycles to re-dispatch the same instructions to the ROB. As the figure shows, RA improves the performance of the OoO core by 16% on average. The performance improvement jumps to 22.8% when the flushing and refilling overhead is avoided.

MLP. PRE improves the degree of MLP that is exposed over prior proposals, for three reasons. First, PRE triggers runahead execution even for relatively short runahead intervals. This allows PRE to invoke runahead execution 1.8× more than RA and RA-buffer. Second, PRE executes only the stalling slices, which enables PRE to uncover long-latency loads at a higher rate than RA per runahead interval, and thus speculate deeper down the dynamic instruction stream. Third, PRE targets multiple stalling load slices during runahead execution in contrast to RA-buffer which speculatively executes only one stalling slice in a loop.

As Figure 5.11 shows, the MLP generated by RA, RA-buffer, RA-hybrid, and PRE is 1.5×, 1.3×, 1.6×, and 2× higher than for the OoO core. PRE improves MLP for most of the applications, except for the few outlier applications that were previously discussed. In general, the higher MLP of PRE reflects its

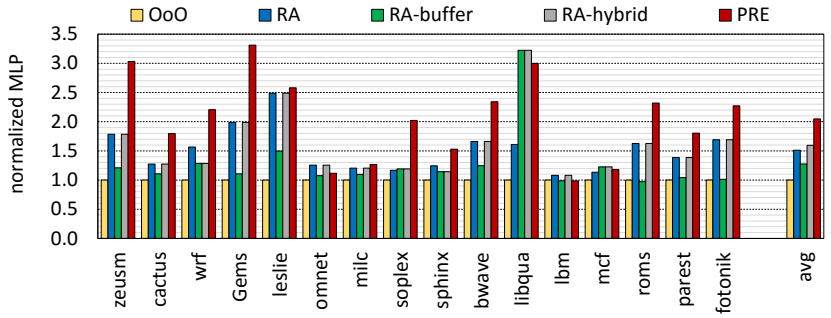

Figure 5.11: Normalized MLP. *PRE improves MLP by 2× compared to an out-of-order core.*

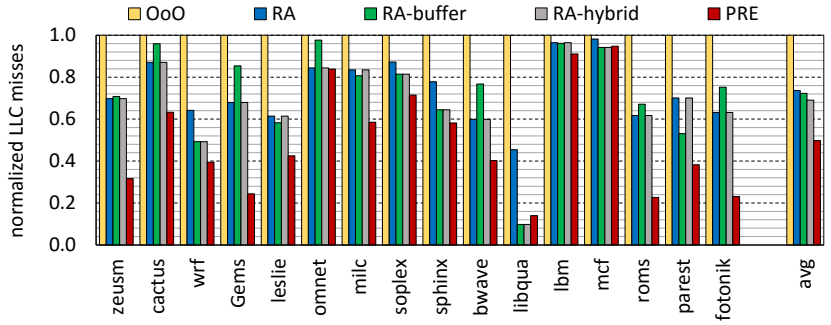

Figure 5.12: Normalized LLC miss count during normal (non-runahead) execution. *PRE's accurate prefetches reduce the number LLC misses by 50% compared to an OoO core.*

superior prefetch quality, which leads to higher overall performance. It is worth noting that although RA-buffer can generate about 2× more memory requests than RA per runahead interval as reported in [79], overall performance is not proportionally improved.

LLC Miss Rate. Figure 5.12 reports normalized LLC miss rate in normal mode for all the runahead techniques. All runahead techniques reduce the number of LLC misses observed during normal mode. However, we find that PRE covers more LLC misses than any other prior runahead technique. On average, RA, RA-buffer, and RA-hybrid reduce the number of LLC misses by 26.4%, 27.7% and 31%, respectively, whereas PRE reduces the number of LLC misses by 50.2%. This higher reduction in LLC miss rate is a result of covering more stalling slices deeper down the dynamic instruction stream.

5.5.2 Energy Analysis

Figure 5.13 shows the energy consumption for all runahead techniques normalized to the OoO core. RA increases energy consumption of an OoO core by 2.4% on average. RA-buffer clock-gates the front-end during runahead mode

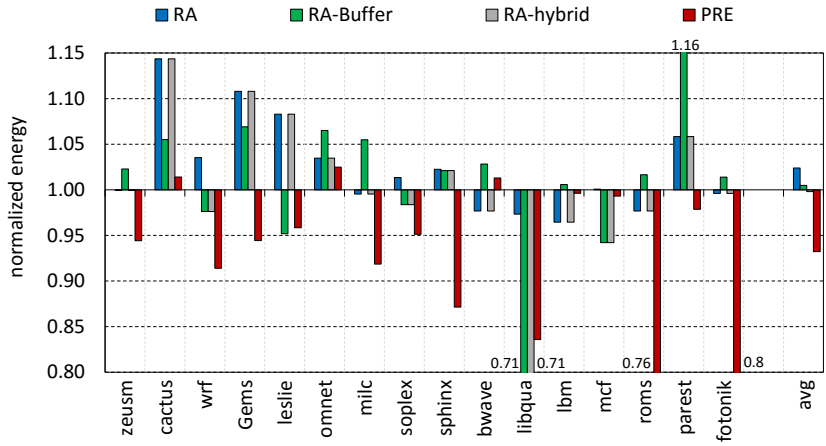

Figure 5.13: Normalized energy consumption. *PRE reduces energy consumption by 6.8% compared to an out-of-order core, while runahead execution slightly increases energy consumption or is energy-neutral.*

to reduce energy overhead to only 0.4% relative to the baseline OoO core. RA-hybrid slightly reduces the energy consumption compared to RA-buffer. In general, we find that the significant performance improvement of PRE allows it to complete the same task with less energy than the other techniques for most of the applications. Similar to our earlier discussion, only few outlier cases such as `libquantum` and `mcf` consume less energy using RA-buffer than with PRE. On average, PRE performs the same task with 6.8% less energy compared to the baseline OoO core.

5.5.3 Front-End Energy Optimization

PRE requires the front-end of the processor to remain active in runahead mode to find stalling slices further down the dynamic instruction stream. Upon resuming normal mode, the processor fetches and executes all the instructions that were *fetched* in runahead mode again. In Section 5.3.7, we proposed the EMQ as an optimization to save the energy consumed by the front-end in runahead mode. The EMQ is a design choice that trades off performance for energy.

Figure 5.14 shows the performance-energy trade-off for PRE with an EMQ of different sizes in multiples of the ROB size. (For example, an EMQ of size 2 × has 384 entries.) Without an EMQ, PRE keeps exploring the code throughout the entire runahead interval, leading to the highest performance improvement, however, this requires refetching instructions upon return to normal mode. With a limited EMQ, PRE can save the work of the front-end but may halt runahead execution before the end of the runahead interval. In contrast, larger EMQs enable PRE to explore more code than smaller ones, leading to higher performance and saving more work in the front-end. Thus, with a larger EMQ size, performance improves and energy consumption decreases. With a suffi-

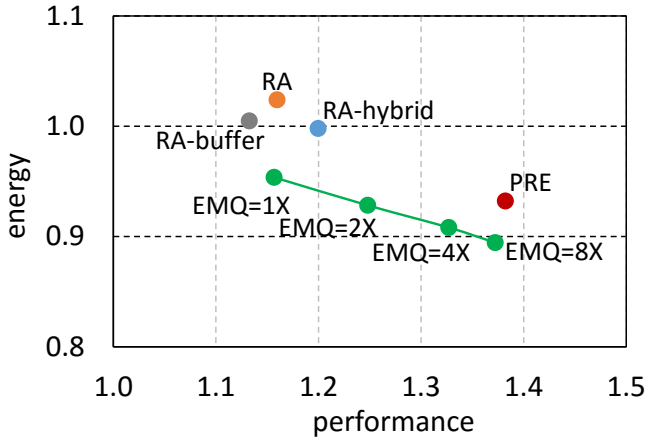

Figure 5.14: Performance versus energy normalized to the OoO core. *PRE improves performance and reduces energy consumption compared to an out-of-order core. Increasing the size of the (optional) EMQ further reduces energy consumption and presents an energy-performance trade-off.*

ciently large EMQ, it is possible to find design points that achieve comparable performance to PRE (without EMQ) while significantly saving energy, such as in the case for the EMQ=8× and EMQ=4× configurations. This comes at an increase in hardware cost though, e.g., an EMQ=4× storing 4 Bytes per entry requires 3 KB.

Interestingly, Figure 5.14 also shows that augmenting PRE with EMQ provides better performance-energy trade-off points than prior runahead techniques even with limited EMQ sizes. For example, for the EMQ=1× configuration, PRE yields higher performance than RA-buffer at a lower energy cost. Similarly, for the EMQ=2× configuration, PRE yields higher performance than all prior runahead techniques at a lower energy cost. Whether to use an EMQ or not, and which EMQ size to select, are design alternatives that can be selected at design time based on the available energy and area budgets.

5.5.4 Architecture Sensitivity

Hardware Prefetching. Hardware prefetchers and runahead techniques both aim at bringing data into the on-chip caches before it is needed by the workload. Generally speaking, hardware prefetchers exploit memory access patterns to predict which data to prefetch. On the other hand, runahead techniques generate prefetch requests by pre-executing the code. Both techniques are complementary to each other. If the hardware prefetchers are able to predict LLC misses and convert them into hits, runahead execution is not triggered. Conversely, when runahead techniques are effective at prefetching data, hardware prefetchers are invoked fewer times.

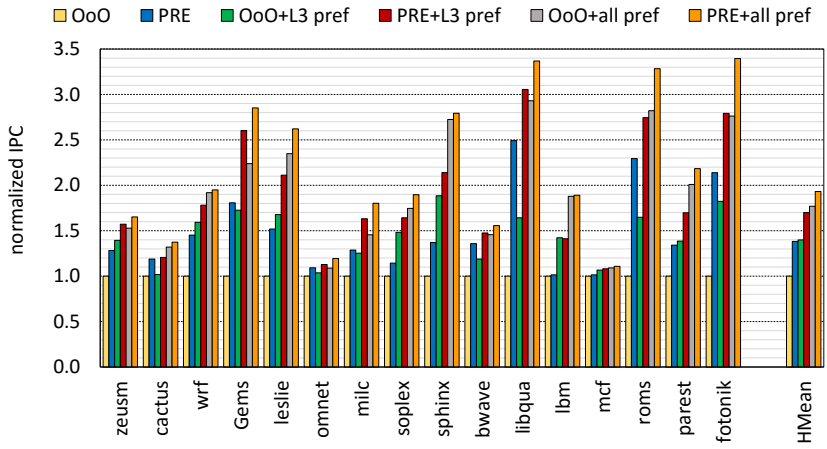

Figure 5.15: Performance relative to the baseline OoO core (without prefetching) when hardware prefetching is enabled at the LLC and all the cache levels. *PRE improves performance even when conventional stride prefetching is enabled at the LLC and all cache levels.*

Figure 5.15 shows the performance improvement of the baseline OoO core and PRE when augmented with hardware prefetchers. We evaluate two configurations: (i) a stride-based LLC hardware prefetcher with 16 streams, and (ii) a stride-based hardware prefetcher with 16 streams incorporated at all levels in the hierarchy. PRE leads to significant performance improvements even for processor configurations with conventional hardware prefetchers. For the configuration with the LLC prefetcher as a baseline, PRE improves performance by 21.5%. For the configuration with prefetchers engaged at all cache levels, PRE improves performance by 9.1%. The performance benefit obtained through PRE is expected to reduce with more aggressive hardware prefetching. Nevertheless, we conclude that PRE offers non-trivial performance improvements even under (aggressive) hardware prefetching.

Physical Register File. PRE leverages available PRF entries to speculate beyond a full ROB. Figure 5.16(a) quantifies PRE's average performance improvement as we scale the number of PRF entries (PRF=N means N integer and N floating-point registers). PRE performance is (obviously) sensitive to the number of physical registers. Small PRF sizes exhaust the number of available PRF entries, preventing PRE from speculating beyond the full ROB. Our baseline configuration assumes a balanced PRF size of 168 entries. Smaller PRF sizes, even if this leads to an unbalanced baseline design, would still experience a non-trivial improvement through PRE: 19.7% average performance improvement for a PRF size of 128 and 31.2% for a PRF size of 144.

Issue Queue. Similarly, PRE leverages available issue queue (IQ) sizes to speculate beyond a full ROB. Figure 5.16(b) reports the average performance improvement achieved through PRE as a function of IQ size. Small IQ sizes limit the number of resources that PRE can use during runahead mode, which

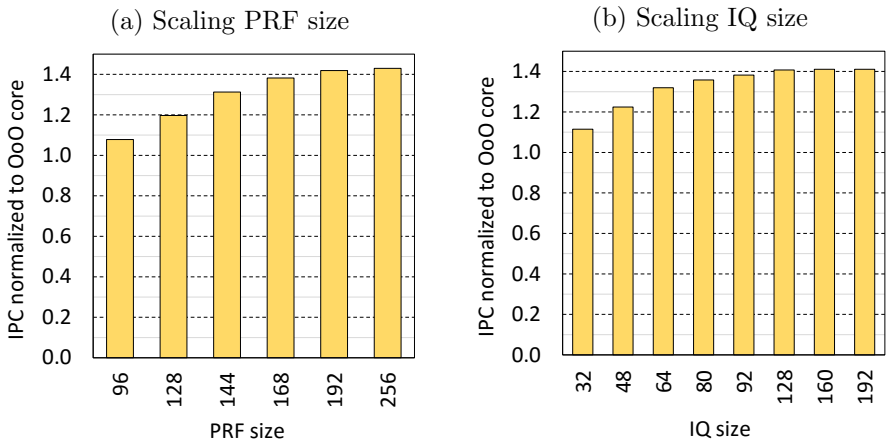

Figure 5.16: Performance improvement through PRE as a function of PRF and IQ size. *PRE improves performance even for PRF and IQ sizes that are underprovisioned.*

limits the performance improvement achieved by PRE. Our baseline assumes an IQ size of 92. Smaller IQ sizes still enable PRE to achieve substantial performance improvements: 31.9% for an IQ size of 64 and 35.8% for an IQ size of 80.

5.6 Impact on Reliability

RA and RA-buffer trigger runahead mode upon a full-window stall. All instructions executed in runahead mode are executed again when the core returns to the normal mode. Therefore, the bits exposed by instructions in runahead mode do not count toward the ACE bit count of the application. By consequence, RA and RA-buffer, while improving performance, also improve soft error reliability. The longer the duration of the core in runahead mode, the higher the improvement in reliability, assuming that the execution in runahead mode leads to some performance improvement. In PRE, not all instructions executed in runahead mode are speculative as the instructions within the ROB are always normal-mode instructions. These ROB instructions, and the back-end structures occupied by them, are thus vulnerable to soft errors. However, the instructions executed beyond the ROB in runahead mode are speculative. Similar to RA and RA-buffer, bits exposed by these instructions are un-ACE bits. Since these instructions substantially improve the performance of the applications in PRE without contributing any vulnerable microarchitectural state, the overall number of bits exposed in PRE is lower than is the case for a conventional out-of-order core.

We again use ACE analysis (see Section 2.3) for computing vulnerable processor bits, and the bits per entry for back-end structures are calculated according to Table 4.2. We use ABC as the metric for quantifying soft error

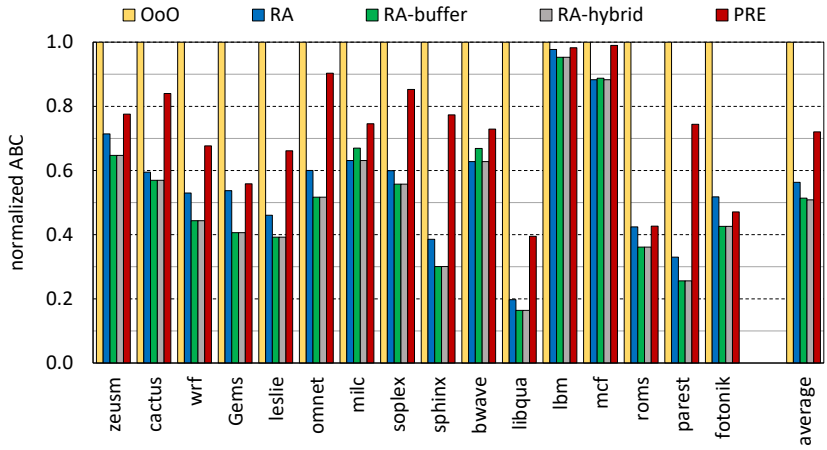

Figure 5.17: Comparing the impact on reliability for all runahead techniques. *Relative to an out-of-order core, all runahead techniques improve reliability, however, the improvement for prior runahead techniques is higher than PRE.*

reliability of applications. It is worthwhile to note that we deem mean time to failure (MTTF) as an appropriate metric in Chapter 4 while quantifying reliability of applications under dispatch halting. The reason MTTF correctly quantifies reliability of single-program workloads under dispatch halting is because the dispatch halting has a minor impact on performance. Therefore, ABC directly translates to MTTF; FIT rate and MTTF have also been widely popular metrics in the fault-tolerance community for a long time. Since all the runahead techniques improve single-thread performance on an out-of-order processor, ABC is a more suitable metric than MTTF for quantifying the impact of all runahead techniques on reliability.

Figure 5.17 shows ABC for RA, RA-buffer, RA-hybrid, and PRE, relative to the baseline OoO core. Please refer to Figure 5.9 for the corresponding performance comparison of all the techniques. Overall, the ABC for RA decreases with increasing gains in performance, and RA is able to reduce ABC by 43.7% across all benchmarks. For benchmarks such as `sphinx3`, `libquantum` and `parest`, RA reduces the number of exposed ACE bits by more than 60%. For these benchmarks, RA also reports a substantial performance gain of 15.6%, 53.6% and 21.9%, respectively. The majority of other benchmark — like `cactusADM`, `wrf`, `GemsFDTD`, `leslie3d`, and `fotonik`, etc., experience a reduction in ABC under RA in the range of 40–60%. For benchmarks that do not encounter frequent full-window stalls, like `lbm` and `mcf`, the improvement in ABC (and performance) is also the lowest among all benchmarks. Interestingly, RA-buffer achieves only marginally higher reduction in ABC compared to RA, while also performing slightly worse than RA. This is because the core is in runahead mode for a longer duration than RA and each runahead interval brings a lower gain in performance than RA. RA-buffer only executes future slices of a stalling load, missing opportunity to trigger memory requests for other long-latency loads. The other long-latency loads also trigger runahead

	performance benefit	**energy requirement**	**reliability improvement**
OoO	neutral	neutral	neutral
RA	high	high	high
RA-buffer	high	neutral	high
PRE	very high	low	medium

Table 5.2: Performance, energy and reliability of the runahead techniques compared to an out-of-order processor.

mode when they lead to full-window stalls. Overall, analogous to their respective gains in performance, RA and RA-buffer achieve similar improvements in ABC. RA-hybrid, which selects the better choice between RA and RA-buffer for each benchmark, also achieves similar reductions in ABC as RA-buffer. In PRE, the ACE bits exposed by the instructions in the ROB, and the other back-end structures these instructions occupy between dispatch and commit, are also vulnerable. This is clearly reflected in ABC for PRE in Figure 5.17. For all benchmarks, PRE exposes more ACE bits than RA and RA-buffer, and achieves an overall reduction in ABC by 28% compared to the OoO core.

Table 5.2 compares the performance, energy and reliability of runahead techniques to an out-of-order core. Although both RA and RA-buffer substantially improve performance, nevertheless, their performance can be further improved by mitigating overheads and improving prefetch coverage. PRE almost doubles the performance benefit of runahead techniques by eliminating these overheads. While RA-buffer is energy-neutral relative to the OoO core because it clock-gates the front-end in runahead mode, RA increases the energy requirement of the core as a result of processing instructions executed in runahead mode twice. PRE, in contrast, reduces the energy requirement of the core because it does not process instructions in the ROB again, and it only (speculatively) executes instructions that are required to generate prefetches. All runahead techniques improve reliability, however, RA and RA-buffer achieve significantly higher gains than PRE. PRE exposes more vulnerable microarchitectural state than RA and RA-buffer because PRE never flushes the ROB and the other back-end structures occupied by the instructions in the ROB. Overall, PRE is a a new runahead execution technique that improves performance, energy, and reliability of the out-of-order core with a small area overhead of only 1.24 KB.

5.7 Related Work

A large body of processor microarchitecture research has focused on improving single-thread performance over the past four decades. Some proposals scaled microarchitecture structures for better performance and energy-efficiency. Examples include the work that dynamically scale operating voltage and clock frequency [28, 92, 165] or resize critical structures like issue queue [29, 63, 87, 108, 163] and caches [4, 5, 14] or throttle the front-end

pipeline [125]. PRE, however, fits in the category of work that performs some form of runahead execution, pre-execution or prefetching.

Runahead. PRE improves upon the runahead execution proposed within a single core [79, 139, 140, 141, 142, 143]. Since traditional runahead execution cannot prefetch dependent long-latency load instructions, address-value delta [142] predicts the data value of earlier long-latency load instruction to enable the execution of future long-latency load instructions. Enhanced memory controller [80] filters this chain of dependent long-latency load instructions and executes it at the memory controller; now, the dependent load instruction can execute as soon as the data is available from DRAM. Because the effective runahead interval shortens with the increasing size of the ROB, continuous runahead [81] proposes a tiny accelerator that is located at the last-level cache controller of a multi-core chip. The accelerator executes the dependency chain that leads to the highest number of full-window stalls within the core. However, the area overhead of the accelerator is 2% of a quad-core chip, and possibly higher for a single core. Prior work have also enabled runahead threads in an SMT processor [48, 168, 169]. PRE is a runahead technique that does not require a separate core or runahead thread to pre-execute stalling slices.

Pre-Execution. This category of work executes performance critical future instruction slices early in a software-only, hardware-only or a hardware-software cooperative fashion. Helper threads [102] and speculative precomputation [46] are software-only techniques that require a hardware context for early execution. Hardware-only techniques filter critical instruction slices from the back-end of a processor for early execution on a separate hardware context [47, 225]. Waiting instruction buffer (WIB) [115] and continual flow pipelines (CFP) [195] can execute a large number of independent instructions by releasing the resources occupied by miss-dependent instructions. BOLT [85] builds upon CFP but reuses SMT hardware to rename deferred slices and introduces a set of mechanism to avoid useless pre-execution slices. Slipstream processors [198] also improves performance and reliability by precomputing demand misses. Dependence graph precomputation (DGP) [9] dynamically precomputes and executes instructions responsible for memory accesses on a separate execution engine. Dual-core [220] and explicitly-decoupled architecture (EDA) [65, 66, 107, 160] use two hardware threads where one thread feeds its output to the other. Hardware-software cooperative techniques involve new instructions, advanced profiling or binary translation to separating critical instructions slices. Examples of such proposal are DAE [184], speculative slice execution [224], flea-flicker multi-pass pipelining [16], braid processing [205] and OUTRIDER [49]. Instruction slices have also been exploited to improve the energy-efficiency of both inorder and out-of-order processors [33, 114, 174, 202, 203]. PRE does not require a separate helper thread, hardware context or support from software for converting demand misses into hits.

Prefetching. Hardware prefetchers are typically employed in modern processors [91]. Stride or stream prefetchers are able to prefetch simple data

access patterns that are independent of other memory accesses [51, 95, 159]. The accesses are either contiguous or separated by a constant stride. Address correlating prefetchers require larger tables and target pointer-chasing access patterns [13, 35, 41, 187, 188, 215, 216, 219]. These prefetchers build on the premise that data structures are typically accessed in the same manner, generating same cache misses repeatedly. Global History Buffer (GHB) [152] splits the correlation table into two separate structures and also lowers the harware overhead. PRE is implemented completely within the core and it is orthogonal to the hardware prefetching techniques.

5.8 Summary

Hiding and minimizing long memory accesses continues to be a challenging task for improving performance. Runahead execution improves processor performance by accurately prefetching long-latency memory accesses. When the instruction window of an out-of-order core is stalled on a long-latency load, the runahead execution marks the result of the stalling load instruction and its dependents as bogus and continues to speculatively execute future long-latency load instructions to prefetch their data closer to the core. All the instructions starting from the stalling load instructions are re-executed when normal execution resumes. To reduce the overhead of runahead execution, runahead buffer stores the chain of instructions leading to the stalling load at the dispatch stage of the pipeline and executes only this chain after a stalled window for improved energy-efficiency. We show that the performance of prior runahead proposals is limited by the high overhead they incur and the limited prefetch coverage they achieve. Prior proposals release processor state when entering runahead mode and need to re-fill the pipeline when resuming normal operation. This operation introduces significant performance overhead. Moreover, prior proposals have limited prefetch coverage due to executing instructions that are unnecessary to generate prefetches as in the original runahead proposal, or due to not exploring all possible stalling loads as in runahead buffer.

In this chapter, we propose *Precise Runahead Execution (PRE)*, to alleviate the shortcomings of prior runahead proposals. We observe that at the entry of runahead mode, there are sufficient free PRF and IQ resources to speculatively execute instructions without having to release processor state. PRE does not incur the performance overhead of refilling the pipeline when resuming normal operation, by featuring a novel mechanism to quickly recycle physical registers in runahead mode. Furthermore, PRE tracks all stalling slices in a dedicated cache, which it executes in runahead mode, i.e., PRE filters unnecessary instructions and pre-executes all stalling slices to improve prefetch coverage. In PRE, the instructions executed in runahead mode beyond the ROB need to be fetched and decoded again. Therefore, PRE also optionally buffers these instructions in the EMQ. For a set of representative memory-intensive benchmarks, PRE improves performance by 38.2% relative to the baseline out-of-order core, while the best performing runahead tech-

nique achieves only 20%. PRE also reduces energy consumption by 6.8% while runahead techniques are energy-neutral relative to the out-of-order core. This substantial gain in both performance and energy is an outcome of no pipeline refilling overhead, increased memory-level parallelism, and high prefetch coverage in PRE. All the runahead techniques including PRE improve soft error reliability of the out-of-order core as a consequence of speculatively executing instructions. Although the gain in reliability incurred by PRE is lower than the prior runahead techniques, nevertheless, PRE still reduces the soft error vulnerability of the out-of-order core by 28%.

Chapter 6

Conclusion

This chapter summarizes the key conclusions drawn from this dissertation. In addition, the chapter presents several directions for future work.

6.1 Summary

Soft errors pose an imminent threat to the correct working of modern processors. The charge that distinguishes between the binary states of a processor bit has reduced to a very small amount as a result of the continuous device scaling trends. Energy particles from cosmic rays and packaging impurities can easily strike the vulnerable processor bits and result in incorrect program execution. Large amount of microarchitectural state inside modern general-purpose out-of-order cores requires novel solutions to mitigate the chance of an application encountering a soft error on these *big* cores. An out-of-order core is equally vulnerable when executing single-threaded applications or when integrated alongside cores of different complexity and size, as in heterogeneous multicore processors. This dissertation contributes three new techniques for improving soft error reliability and performance on modern processors.

Scheduling for Optimizing Reliability on Heterogeneous Multicore Processors. Reliability-aware scheduling, our first contribution, mitigates soft error vulnerability on heterogeneous multicore processors. We observe that in addition to performance and energy-efficiency, each core type in an HCMP also shows different soft error vulnerability characteristics. A big out-of-order core features substantially more transistors, and is therefore more vulnerable to soft errors than a small core. On the other hand, a big core executes an application faster, reducing its exposure to soft errors between launching and finishing the application. The difference in soft error vulnerability across core types and applications opens opportunities for reliability-aware scheduling to improve system reliability. We demonstrate that reliability-aware scheduling can substantially improve the soft error reliability of multiprogram workloads

running on a heterogeneous multicore processor. When analyzing the ACE bits on a multicore processor, one of our key observations was the lack of a suitable metric to assess reliability, since benchmarks running on a multicore processor impact the performance of one another. Therefore, we also developed System Soft Error Rate (SSER), a metric for quantifying reliability of multi-program workloads on (heterogeneous) multicore processors. We evaluate our scheduler across core count, different frequency settings, and varying degree of heterogeneity. The impact on performance is low, only 6.3%, for a 25.4% improvement in system reliability when running four-program workloads on two big and two small cores.

Reliability-aware scheduling also reduces power consumption — this is because an application that executes on a big core for high performance can be scheduled on a small core for high reliability, thus also reducing power consumption. For systems with a strict performance limit, we noticed that there is higher improvement in reliability through reliability-aware scheduling as the performance limit is relaxed. Compared to the diverse runtime behavior of benchmarks making multiprogram workloads, threads of multithreaded workloads show mostly homogeneous behavior. Consequently, there is not a significant opportunity to improve reliability for multithreaded workloads. Compared to the core, the amount of vulnerable state maintained inside on-chip caches is significantly higher. Therefore, the difference in the number of exposed ACE bits between big and small cores with equally-sized on-chip caches is relatively smaller, which lowers the potential gain from reliability-aware scheduling.

Dispatch Halting for Optimizing Reliability on Out-of-Order Cores. Dispatch halting, our second contribution, reduces soft error vulnerability of memory-intensive applications on out-of-order cores. Out-of-order cores expose a large vulnerable microarchitectural state when running applications. We primarily focus on the memory-intensive benchmarks since the instructions accessing memory take much longer to leave the pipeline than other instructions. A memory access typically stalls the processor for (at least) a couple hundreds of processor cycles. In particular, a long-latency load miss that waits for memory blocks commit, while the front-end keeps on dispatching instructions into the back-end; eventually, the ROB and possibly the IQ, Load Queue (LQ) and Store Queue (SQ) fill up with instructions. This exposes a large architectural state for a long time window. We find that 67% (and up to 87%) of the vulnerable correct-path state is exposed due to long-latency load misses for the memory-intensive workloads in SPEC CPU2006.

Dispatch halting is built on the key insight that it is better to speculate than execute in normal mode under a memory access. Therefore, dispatch halting converts instructions following a long-latency load into speculative instructions, and buffers a copy of those instructions for future execution in normal mode. In normal mode, the same instructions occupy the processor back-end for a much shorter duration. We propose two variants, proactive and reactive dispatch halting. Proactive dispatch halting stops dispatch after a (predicted) long-latency load miss, and to preserve performance on par with a conventional out-of-order core, selectively copies loads and branches, along with their producer

instructions, from EMQ to the back-end for speculative pre-execution to expose memory-level parallelism (MLP) and resolve mispredicted branches that are independent of the long-latency load. In reactive dispatch halting, dispatch is halted when a long-latency load miss blocks commit at the ROB head for a given number of cycles. The instructions in the ROB turn into speculative execution mode to preserve MLP, and are flushed when the load is about to return. When exiting dispatch halting, the instructions buffered in the EMQ are dispatched into the back-end for normal execution. We demonstrate that dispatch halting is able to successfully increase the mean time to failure for memory-intensive benchmarks by a factor of 2× while incurring marginal overheads in terms of performance, power and chip area.

Precise Runahead Execution for Optimizing Performance on Out-of-Order Cores. Precise runahead execution, our third contribution, focuses on single-thread performance on out-of-order cores, and also provides a comparison of the impact of runahead techniques on reliability. We notice that the performance benefits of prior runahead proposals are limited by their prefetch coverage and the overheads associated with speculative code execution. Prior runahead techniques flush the instruction window, execute unnecessary instructions, and do not exploit the short runahead intervals for generating prefetches. PRE remedies these shortcomings of prior runahead techniques.

PRE only speculatively pre-executes slices of load instructions that lead to full-window stalls. These slices are executed within an existing single core without the support of any extra (helper) core or context. PRE does not flush any of the back-end structures and leverages the available issue queue and physical register file entries to speculatively execute instructions in runahead mode. We note that the issue queue entries do not hinder the execution in PRE, however, physical register file entries must be recycled to continue making forward progress for the entire duration of a stalled window. Therefore, we devise a novel mechanism to efficiently recycle physical registers in runahead mode. Additionally, PRE optionally buffers decoded micro-ops during runahead mode in an extended micro-op queue to avoid re-fetching and re-decoding instructions, thereby saving energy. PRE improves reliability compared to an out-of-order core as a consequence of speculatively executing instructions beyond the ROB. However, since instructions in the ROB are not flushed in PRE, the reliability improvement accrued by PRE is lower than prior runahead techniques. The microarchitectural enhancements added by PRE over prior runahead techniques also lead to a substantially higher performance and lower energy requirement.

6.2 Future Work

Reliability-Aware Scheduling on Heterogeneous Processor and Memory Architectures. Heterogeneous memory architectures are widely adopted today for high capacity, high bandwidth, and low energy requirements. However, memory systems comprising of disparate memory technologies like dynamic random access memory (DRAM), die-stacked memory or non-volatile

memory like phase change memory, also exhibit different performance and reliability trade-offs. For example, die-stacked memories provide higher bandwidth but are less reliable than traditional DRAM. Prior work [75] has improved reliability of applications running on hybrid memory system designed with die-stacked memory alongside traditional DRAM.

Reliability-aware scheduling proposed in this dissertation improves reliability on heterogeneous multicores. In a system where both memory and processor are heterogeneous, one direction for future work is to explore the impact of optimizing processor reliability on (heterogeneous) memory systems, and, vice-versa. A novel reliability-aware scheduling mechanism can be designed to improve the overall system reliability to soft errors.

Exploring the Security Implications of Runahead Execution. In an out-of-order core, the number of instructions executed along a wrong path after a mispredicted branch or a trap is limited by the ROB size. Runahead execution extends the window of executed instructions to a larger size than the ROB, thus increasing the security risk as a result of exploiting aggressive speculation. If there is a (mispredicted) branch in the ROB that depends on a stalling load, PRE will go too deep in the dynamic instruction stream, triggering future memory accesses. These prefetches along the wrong path for the entire duration of a stalling load can leak a large amount of information, in a Spectre-like fashion [106].

Similarly, if a load instruction causing a trap depends on a stalling load (for example, miss-dependent misses), PRE will again fetch a large amount of data into the caches as a result of executing load instructions that are later flushed when the (delayed) trap check is performed. This causes security risks in a Meltdown-like fashion [122]. An interesting direction for future research is to quantify the amount of wrong-path data prefetched into caches by the runahead techniques. Furthermore, it can be explored whether the latest techniques (for example, CleanupSpec [173]) are able to mitigate attacks under aggressive speculation with marginal performance overhead. It is possible that the runahead techniques require designing even better mitigation techniques for defense against speculation-based attacks.

Bibliography

[1] A. Adileh, S. Eyerman, A. Jaleel, and L. Eeckhout. Maximizing heterogeneous processor performance under power constraints. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(3), Sept. 2016.

[2] A. Adileh, S. Eyerman, A. Jaleel, and L. Eeckhout. Mind the power holes: Sifting operating points in power-limited heterogeneous multicores. *IEEE Computer Architecture Letters*, 16(1):56–59, 2017.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–434, Dec. 2003.

[4] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 248–259, Nov. 1999.

[5] D. H. Albonesi, R. Balasubramonian, S. G. Dropsbo, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, 2003.

[6] R. Alves, A. Ros, D. Black-Schaffer, and S. Kaxiras. Filter caching for free: The untapped potential of the store-buffer. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, pages 436–448, 2019.

[7] AMD. The future is fusion: The industry-changing impact of accelerated computing. http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf, 2008.

[8] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3 GHz fifth generation SPARC64 microprocessor. *2003 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 246–491, 2003.

[9] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–61, 2001.

[10] J. Arlat, Y. Crouzet, and J. . Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, pages 348–355, 1989.

[11] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. . Fabre, J. . Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *IEEE Transactions on Software Engineering*, pages 166–182, 1990.

[12] G.-H. Asadi, V. S. Mehdi, B. Tahoori, and D. Kaeli. Balancing performance and reliability in the memory hierarchy. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 269–279, 2005.

[13] J. . Baier and G. R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Transactions on Software Engineering*, SE-2(1): 54–62, 1976.

[14] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO)*, pages 245–257, 2000.

[15] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, pages 26–37, 2001.

[16] R. D. Barnes, S. Ryoo, and W. W. Hwu. "flea-flicker" multipass pipelining: an alternative to the high-power out-of-order offense. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 319–330, 2005.

[17] R. Baumann. *IEEE Reliability Physics Tutorial Notes, Reliability Fundamentals.*

[18] R. Baumann, T. Hossain, E. Smith, S. Murata, and H. Kitagawa. Boron as a primary source of radiation in high density drams. In *Symposium on VLSI Technology*, pages 81–82, June 1995.

[19] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5 (3):305–316, Sept 2005.

[20] A. Bhattacharjee and M. Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 290–301, June 2009.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct. 2008.

[22] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 532–543, 2005.

[23] A. Biswas, P. Racunas, J. Emer, and S. Mukherjee. Computing accurate avfs using ace analysis on performance models: A rebuttal. *IEEE Computer Architecture Letters*, 7(1):21–24, 2008.

[24] A. Biswas, C. Recchia, S. S. Mukherjee, V. Ambrose, L. Chan, A. Jaleel, A. E. Papathanasiou, M. Plaster, and N. Seifert. Explaining cache SER anomaly using DUE AVF measurement. In *Proceedings of the 16th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[25] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, pages 10–16, 2005.

[26] S. Borkar, N. P. Jouppi, and P. Stenstrom. Microprocessors in the era of terascale integration. In *2007 Design, Automation Test in Europe Conference Exhibition*, pages 1–6, 2007.

[27] G. Bronevetsky, B. R. de Supinski, and M. Schulz. A foundation for the accurate prediction of the soft error vulnerability of scientific applications. Technical report, Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2009.

[28] T. Burd and R. Brodersen. Design issues for dynamic voltage scaling. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 9–14, June 2000.

[29] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, pages 147–156, 2003.

[30] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron CMOS technology. *IEEE Transactions on Nuclear Science*, pages 2874–2878, 1996.

[31] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, and M. Snir. Toward exascale resilience. pages 374–388, 2009.

[32] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):28, 2014.

[33] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout. The load slice core microarchitecture. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 272–284, 2015.

[34] J. Carreira, H. Madeira, and J.-P. Gabriel. Xception: Software fault injection and monitoring in processor functional units. In *Proceedings of 5th International Working Conference on Dependable Computing for Critical Applications*, 1995.

[35] M. J. Charney. *Correlation-based Hardware Prefetching.* PhD thesis, 1995.

[36] A. Chatzidimitriou and D. Gizopoulos. Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 69–78, 2016.

[37] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos, and P. Rech. Demystifying soft error assessment strategies on ARM CPUs: Microarchitectural fault injection vs. neutron beam experiments. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 26–38, 2019.

[38] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct. 2009.

[39] J. Chen and L. K. John. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th Annual Design Automation Conference (DAC)*, pages 927–930, 2009.

[40] E. Cheng, S. Mirkhani, L. G. Szafaryn, C. Y. Cher, H. Cho, K. Skadron, M. R. Stan, K. Lilja, J. A. Abraham, P. Bose, and S. Mitra. Tolerating soft errors in processor cores using CLEAR (Cross-Layer Exploration for Architecting Resilience). *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–16, 2017.

[41] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 199–209, 2002.

[42] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer. Quickia: Exploring heterogeneous architectures on real prototypes. In *Proceedings of the High Performance Computer Architecture (HPCA)*, pages 1–8, 2012.

[43] H. Cho, S. Mirkhani, C. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Annual Design Automation Conference (DAC)*, pages 1–10, 2013.

[44] C. Chou, A. Jaleel, and M. Qureshi. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, pages 268–280, 2017.

[45] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 76–87, June 2004.

[46] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 14–25, July 2001.

[47] J. D. Collins, D. M. Tullsen, , and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual International Symposium on Microarchitecture (MICRO)*, pages 306–317, 2001.

[48] K. V. Craeynest, S. Eyerman, and L. Eeckhout. MLP-aware runahead threads in a simultaneous multithreading processor. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, pages 110–124, 2009.

[49] N. C. Crago and S. J. Patel. OUTRIDER: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 117–128, 2011.

[50] E. W. Czeck and D. P. Siewiorek. Effects of transient gate-level faults on program behavior. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing*, pages 236–243, 1990.

[51] F. Dahlgren and P. Stenström. Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors. In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 68–, 1995.

[52] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[53] J. J. Dongarra, H. Meuer, and E. Strohmaier. Top500 supercomputer sites, June 2018. URL `http://www.top500.org`.

[54] J. Doweck, W. F. Kao, A. K. y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, pages 52–62, 2017.

[55] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 511–522, June 2013.

[56] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 355–372, Oct. 2013.

[57] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 68–75, July 1997.

[58] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for HPC. In *Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 615–626, 2012.

[59] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[60] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multi-Program Workloads. *IEEE Micro*, 28(3):42–53, May/June 2008.

[61] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 221–230, 2014.

[62] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs. `https://www.agner.org/optimize/microarchitecture.pdf`.

[63] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 230–239, 2001.

[64] V. Fratin, D. Oliveira, C. Lunardi, F. Santos, G. Rodrigues, and P. Rech. Code-dependent and architecture-dependent reliability behaviors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 13–26, 2018.

[65] A. Garg and M. C. Huang. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st International Symposium on Microarchitecture (MICRO)*, pages 306–317, 2008.

[66] A. Garg, R. Parihar, and M. C. Huang. Speculative parallelization in decoupled look-ahead. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 413–423, 2011.

[67] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA)*, pages 98–109, 2003.

[68] A. González. *Trends in Processor Architecture*. Springer International Publishing, 2019.

[69] A. Gonzalez, S. Mahlke, S. Mukherjee, R. Sendag, D. Chiou, and J. J. Yi. Reliability: Fallacy or reality? *IEEE Micro*, 27(6):36–45, 2007.

[70] A. González, F. Latorre, and G. Magklis. *Processor Microarchitecture: An Implementation Perspective*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2011.

[71] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, 2003.

[72] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7: Improving energy efficiency in high-performance mobile platforms. http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf, Sept. 2011.

[73] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proceedings of the 19th International Symposium on Fault-Tolerant Computing*, pages 340–347, 1989.

[74] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta. Compiler techniques to reduce the synchronization overhead of GPU redundant multithreading. In *Proceedings of the 54th Design Automation Conference (DAC)*, pages 1–6, 2017.

[75] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta. Reliability-aware data placement for heterogeneous memory architecture. In *Proceedings of the 24th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 583–595, 2018.

[76] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne,

R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton. Haswell: The fourth-generation intel core processor. *IEEE Micro*, pages 6–20, 2014.

[77] I. S. Haque and V. S. Pande. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID)*, pages 691–696, 2010.

[78] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer. SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258, 2017.

[79] M. Hashemi and Y. N. Patt. Filtered runahead execution with a runahead buffer. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 358–369, 2015.

[80] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt. Accelerating dependent cache misses with an enhanced memory controller. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, pages 444–455, 2016.

[81] M. Hashemi, O. Mutlu, and Y. N. Patt. Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In *Proceedings of the 49th International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[82] W. Heirman, S. Sarkar, T. E. Carlson, I. Hur, and L. Eeckhout. Power-aware multi-core simulation for early design stage hardware/software co-optimization. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–12, 2012.

[83] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proceedings of the 50th Design Automation Conference (DAC)*, pages 1–10, 2013.

[84] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition, 2003.

[85] A. Hilton and A. Roth. Bolt: Energy-efficient out-of-order latency-tolerant execution. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[86] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[87] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proceedings of the 2004 International Symposium on Low Power Electronic Design (ISLPED)*, pages 32–37, 2004.

[88] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[89] Intel. 2nd generation Intel Core vPro processor family. http://www.intel.com/content/dam/doc/white-paper/core-vpro-2nd-generation-core-vpro-processor-family-paper.pdf, 2008.

[90] Intel. Intel® Xeon Phi™ coprocessor system software developers guide. `https://software.intel.com/sites/default/files/managed/09/07/xeon-phi-coprocessor-system-software-developers-guide.pdf`, 2014.

[91] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, Apr. 2019.

[92] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 347–358, Dec. 2006.

[93] JEDEC. Measurement and reporting of alpha particle and terrestrial cosmic ray-induced soft errors in semiconductor devices. Technical report, JESD89A, JEDEC Standard, 2006.

[94] J. Joao, M. Suleman, O. Mutlu, and Y. Patt. Bottleneck Identification and Scheduling in Multithreaded Applications . In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–234, Mar. 2012.

[95] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, pages 364–373, June 1990.

[96] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke,

A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[97] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49:589–604, July 2005.

[98] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez. MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment. In *Proceedings of the 44th International Symposium on Computer Architecture (ISCA)*, pages 241–254, 2017.

[99] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: a tool for the validation of system dependability properties. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 336–344, 1992.

[100] J. Karlsson and P. Folkesson. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *Proceedings of the 5th Annual IEEE International Working Conference on Dependable Computing for Critical Applications*, pages 267–287. IEEE Computer Society Press, 1995.

[101] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2008.

[102] D. Kim and D. Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Transactions on Computer Systems (TOCS)*, 22(3):326–379, 2004.

[103] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, 2003.

[104] S. Kim and A. K. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 416–425, 2002.

[105] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the*

41st Annual International Symposium on Computer Architecture (ISCA), pages 361–372, 2014.

[106] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[107] S. Kondguli and M. Huang. R3-DLA (reduce, reuse, recycle): A more efficient approach to decoupled look-ahead architectures. In *Proceedings of the 25th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 533–544, 2019.

[108] Y. Kora, K. Yamaguchi, and H. Ando. MLP-aware dynamic instruction window resizing for adaptively exploiting both ILP and MLP. In *Proceedings of the 46th International Symposium on Microarchitecture (MICRO)*, pages 37–48, 2013.

[109] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, pages 125–138, 2010.

[110] F. Kriebel, A. Subramaniyan, S. Rehman, S. J. B. Ahandagbe, M. Shafique, and J. Henkel. R2cache: Reliability-aware reconfigurable last-level cache architecture for multi-cores. In *Proceedings of the International Conference on Hardware and Software Codesign and System Synthesis (CODES + ISSS)*, pages 1–10, 2015.

[111] F. Kriebel, S. Rehman, A. Subramaniyan, S. J. B. Ahandagbe, M. Shafique, and J. Henkel. Reliability-aware adaptations for shared last-level caches in multi-cores. *ACM Transactions on Embedded Computing Systems*, 15(4):1–26, Aug. 2016.

[112] 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 *36th International Symposium on Microarchitecture (MICRO)*, pages 81–92, 2003.

[113] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 64–75, 2004.

[114] R. Kumar, M. Alipour, and D. Black-Schaffer. Freeway: Maximizing mlp for slice-out-of-order execution. In *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 558–569, 2019.

[115] A. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings*

of the 29th Annual International Symposium on Computer Architecture (ISCA), pages 59–70, 2002.

[116] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *Proceedings of the 2009 Conference on Design Automation and Test in Europe (DATE)*, pages 502–506, 2009.

[117] D. Li, J. S. Vetter, and W. Yu. Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2012.

[118] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec. 2009.

[119] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi. CACTI-P: Architecture-level modeling for SRAM-based structures with advanced leakage reduction techniques. In *ICCAD)*, pages 694–701, 2011.

[120] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Online estimation of architectural vulnerability factor for soft errors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, pages 341–352, 2008.

[121] D. Lide, L. Bin, and P. Lu. Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 129–140, 2009.

[122] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[123] G. H. Loh. 3D-Stacked memory architectures for multi-core processors. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pages 453–464, 2008.

[124] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 317–328, 2012.

[125] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, (ISCA)*, pages 132–141, 1998.

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

[127] N. Mehta, B. Singer, R. I. Bahar, M. Leuchtenburg, and R. Weiss. Fetch halting on critical load misses. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 244–249, Oct. 2004.

[128] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, pages 75–82, 1997.

[129] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, pages 329–335, 2005.

[130] S. E. Michalak, A. J. DuBois, C. B. Storlie, H. M. Quinn, W. N. Rust, D. H. DuBois, D. G. Modl, A. Manuzzato, and S. P. Blanchard. Assessment of the impact of cosmic-ray-induced neutrons on hardware in the Roadrunner supercomputer. *IEEE Transactions on Device and Materials Reliability*, pages 445–454, 2012.

[131] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics Magazine*, 38(8):1–6, Apr. 1965.

[132] G. E. Moore. Progress in digital integrated electronics. *Technical Digest. International Electron Devices Meeting, IEEE*, pages 11–13, 1975.

[133] A. Moshovos. Checkpointing alternatives for high-performance, power-aware processors. In *Proceedings of the 2003 International Symposium on Low Power Electronic Design (ISLPED)*, pages 318–321, 2003.

[134] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers, 2008.

[135] S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt. Cache scrubbing in microprocessors: Myth or necessity? In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 37–42, 2004.

[136] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, pages 99–110, 2002.

[137] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 29–40, 2003.

[138] T. S. Muthukaruppan, A. Pathania, and T. Mitra. Price theory based power management for heterogeneous multi-cores. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 161–176, 2014.

[139] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 129–140, Feb. 2003.

[140] O. Mutlu, , J. Stark, and Y. N. Patt. On reusing the results of pre-executed instructions in a runahead execution processor. *IEEE Computer Architecture Letters*, 4(1):2–2, Jan 2005.

[141] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 370–381, June 2005.

[142] O. Mutlu, H. Kim, and Y. N. Patt. Address-value delta (AVD) prediction: increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 233–244, 2005.

[143] O. Mutlu, , and Y. N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro*, 26(1):10–20, 2006.

[144] A. A. Nair, L. K. John, and L. Eeckhout. AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors. In *Proceedings of the 43rd Annual International Symposium on Microarchitecture (MICRO)*, pages 125–136, 2010.

[145] A. A. Nair, S. Eyerman, L. Eeckhout, and L. K. John. A first-order mechanistic model for architectural vulnerability factor. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 273–284, 2012.

[146] A. Naithani and L. Eeckhout. Dispatch halting. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2020.

[147] A. Naithani, S. Eyerman, and L. Eeckhout. Reliability-aware scheduling on heterogeneous multicore processors. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 397–408, 2017.

[148] A. Naithani, S. Eyerman, and L. Eeckhout. Optimizing soft error reliability through scheduling on heterogeneous multicore processors. *IEEE Transactions on Computers*, 67(6):830–846, 2018.

[149] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. Precise runahead execution. *IEEE Computer Architecture Letters*, 18(1):71–74, 2019.

[150] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout. Precise runahead execution. In *Proceedings of the 26th IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[151] National Security Agency, Advanced Computing Systems. Inter-agency workshop on HPC resilience at extreme scale, 2012.

[152] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *Proceedings of the 10th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 96–105, 2004.

[153] M. Nicolaidis. Design for soft error mitigation. *IEEE Transactions on Device and Materials Reliability*, 5(3):405–418, 2005.

[154] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, pages 2742–2750, 1996.

[155] NVidia. Variable SMP – a multi-core CPU architecture for low power and high performance. http://www.nvidia.com/content/PDF/tegra_white_papers/Variable-SMP-A-Multi-Core-CPU-Architecture-for-Low-Power-and-High-Performance.pdf, 2011.

[156] D. Oliveira, L. Pilla, N. DeBardeleben, S. Blanchard, H. Quinn, I. Koren, P. Navaux, and P. Rech. Experimental and analytical study of xeon phi reliability. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 28:1–28:12, 2017.

[157] D. A. G. D. Oliveira, L. L. Pilla, M. Hanzich, V. Fratin, F. Fernandes, C. Lunardi, J. M. Cela, P. O. A. Navaux, L. Carro, and P. Rech. Radiation-induced error criticality in modern hpc parallel accelerators. In *Proceedings of the 23rd Annual International Symposium on High Performance Computer Architecture (HPCA)*, pages 577–588, 2017.

[158] K. Olukotun, L. Hammond, and J. Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency.* Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2007.

[159] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 24–33, 1994.

[160] R. Parihar and M. C. Huang. Accelerating decoupled look-ahead via weak dependence removal: A metaheuristic approach. In *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 662–677, 2014.

[161] J. Peir, S. Lai, S. Lu, J. Stark, and K. Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proceedings of the 2002 International Conference on Supercomputing (ICS)*, pages 347–356, 2002.

[162] S. Petit, R. Ubal, J. Sahuquillo, and P. López. Efficient register renaming and recovery for high-performance processors. *IEEE Transaction on VLSI Systems*, pages 1506–1514, 2014.

[163] D. V. Ponomarev, G. Kucuk, O. Ergin, K. Ghose, and P. M. Kogge. Energy-efficient issue queue design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(5):789–800, 2003.

[164] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, 2014.

[165] G. Qu. What is the limit of energy saving by dynamic voltage scaling? In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 560–563, Nov. 2001.

[166] M. K. Qureshi, O. Mutlu, and Y. N. Patt. Microarchitecture-based introspection: a technique for transient-fault tolerance in microprocessors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 434–443, 2005.

[167] S. Raasch, A. Biswas, J. Stephan, P. Racunas, and J. Emer. A fast and accurate analytical technique to compute the avf of sequential bits in a processor. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pages 738–749, 2015.

[168] T. Ramirez, A. Pajuelo, O. J. Santana, and M. Valero. Runahead threads to improve SMT performance. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 149–158, Feb. 2008.

[169] T. Ramirez, A. Pajuelo, O. J. Santana, O. Mutlu, and M. Valero. Efficient runahead threads. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 443–452, 2010.

[170] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 25–36, 2000.

[171] E. Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 84–, 1999.

[172] E. Safi, A. Moshovos, and A. Veneris. On the latency and energy of checkpointed superscalar register alias tables. *IEEE Transaction on VLSI Systems*, pages 365–377, 2010.

[173] G. Saileshwar and M. K. Qureshi. CleanupSpec: An "undo" approach to safe speculation. In *Proceedings of the 52nd International Symposium on Microarchitecture (MICRO)*, pages 73–86, 2019.

[174] A. Sembrant, T. Carlson, E. Hagersten, D. Black-Shaffer, A. Perais, A. Seznec, and P. Michaud. Long term parking (LTP): Criticality-aware resource allocation in ooo processors. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 334–346, 2015.

[175] L. Sethumadhavan. *Scalable Hardware Memory Disambiguation.* PhD thesis, The University of Texas at Austin, 2007.

[176] Seungjae Han, K. G. Shin, and H. A. Rosenberg. DOCTOR: an integrated software fault injection environment for distributed real-time systems. In *Proceedings of International Computer Performance and Dependability Symposium*, pages 204–213, 1995.

[177] A. Seznec. TAGE-SC-L branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[178] S. Z. Shazli, M. Abdul-Aziz, M. B. Tahoori, and D. R. Kaeli. A field analysis of system-level effects of soft errors occurring in microprocessors used in information systems. In *IEEE International Test Conference*, pages 1–10, 2008.

[179] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. Hass: a scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review*, 43 (2):66–75, 2009.

[180] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, 2002.

[181] K. G. Shin and Y.-H. Lee. Measurement and application of fault latency. *IEEE Transactions on Computers*, pages 370–375, 1986.

[182] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[183] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind,

M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. Ibm power8 processor core microarchitecture. *IBM Journal of Research and Development*, pages 2:1–2:21, 2015.

[184] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th Annual International Symposium on Computer Architecture (ISCA)*, pages 112–119, 1982.

[185] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, pages 562–573, 1988.

[186] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 224–234, 2004.

[187] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, pages 252–263, 2006.

[188] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 69–80, 2009.

[189] D. Sorin. *Fault Tolerant Computer Architecture*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2009.

[190] N. K. Soundararajan, A. Parashar, and A. Sivasubramaniam. Mechanisms for bounding vulnerabilities of processor structures. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 506–515, 2007.

[191] V. Sridharan and D. R. Kaeli. Using hardware vulnerability factors to enhance AVF analysis. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 461–472, 2010.

[192] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi. Feng shui of supercomputer memory positional effects in dram and sram faults. In *Proceedings of Supercomputing: the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2013.

[193] J. Srinivasan, S. V. Adve, P. Bose, and J. Rivers. The case for lifetime reliability-aware microprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 276–287, June 2004.

[194] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 177–186, 2004.

[195] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 107–119, Oct. 2004.

[196] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh. Criticality-based optimizations for efficient load processing. In *Proceedings of the 15th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 419–430, 2009.

[197] A. Subramaniyan, S. Rehman, M. Shafique, A. Kumar, and J. Henke. Soft error-aware architectural exploration for designing reliability adaptive cache hierarchies in multi-cores. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, pages 37–42, 2017.

[198] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 257–268, 2000.

[199] K. Swaminathan, N. Chandramoorthy, C. Cher, R. Bertran, A. Buyuktosunoglu, and P. Bose. BRAVO: Balanced reliability-aware voltage optimization. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 97–108, 2017.

[200] H. Tabani, J. Arnau, J. Tubella, and A. Gonzalez. A novel register renaming technique for out-of-order processors. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 259–270, 2018.

[201] M. Taram, A. Venkat, and D. Tullsen. Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, pages 624–637, 2018.

[202] K. A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean. Clairvoyance: Look-ahead compile-time scheduling. In *Proceedings of the International Conference on Code Generation and Optimization (CGO)*, pages 171–184, 2017.

[203] K. A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras. SWOOP: Software-hardware co-design for non-speculative, execute-ahead, in-order cores. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 328–343, 2018.

[204] T. K. Tsai, R. K. Iyer, and D. Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, pages 314–323, 1996.

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

[206] R. Uma, V. Vijayan, M. Mohanapriya, and S. Paul. Area, delay and power comparison of adder topologies. *International Journal of VLSI design & Communication Systems (VLSICS)*, 3(1), 2012.

[207] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE). In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 213–224, June 2012.

[208] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 177–188, 2013.

[209] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*, pages 87–98, 2002.

[210] K. R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 516–527, 2007.

[211] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Hybrid checkpointing for MPI jobs in HPC environments. In *Proceedings of the 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 524–533, 2010.

[212] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 61–70, 2004.

[213] S. Wang, J. Hu, and S. G. Ziavras. On the characterization and optimization of on-chip cache reliability against soft errors. *IEEE Transactions on Computer*, 58(9):1171–1184, Sept. 2009.

[214] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 264–275, 2004.

[215] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, pages 222–233, 2005.

[216] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In *Proceedings of the 15th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 79–90, 2009.

[217] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. In *2001 International Conference on Dependable Systems and Networks (DSN)*, pages 421–430, 2001.

[218] G. Yalcin, O. S. Unsal, A. Cristal, and M. Valero. FIMSIM: A fault injection infrastructure for microarchitectural simulators. In *Proceedings of the 29th International Conference on Computer Design (ICCD)*, pages 431–432, 2011.

[219] X. Yu, C. J. Hughes, N. Satish, and S. Devadas. Imp: Indirect memory prefetcher. In *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO)*, pages 178–190, 2015.

[220] H. Zhou. Dual-core execution: Building a highly scalable single-thread instruction window. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 231–242, 2005.

[221] Y. Zhu, M. Halpern, and V. J. Reddi. Event-based scheduling for energy-efficient QoS (eQoS) in mobile web applications. In *21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 137–149, 2015.

[222] J. Ziegler, H. Curtis, H. Muhlfeld, C. Montrose, B. Chin, M. Nicewicz, C. Russell, W. Wang, L. Freeman, P. Hosier, L. LaFave, J. Walsh, J. Orro, G. Unger, J. Ross, T. O'Gorman, B. Messina, T. Sullivan, A. Sykes, H. Yourke, T. Enger, V. Tolat, T. Scott, A. Taber, R. Sussman, W. Klein, and C. Wahaus. IBM experiments in soft fails in computer electronics. *IBM Journal of Research and Development*, pages 3–18, 1996.

[223] J. F. Ziegler and W. A. Lanford. The effect of cosmic rays on computer memories. *Science*, 206(776), 1979.

[224] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA)*, pages 2–13, July 2001.

[225] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 172–181, June 2000.