NORTHWESTERN UNIVERSITY

Exploring Program Locality for Efficient Online Fault Detection

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Engineering

By

Feng Lu

Evanston, Illinois

June 2020

Abstract

As technology scales down, challenges in fabrication, thermal stress, and in-field degradation have put the reliability of processors at risk. Among different fault types, transient faults manifest themselves frequently due to high chip density, aggressive voltage scaling, and high clock frequency. Some dependable processor architectures have been proposed to counter these faults, by integrating various online solutions for error detection and recovery. Recently proposed techniques, including perturbation-based fault screening and ternary content-addressable memory anomaly detection, exploit locality in memory addresses and values for transient fault detection. Their fault coverage comes at a high energy cost and numerous false positives.

This dissertation addresses the fault detector's efficiency problem. We exploit the locality in memory strides, instead of references, to reduce the amount of data needed for fault detection. We propose using Bloom filters to store the hashed form of memory patterns, instead of their original form in TCAM to reduce the hardware and energy cost. We also explore program phase-level locality and propose a framework to customize the fault detectors for the current phase. Additionally, we present the detector design for both the processor backend and the frontend to achieve high fault coverage (80%) at a low false positive rate (<1%). This greatly improves the resilience of the processor to soft-errors while limiting the energy and performance overheads.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Russ Joseph, for his continuous support and guidance throughout my PhD study. He taught me how to conduct research and present the results effectively, and always provided insightful technical feedback and advice. Without his motivation, patience, and immense knowledge, this work would not be near where it is today.

I also want to thank Professor Nikos Hardavellas and Professor Gokhan Memik for reviewing this dissertation. Their comments regarding the limitations of the fault detector were critical for making this dissertation complete.

Part of my PhD study was on efficient parameter variation sampling, which also appears in this dissertation. I want to thank Dr. Goce Trajcevski for his input on low-discrepancy sequences, which was fundamental to some of the key results in this project.

Toward the end of my PhD years, I had to request for an extension to finish the writing of this dissertation while accommodating my family needs. I want to thank Professor Joseph and Professor Hardavellas for their tremendous support, and Kristine Emrich for her sage advice. This granted me time to shape and polish this dissertation and present the results of my research.

Finally, I want to thank the Electrical and Computer Engineering Department for the infrastructure and computing resources they made available to researchers. Financial-wise, I am lucky to have been supported by a Murphy Fellowship. Also, this work is supported in part by the National Science Foundation (NSF), with grants CCF-1116610, CAREER CCF-0644332, CNS-0720820, and CNS-0910952.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

Modern processors have become more susceptible to transient faults as technology scales down. Transient faults arise mostly from cosmic rays and alpha particles from the packaging [37], where enough charges are collected to invert the state of a logical device. It is widely known for its significant impact on DRAM reliability [38]. While RAM structures are generally covered by parity and error-correcting code (ECC), the processor core is at risk and needs extra measures for fault protection.

Like discussed in existing works [16, 17, 69], transient faults materialize as single bit flips. Within the processor core, a bit flip in stale data or code is masked, and no error is introduced to program execution. Others can manifest themselves prominently by causing program crashes. The remaining faults flip a bit in active data or code and render incorrect program execution path and possibly corrupted computational results without ever being detected. These faults are called silent data corruptions (SDC) and are the focus of this dissertation.

SDCs can be detected and corrected by extreme redundancy like TMR [69]. Recent works on SDC detection, including Perturbation-Based Fault Screening (PBFS) [16] and FaultHound [17], aim to reduce the fault detection and recovery cost by studying program locality. PBFS checks memory data value ranges, data bit invariants, and matching of recent data values, and considers a violation to these locality measures a fault. While PBFS can achieve high fault coverage, the performance cost can be extremely high (e.g. 100%). FaultHound maintains a ternary content addressable memory (TCAM) to store dynamic data bit invariance and checks for anomalies by searching the TCAM. It also reduced recovery time through a modified replay mechanism. While FaultHound reduced performance cost compared with PBFS, the TCAM implementation is still expensive, and poses a tough trade-off between fault coverage and energy overhead.

This dissertation is an effort to explore program locality at new dimensions and design a fault detection mechanism that increases fault coverage and efficiency. The main contributions of this work include:

- We use memory stride instead of address when checking for locality. PBFS and FaultHound both checks bit-level invariance of references. While references change frequently and the number of unique references can be too large to handle, we observe that strides can maintain a more stable pattern. Furthermore, we identify the main reason why a stride stream loses its pattern, which is that some static instructions, when put in the same stream, disrupts each other's patterns. We present an instruction partitioning mechanism to divide the static instructions into separate partitions. The partial-global stride stream from each partition will have better patterns than the global stream, and the partitioned design requires much smaller storage. The strides, when combined with memory values, are used to represent the program locality.
- We propose using Bloom filters to store and search for data patterns, instead of TCAM. Given its constant search time, TCAM has been widely used in switches and routers for route lookup and packet classification [39, 40], and in search engines for clustering [41]. FaultHound uses TCAM for data clustering. However, TCAM incurs high energy

overhead due to its complex implementation. We achieve data clustering using the instruction partitioning. When storing the stride and value patterns, we use Bloom filters with a parallel Bloom filter implementation [25] which achieves both low energy overhead and fast membership testing and insertion.

- We explore the locality at program phase level to customize the detector for the current phase. Program phases have been studied for multi-configurational hardware [7], and it has been proved that a program execution goes through a series of phases, while some phases repeat a previously-seen phase. For our fault detector, we monitor phase transition. Once an unseen phase starts, we dedicate a learning period to generate the instruction partitions and train the stride and value Bloom filters. We store the partition information and Bloom filter contents for up to 20 unique phases in L3. When a phase repeats a known phase, the content from the known phase is reused.
- We present the detector design for both backend and the frontend faults. The detector's stride and value Bloom filters raise signals when a membership test fails. If the signal is due to a back-end fault, we use a delay buffer to do a light-weight replay at the backend of the out-of-order processor to recover, same as FaultHound. A separate filter is needed to distinguish a front-end fault and a detector false positive. The filter consists of a set of squash state machines that are assigned to segments of the parallel Bloom filters. We present the detailed design of the detector and front-end fault filter.
- We conduct comprehensive fault injection and perform detailed evaluation on fault coverage, false positive rate, and cost on performance, area, and energy.

The rest of the dissertation is organized as follows. Chapter 2 discusses related works on fault detection and further explain the motivation for this dissertation. Chapter 3 describes the evaluation methodology with detailed hardware configuration, simulation methods, workload, and metrics. Chapter 4 gives a high-level view of the fault detection design and explains the program phase-aware framework. Chapter 5 describes the rationale behind using memory strides and values and presents the design and implementation of an instruction partitioning mechanism. Chapter 6 discusses the detector design, especially the Bloom filter design and implementation in detail, and presents the experimental results for back-end fault detection. Chapter 7 describes the front-end fault filter design and presents its experimental results. Chapter 8 includes another part of my PhD studies on efficient parameter variation sampling by using low-discrepancy sequences. Chapter 9 draws conclusion for this dissertation.

Chapter 2 Related Work

Since the rise of processor dependability challenges due to physical faults, architects have explored an abundance of fault tolerance solutions to protect the processor core. In this chapter, we discuss the existing works on fault detection and explains the motivation behind this dissertation. In Section 2.1, we introduce key concepts in fault tolerance and discuss taxonomy of representative fault tolerant architectures. In Section 2.2, we explain the recent works on detecting silent data corruptions (SDC) that are the most related to this dissertation.

2.1. Fault-Tolerant Architectures

Before we discuss detailed fault detection mechanisms, it is important to introduce the key concepts in fault tolerant architecture. First, we divide the fault types into transient faults and permanent faults. A fault-tolerant architecture can cover one of the two types, or both. Second, different types of redundancy can be added for fault detection and correction, including physical, temporal, and information redundancy [42]. The type and level of redundancy determines not only the fault coverage, but also performance and energy cost. Lastly, fault recovery mechanisms can be divided into two major categories, forward recovery that does not require reverting to a previous state, and backward recovery that restores the state of the system to a pre-fault checkpoint. The recovery choice helps make trade-offs between performance cost and energy cost.

A taxonomy of fault-tolerant architectures has been proposed in [43]. Here, we use a similar taxonomy to discuss a few categories of existing works that can protects the processor against transient faults.

N-Modular Redundancy

Dual-Modular Redundancy (DMR) or N-Modular Redundancy (NMR) [1] uses one or more redundant module to detect 100% architectural errors. DMR needs backward recovery to correct a fault, while triple modular redundancy (TMR) [69] and above can make forward progress even when a fault is detected. Challenges with NMR include the significant operating overhead of the redundant hardware, the communication cost between the mirrored modules, and the detection latency which may be compromised to keep a low communication frequency. Fingerprinting [6] proposes to hash the architectural state updates from a checkpoint interval into a fingerprint and detect errors by comparing the fingerprints from the mirrored modules, which reduces some communication cost and detection latency. Sampling + DMR [14] reduces the excessive cost by running in DMR mode in a small percentage of time but is limited since it detects permanent faults only.

Anomaly Detection

These approaches watch for anomalous behaviors or symptoms at hardware and software level to detect faults. SWAT [3] and mSWAT [4] monitors software-level symptoms, such as fatal hardware traps, hangs, high OS activity, and kernel panics, and use them as indications of hardware faults. Trace-based replay on these systems are proposed to diagnose permanent faults [13]. SWAT covers 95% faults within 100K cycles. While SWAT has almost no hardware and performance cost, its high detection latency makes the recovery expensive. ReStore [11] detects transient errors by monitoring microarchitectural anomalous events including exceptions, branch mispredictions, and cache misses, and rolling back to a previous checkpoint upon these events. Pattern change in the outcome of speculative structures can also be used to detect faults [15]. These approaches cannot cover a high percentage of fault or will incur high recovery cost.

Dynamic Verification

With careful hardware design, this category of fault detectors performs checks on the fidelity of control flow and data flow and detect faults upon a violation. Data flow violation can also be detected by monitoring invariants and verifying the value of critical variables. Range-based likely program invariants [9] can be identified during a separate pass of offline run, and faults can be detected by verifying their validity during production run. Another work uses compiler-based static analysis to identify, for each function, variables that have large fan-out, and checks at run time the correctness of their values using backward derivation [12]. Control flow can be checked by adding assertions into a program and detect faults through basic block parity check during program execution [8]. The above checkers all incur high performance and energy costs. Argus [2] reduced the performance penalty and increased fault coverage by implementing hardware checkers for control flow, computation and data flow and monitors all their activities. However, the 17% hardware overhead is still significant.

2.2. Value Locality-Based SDC Detection

Recent works have shown good coverage on SDCs by looking for value locality anomalies. Popular approaches, such as Perturbation-Based Fault Screening [16] (PBFS) and FaultHound

[17], check microarchitectural states related to load and store instructions, in order to detect any faulty behavior of the processor.

Perturbation-Based Fault Screening

This work proposes to establish a profile of program behavior and detect faults by looking for departures from the established behavior. Such departure is called a *perturbation*. PBFS presented five fault screeners and implemented one that is the most realistic: the invariance-based screener. This screener maintains bitmasks associated with each static load or store instruction. Each bitmask represents the set of all bits that are invariant since the first data value of this static instruction. For each dynamic load/store data value, a lookup is performed in the PC-indexed bitmask table. If the new value does not match the filter on all unchanging bit positions, a perturbation is reported, and the bitmask is updated to mark this bit position as variant to avoid future false positives.

On every perturbation trigger, PBFS proposes to flush the pipeline to recover from a fault. With this approach, PBFS reports low false positive rate (e.g., 0.5%) with a fault coverage of about 30% on SDC. If a full rollback (e.g., 100 instructions) is allowed, the coverage rate of SDC becomes over 70% [16]. However, a 0.5% false positive rate implies 50% energy cost in such scenarios (assuming a baseline cycles per instruction of 2).

This design has a few limitations. First, each time a perturbation occurs, the bitmask is updated to mark a new variant bit. Thus, each bit can only detect one perturbation after the periodic clear which indicates loss in fault coverage. Second, PBFS does not distinguish a false positive and a fault. This exposes the intrinsic tension between fault coverage and performance/energy penalty.

Lower-penalty recovery approaches (e.g., pipeline flush) result in loss of coverage, while higher-coverage approaches incurring full rollbacks can be too expensive to trigger at every perturbation.

FaultHound

FaultHound checks load and store address and value, same as PBFS. Address and value are handled by separate filters. Since the PC-indexed value checker in PBFS may unnecessarily separate nearby similar values across multiple table entries, FaultHound proposes to use ternary content-addressable memory to cluster similar values into one filter. Unlike the sticky bit counters in PBFS, the state and bit value in a FaultHound filter are both updated upon a trigger, to adapt the filters to the most recent value. To further reduce false positives triggered by delinquent bits, a second-level filter is implemented to associate a biased state machine to each bit position. The biased state machine ensures that the bit position only triggers a fault or false positive after 7 consecutive no-triggers. To reduce the recovery penalty from a fault, FaultHound adds a delay buffer (e.g., 7 instructions) to an out-of-order issue processor's replay logic and triggers a backend replay instead of full rollback to correct faults on the backend. To distinguish front-end faults that cannot be corrected by replay, a biased state machine per filter is implemented to differentiate rename faults from false positives. A full rollback (~100 instructions) is only triggered when a front-end fault is detected.

FaultHound's TCAM implementation improves clustering efficiency while maintaining constant search time for each value, which leads to 75% fault coverage at 3% false positive rate. However, this approach still has its limitations. First, the TCAM hardware is expensive in area and energy, which limits the number of filters that can be used and indicates possible coverage loss. Second, the memory addresses of a program can have many permutations, which increases noise level in the filters. An alternative using memory stride may provide better patterns and improve filter efficiency. Lastly, the filters are updated on a per-value basis. This can introduce unnecessary false positives during an execution phase transition.

2.3. Program Locality Characterization

One important concept underpinning our work is notion of inherent time redundancy (ITR) [5], which means that the same instructions are executed repeatedly by a program within a short interval, and can be interpreted as the program re-executes itself in a time redundant way. The original ITR [5] work detects faults by recording static instruction trace signatures and checking against existing signatures. A related concept, working set signature, has been applied to detect program phase change and tune to the optimal hardware configuration in a multi-configuration system [7].

Program locality can be used to detect software bugs like performance problems. Toddler [10] identifies repetitive and similar memory-access patterns over time of execution and exposes performance bugs induced by redundant loops or inefficient loops.

Chapter 3 Evaluation Methodology

In this chapter, we present the methodology for evaluating the detector. In the following chapters, all experimental results are based on the experiment configurations described here.

This chapter is organized as follows. In Section 3.1, we explain the detailed configuration of the simulated processor. In Section 3.2, we discuss the simulators we used and simulation methods. Section 3.3 presents the workloads that we used for evaluating the system. In Section 3.4, we explain the metrics that we use in the following chapters to assess the detector.

3.1. Processor Configuration

We simulate the X86 instruction set on an out-of-order-issue, high performance processor with a cache hierarchy like Intel's Sandy Bridge [33] processor. The fault detection mechanism is implemented within each core. Since the detector does not rely on or impact any cross-core functionalities, we opt to simulate the processor with only one core. The detailed hardware parameters of the processor are presented in **Error! Reference source not found.**. The access latencies of the on-chip cache are calculated based on published performance evaluation of the Sandy Bridge processor [32].

Additional hardware is added to the processor core for fault detection purposes, which includes Bloom filters and other caches implemented in SRAM at L1 level. For clarity, these structures will be explained in detail in the following chapters, and their parameters will be provided when presenting the fault detector evaluations.

Base frequency (GHz)	2.6
Technology	32 nm
Fetch, Decode, Issue, Dispatch, Commit width	4
IntALU, IntMulDiv, FPALU, FPMulDiv per core	6, 2, 4, 2
Issue Queue size	32
Re-order Buffer	168
Load Buffers	64
Store Buffers	36
L1 cache	32 KB (I)+32 KB(D), 8-way set associative, 3-cycle hit latency
L2 cache	256KB, 8-way set associative, 9-cycle hit latency
L3 cache	2.5MB, 16-way set associative, 17-cycle hit latency

Table 3-1: Hardware parameters of the simulated processor

3.2. Simulation Methodology

We use the gem5 simulator [34] to conduct fault injection experiments and evaluate fault detection structures. Gem5's system emulation mode is used to perform timing-accurate simulation. We use McPAT [35] to estimate the processor core area and power, and CACTI [36] for the fault detection structure's timing, area, and power.

The performance overhead is generated with timing information from gem5 baseline simulations, CACTI timing results of the fault detection structures, and the false positive rates of the fault detectors. The energy overhead is generated with the power numbers of the processor core, all on-chip caches, the fault detection structures, and the performance overhead.

3.3. Workloads

We chose a set of benchmarks from SPEC CPU® 2006 [29] suites, as shown in Table 3-2. Out of the 12 benchmarks used, 6 are from SPECint® 2006, and the other 6 SPECfp® 2006. Our fault detection mechanism mainly exploits locality in memory access patterns and program phases. The 12 selected benchmarks exhibit a wide range of cache miss rate [70], and have distinct characteristics in program phase length, phase transition, and repetition. These properties make the 12 benchmarks a good mix for evaluating our fault tolerant architecture.

CINT	**CFP**
401.bzip2	410.bwaves
403.gcc	436.cactusADM
429.mcf	444.namd
456.hmmer	454.calculix
458.sjeng	465.tonto
462.libquantum	470.lbm

Table 3-2: Benchmarks

For each benchmark, we select one representative SimPoint [18] of 100 million instructions. With gem5's checkpointing functionality, we fast-forward simulations to this SimPoint's checkpoint and simulate only the 100 million instructions within the SimPoint.

3.4. Metrics

When presenting the evaluation results, we focus mainly on the detector's fault coverage, false positive rate, and impact on hardware area, system performance, and energy consumption.

Since we use Bloom filters in our fault detectors, it is important to not confuse the false positive rate of a Bloom filter and that of the detector. A false positive from a Bloom filter means it falsely reports true when testing a non-existent element's membership. A false positive from the fault detector means the detector falsely raises a signal for an error that does not exist.

Chapter 4 A High-Level Description of the Detector

This chapter describes the high-level design of our fault detector. Since most transient faults are either masked or result in program crash and are thus detected [16, 17, 69], we focus on detecting the remaining faults - silent data corruptions (SDC) that render incorrect program execution path and possibly corrupted computational results without ever being detected. The goal for our detector is to have a high detection rate on SDCs while keeping the cost low. To reach this goal, we utilize program localities. The locality is stored by the detector and used for detecting faults. The detailed design choices for the detector will be discussed in Chapter 6. In this chapter, we focus on how the different components of the fault detector are orchestrated and present the workflow of detecting a fault.

In order to explain our high-level design of the detector, we structure this chapter as follows:

- In Section 4.1, we give an overview of the fault detection's key components and a fault detection workflow.
- In Section 4.2, we explain the use of program execution phases, which is vital to the efficiency of the detector. Identifying a program execution phase transition can help limit the scope of the detector, while identifying a phase repetition enables reuse of known

locality information to save training time. The concepts of a learning period, unique phases, phase transitions, as well as phase repetitions, are introduced in this section.

- In Section 4.3, we discuss how a detector signal is analyzed to distinguish a back-end fault, a front-end fault, and a false positive. This is essential in choosing the correct mechanism for recovering from a fault.
- Finally, in Section 4.4, we explain how an analyzed signal is handled.

4.1. Design and Workflow Overview

4.1.1. Key Components of the Detector Design

Before we dive into the workflow of the detector, there are three key aspects of the detector that need to be introduced: What information from the program do we use to represent program locality, what structure is used to store the locality and detect fault, and how the fault detection algorithm is designed to make the detector efficient.

Capturing Program Locality

As described in Chapter 3, the faults we are trying to detect are transient faults that result in silent data corruption (SDC). These faults do not affect the program execution in a catastrophic way, but only change some data values that produce incorrect results. Thus, neither micro-architectural state [11] nor software-level events [3] can distinguish a program's faulty execution from its correct behavior. Instead, established program locality can help detect data anomalies and predict data corruption. Works including PBFS [16] and FaultHound [17] have used the effective address and load/store value of memory instructions to represent program locality. In our work, we also utilize memory instruction information to represent program locality. Beside the load/store value, the stride of the effective addresses is captured instead of the addresses themselves. The benefit of using the stride will be explained in Chapter 6. In this chapter, we will use the abstract term *program locality information* as an abbreviation for the memory strides and values that are used in the detector.

Using Bloom Filters for Fault Detection

A Bloom filter is a data structure that efficiently stores a set of elements and quickly tests whether a given element is a member of the set. In our detector, we use the membership of a Bloom filter to detect faults. During a training period, memory strides and values are stored in the Bloom filter. After training, the Bloom filter tests incoming memory strides and values for set membership. If an item is decided to not be a part of the known set, the detector believes it has detected a fault. The design of the Bloom filter will be explained in detail in Chapter 6.

Phase-Aware Fault Detection Algorithm

Known as time-varying behavior [19], a program execution goes through a number of phases. Program locality can change as phases shift. Thus, it is important that our fault detection algorithm monitors phase transitions and adjusts the detector accordingly.

There are three major advantages for our fault detector to be aware of program phases:

1. The detector can be better built to suit the program locality of the current phase. This way, the scope of the detector is limited, which makes the fault detection more effective and the detector more cost-efficient.

2. The detector can monitor phase transitions. If a phase transition occurs, i.e. the program enters a phase that is different from the current phase, the detector can determine that it needs to be cleared and re-trained for the next phase. Otherwise, the detector can continue to work.
3. The detector can monitor phase repetitions. As will be discussed in Section 4.2.2, it is quite common for a phase to repeat at different points of a program execution. The detector content of a phase can be saved and possibly reused if the detector determines there is a repetition of this phase. This can reduce the cost of training since the time for re-loading the Bloom filters are much smaller than the time required to train the filters.

The way that our detector monitors phase transitions, phase repetitions, as well as how the detector is trained or reloaded, will be discussed in detail in Section 4.2.

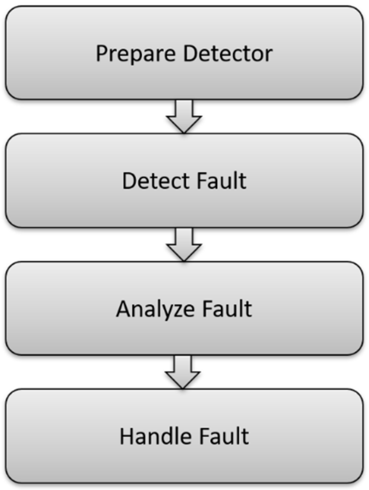


Figure 4-1: Overview of the fault detection workflow

4.1.2. The Workflow

When a fault is present in a system, an end-to-end workflow is needed to detect the fault and help the system recover from the fault. An overview of our fault detection workflow is presented in Figure 4-1.

First, the detector needs to be prepared before predicting faults as shown in Figure 4-2. To capture program locality and use it for fault detection, our detector needs to be trained when a fresh phase starts, which we refer to as Intra-Phase Preparation. In addition, when a phase transition occurs and if the new phase is a repetition of a past phase, the detector reloads the content of the original phase, and we refer to this as Inter-Phase Preparation. The details will be discussed in Section 4.2.

Then, the detector is ready to predict a fault. At this stage, the outcome of our detector is binary. If an incoming data pattern violates the existing program locality information, the detector raises a signal; otherwise, the detector does not raise a signal and continues to test the next incoming data pattern. The design details of the fault detector will be discussed in Chapter 6. For this current chapter, this part of the detector can remain a black box that outputs a “fault” or “no fault”.

Once the detector predicts a fault, the fault is analyzed, to the detector’s best knowledge, to see whether it is a real fault and if so, which type of fault it is. Thus, this fault analyzing stage signals “false positive”, “back-end fault”, or “front-end fault”. Section 4.3 describes how a fault is analyzed by our detector.

Finally, the outcome of the fault analyzing stage gets handled accordingly. If a false positive is detected, our fault detector updates its program locality information to include the pattern in the false positive. If a real fault is detected, depending on whether it is back-end or front-end, the detector initiates corresponding recovery mechanism to get the system back to a correct state. Section 4.4 describes how different faults are handled in recovery.

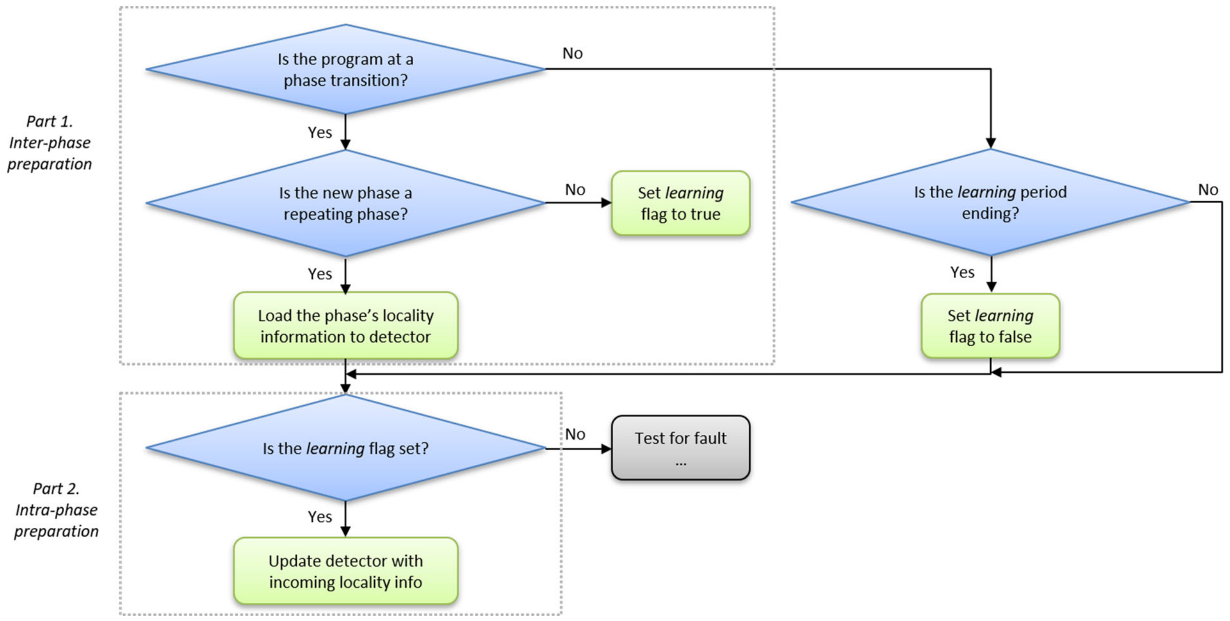

Part 1.
Inter-phase
preparation
Is the program at a phase transition?
No
Yes
Is the new phase a repeating phase?
No
Set learning flag to true
Yes
Load the phase's locality information to detector
Is the learning period ending?
No
Yes
Set learning flag to false
Part 2.
Intra-phase
preparation
Is the learning flag set?
No
Test for fault ...
Yes
Update detector with incoming locality info

Figure 4-2: Detector content preparation

4.2. Preparing the Detector with Program Locality

For the detector to identify anomalies in program locality, it needs to be prepared before performing its fault detection functionality. The preparation is analogous to warming up any cache – the program locality information needs to be loaded into the detector.

4.2.1. Intra-Phase Preparation

For a detector to be effective, it needs to be trained with some program locality information, and that information is used for detecting anomalies afterwards. On an extreme end, the program execution can be viewed as one big phase, and the warming up is done only once. However, for the detector to be more effective, phases at a finer time granularity should be identified, and the detector should be trained at the beginning of each phase.

We call the training period within a phase a *Learning Period*. As shown in Figure 4-3, the program execution can be composed of multiple phases, and each phase has its own learning period. For simplicity, each learning period is of a fixed length. The length of the learning period has a significant impact on fault detection rate and performance cost. We will discuss more of the trade-offs on the learning period in Chapter 6.

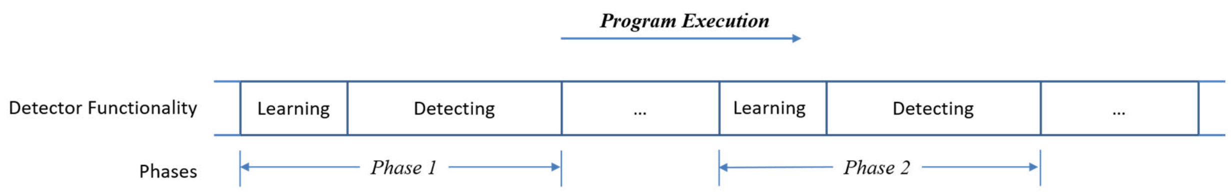


Figure 4-3: Learning period and detection period within a phase

The program locality information gained through the learning period will remain in the detector until a phase transition occurs. As will be discussed in Chapter 6, our detector consists of multiple Bloom filters. Each Bloom filter is a bit vector. During the learning period, a selection of memory values and effective address strides are added to the Bloom filters. When the learning period reaches its end, for the rest of the phase, incoming memory values and strides are tested by the Bloom filters to predict a fault. Once a phase transition is detected, the Bloom filters' content cannot be used any more, and the detector needs to determine whether to train or reload existing content for the next phase.

4.2.2. Inter-Phase Preparation

The detector content can be customized for each individual phase. The customization follows two rules:

1. The detector content cannot be shared by two phases that are dissimilar.
2. The detector content can be reused by two phases that are similar.

The detector identifies when the program execution enters a phase that is dissimilar to the current phase, which is called a *phase transition*. Detecting a phase transition is essential for keeping the detector content sensitive to the current phase. However, clearing the detector content and restart a learning period can be costly. If redundancy between phases can be detected, the detector content can be reused when a phase repeats itself. Thus, the detector also determines whether the new phase should be considered a repetition of a previously seen phase. In this section, we will explain how phase transitions and repetitions are detected, and how the detector functions to take advantage of them.

Program Execution Phases in Other Works

The concept of program execution phases has been studied to help optimize hardware reconfiguration [7]. The need for tuning hardware proves that different program phases have distinct characteristics, e.g. instructions per cycle, miss rate, etc. This implies that the program phases are drastically different in memory accesses and data values. Thus, to achieve high efficiency, the fault detector should adapt to the current program execution phase.

On a side note, instruction locality has been explored to detect transient faults by Inherent Timing Redundancy [5]. This work is orthogonal to our fault detection work.

Working set analysis [7] shows that program execution phase transition is highly predictable by the transition of the program's instruction working set. A compressed representation of instruction working set is monitored to detect working set changes, and thus detect program phase changes. We will explain how we adopt the working set signature approach for detecting a program execution phase.

Forming an Instruction Working Set Signature

A working set is bound to an interval of execution, or a *window*. The instruction working set is comprised of all the static instruction PCs within a window. The phases can be resolved at a granularity no finer than the window size. In this work, the window size is set to 100K instructions. Also, a series of non-overlapping windows are used instead of a sliding window.

Compression is crucial when storing working set information. An instruction working set containing n static instructions, if not compressed, can take $n \times 64$ bits in a 64-bit system. To achieve an efficient representation, we adopt the working set signature approach proposed in [7] to represent an instruction working set.

The working set signature is a vector of n bits. It is generated by mapping each instruction PC onto one bucket of a vector, as shown in Figure 4-4. First, instead of using raw instruction PCs, the working set elements are set to be of cache line granularity. For a cache line size of 2^b, the lowest b bits of the instruction PC are dropped and only the remaining m bits are used. Second, the highest m bits of a PC are compressed by a hash transformation. These m bits are hashed into a value k, which is a number between 0 and $n-1$. The k_{th} bit of the signature gets set to 1.

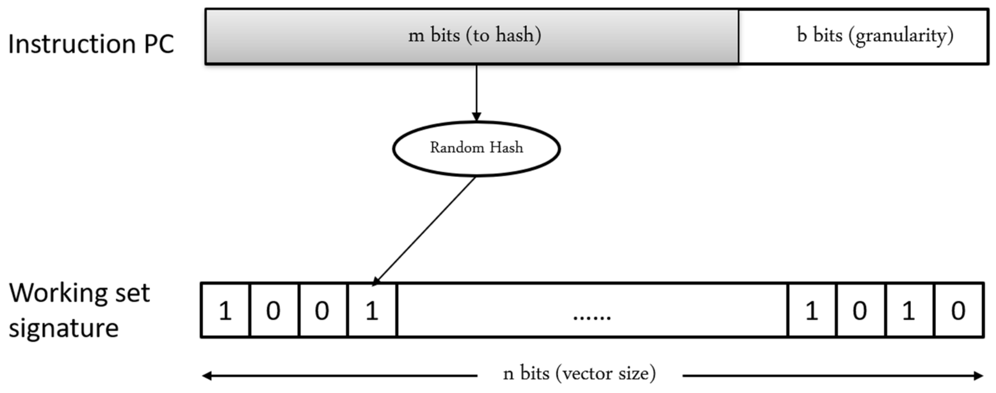


Figure 4-4: Working set signature generation

We made a few modifications to the working set signatures in our adoption. First, to reduce performance and power cost, only branch and memory instructions are used. This is different from the working set analysis work [7] where every committed instructions PC is used. Second, since the working set being monitored is smaller, the signature vector size can be much smaller as well. We use a signature size of 64-bit, in comparison to the 128-byte signature size in the working set analysis work. This size reduction significantly saves the amount of memory we need for detecting phase repetition, as will be discussed in the later section.

Putting it all together, for each non-overlapping window of 100k instructions, we form a 64-bit working set signature with a subset of PCs that are executed in this window. We use a cache line size of 64 bytes. This means b is set to 6 bits and we only use the highest 58 bits of the instruction PC. For each PC, the highest 58 bits are randomly hashed into a number between 0 and 63 to update the signature.

Phase Detection Architecture

We need to store a few different signatures for phase resolution, which is shown in Figure 4-5: A current signature, a previous signature, and a set of past unique signatures. These stored signatures are used for detecting phase transitions and repetitions.

The current signature is the signature that is being built within a window. It is updated by each instruction PC in the working set. This signature is stored in a dedicated memory location.

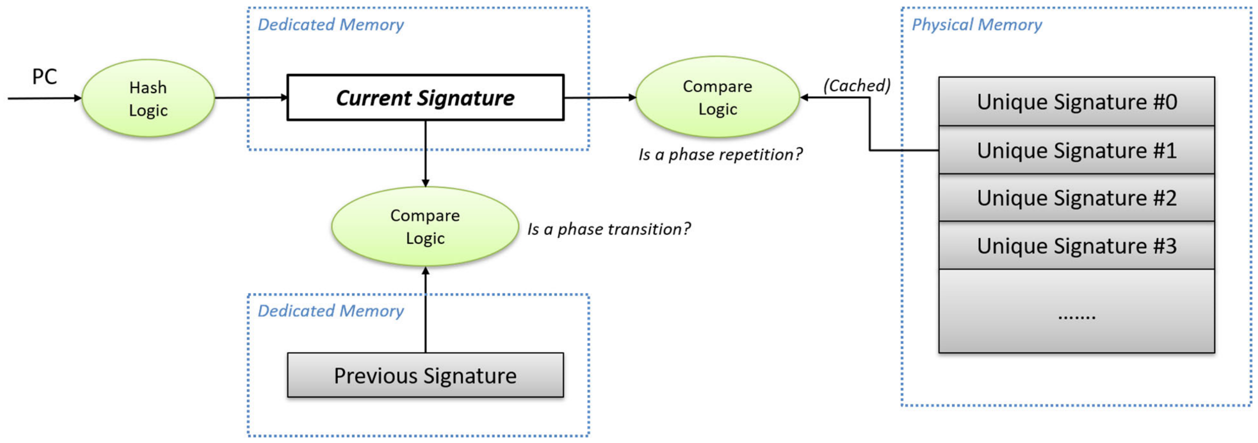


Figure 4-5: Working set signatures are stored in memory locations for phase resolution

The previous signature is used for detecting phase transition. Once a window ends, the current signature is finalized and is then compared to the previous signature for phase transition. If the compare logic determines there is not phase transition, the current windows is considered a continuation of the previous window. The previous signature is stored in a dedicated memory location like the current signature.

A set of unique signatures is used for detecting phase repetition. These signatures are stored in the physical memory. When the current signature finalizes and if a phase transition is detected, the unique signatures will be cached, and each is compared with the current signature to detect phase repetition. If the compare logic determines there exists a past unique signature that the current signature is repeating on, the current phase is considered a repetition of that past unique phase, and the detector can be warmed up using that unique phase's detector content.

The process of detecting phase transitions incurs some performance cost. The program execution needs to be stalled in order to perform these phase detection operations:

- Operation #1: Comparing the current signature against the previous signature at every window boundary.
- Operation #2: Loading the unique signatures to a near cache if a phase transition has been signaled by Operation #1.
- Operation #3: Once Operation #2 finishes, searching the unique signatures for one that qualifies the current signature as a repetition.

The cost of these operations depends on the signature size, the comparison logic, and the number of unique signatures that are stored. The signature size is 64 bits, as explained in the previous subsection. The comparison logic, which will be explained next, consists of an XOR operation, and OR operation, a one's count, and a division. The number of unique signatures, as will be explained in Figure 4-7, is 13 within one 1000 window SimPoint if we take the average of the twelve benchmarks we evaluated. The average phase length is 11.5 windows. Thus, for every window, one comparison will take place. For every 11.5 windows, 13 unique signatures need to be loaded to a near cache, and 14 comparison operations will take place to search for phase repetition. The 13 unique signatures take 2 cache lines, assuming a 64-byte cache line size, and thus incurs a stall of two memory loads. Each of the 14 comparison operations incur a stall of 4 ALU operations. Assuming a wide-issue pipeline, the memory load stall and the ALU stall together take ~200 CPU cycles per 11.5 window. This is still much more efficient than the ~200,000-instruction training time (as will be discussed in Chapter 5 and Chapter 6) if a phase repetition is not detected.

Detecting a Phase Transition

Detecting program phase transition can help the detector determine if it needs to be cleared and start re-learning.

Program phase change can be detected by comparing the working set signature of the current window to that of the previous window. The working set analysis work [7] defined the *relative working set distance*, to represent the similarity between two working sets. Once the distance exceeds some threshold, the program is considered entering a different phase.

Since each signature is essentially a set of bits, the relative working set distance is measured using the complement of Jaccard similarity. Jaccard similarity is the relative size of the intersection of the two sets, which is a commonly used metric in classification and clustering problems [71, 72]. The Jaccard similarity of sets S and T is $|S \cap T|/|S \cup T|$. The complement of Jaccard similarity measures the relative size of the difference of the two sets. Let the new signature be S_2 and its previous signature be S_1, the relative working set distance is defined as the ones count of exclusive OR relative to that of inclusive OR of the two signatures, which can be represented by the following formula, as introduced by [7]:

$$\Delta = \frac{|S_1 \oplus S_2|}{|S_1 + S_2|} .$$

A relative distance of 50% or higher is required to qualify for phase transition, i.e. Δ needs to be no less than 0.5.

Detecting a Repetition on a Unique Phase

Not only can a signature predict a program phase change, it can also predict phase repetition. The detector content of a phase can be reused when a repetition is detected.

We use a concept called *unique phase* to explain the phase repetition phenomenon. If the relative working set distance between the current signature and a signature that's previously seen is below some threshold, the current working set windows is considered a repetition of the previously seen window, and the two windows are considered two occurrences of the same program execution phase. The original phase that has been repeated is called a unique program execution phase, or a *unique phase*.

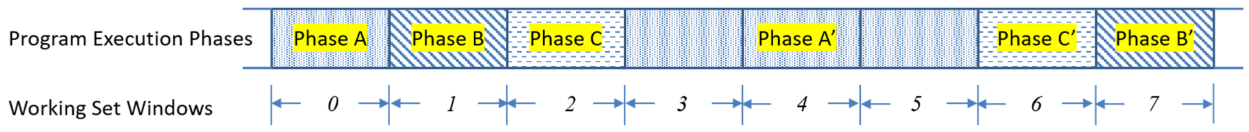


Figure 4-6: Unique phases and their repetitions

A unique phase can be repeated anywhere in the program execution and for length of any number of windows. As illustrated in Figure 4-6, three unique phases A, B, and C are repeated at different points of the program. Phase A is seen repeated for three windows lengths - windows 3, 4, and 5, while Phase B and Phase C are repeated one window each.

All unique phase signatures are saved for detecting phase repetitions. When a phase transition occurs, a repeated phase can be detected by comparing the current working set signature against every saved unique phase signature. If no match is found, the current signature is saved as a unique phase signature.

In our work, we set the threshold of relative working set distance to be $\leq 10\%$ for detecting a phase repetition. Based on the working set analysis work [7], there can be noise in measuring the similarity of two windows due to the misalignment of phase boundaries and window boundaries. Thus, a 90% similarity gives high confidence that the two windows belong in the same phase.

We studied the unique phases and their repetitions in a representative SimPoint [18] of twelve SPEC CPU2006 benchmarks, each SimPoint of length 100 million instructions. The results are shown in Figure 4-7. On average, there are 87 phase transitions within the SimPoint, within which 13 are unique phases and 74 are repetitive phases. This implies two things:

1. The average phase length is approximately 1.15 million instructions. Given the 100K-instruction working set window size, a phase transition occurs approximately every 11.5 working set windows. This means the phase is resolved at a granularity that's one order of magnitude longer than the working set window length. This is biased by two outliers: lbm and $sjeng$. These benchmarks executed only one unique program phase within the SimPoint. Without these two benchmarks, the average phase length becomes 500K

instructions, or 5 working set windows. This impacts the performance cost which will be discussed in Chapter 6.

2. On average, there are 13 unique working set signatures that need to be stored for detecting phase repetition, and these signatures only take 104 bytes of storage. Even for the benchmark with the highest number of unique phases, gcc, its 44 unique working set signatures only takes 352 bytes of storage, which is still feasible to be stored in the L1 cache.

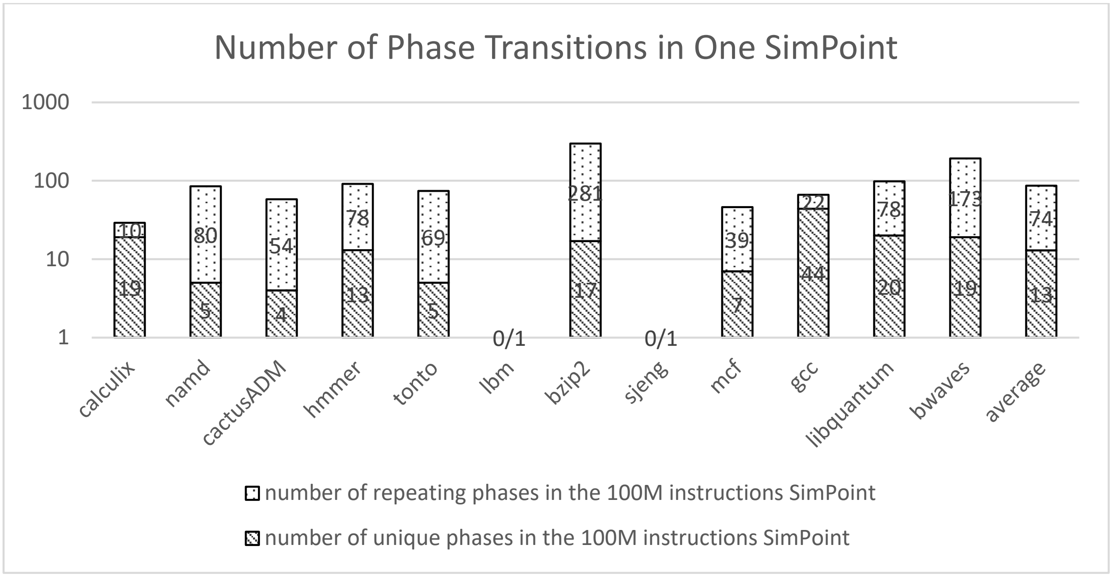


Figure 4-7: Number of phase transitions of in one SimPoint, divided into unique and repeating phases

Optimizing Detector Efficiency using Phase Information

Phase transition and repetition information can be used to improve our detector's efficiency. Essentially, a unique phase's detector content, or the program locality information of that phase, can be reused when a repetition phase is detected.

Each unique phase has its own learning period. Once a unique phase ends, its Bloom filter contents are stored in farther cache. When a phase repetition occurs, the learning period is replaced by loading the Bloom filter content of the unique phase that it repeats on.

The detector functionality in the case of a phase transition and phase repetition is described by the pseudo code in Figure 4-8.

The store and load of the Bloom filter contents depend on the sizes of the Bloom filters. This will be discussed in Chapter 6.

```
  if (is_learning == true) AND (phase_instruction_count ==
learn_period_length):
    is_learning = false
    storeBloomFilterContents(current_signature, PC)
  if (instruction_count % working_set_window_size == 0):  // should check for
phase transition
    change_rate = calculateChangeRate(current_signature, previous_signature)
// check for phase transition
    if change_rate >= phase_transition_threshold:
    (is_repeating_phase, original_phase) = isRepeatingPhase(current_sig,
existing_sigs)  // check for phase repetition
    if is_repeating_phase:
      loadBloomFilterContents(original_phase)
    else:
      is_learning = true  // start learning for a new unique phase
```

Figure 4-8: Pseudo code on how to detect phase transition and phase repetition and how the detector functions to utilize the phase information

4.3. When a Fault is Predicted: Analyzing a Fault

Before recovering from a fault, we need to categorize the fault in order to trigger the correct recovery mechanism. In this work, we differentiate the outcome of the fault detector into three categories: back-end fault, front-end fault, and false positive. Figure 4-9 Shows how these faults are analyzed into the three categories, as well as which mechanism each category uses to correct a fault.

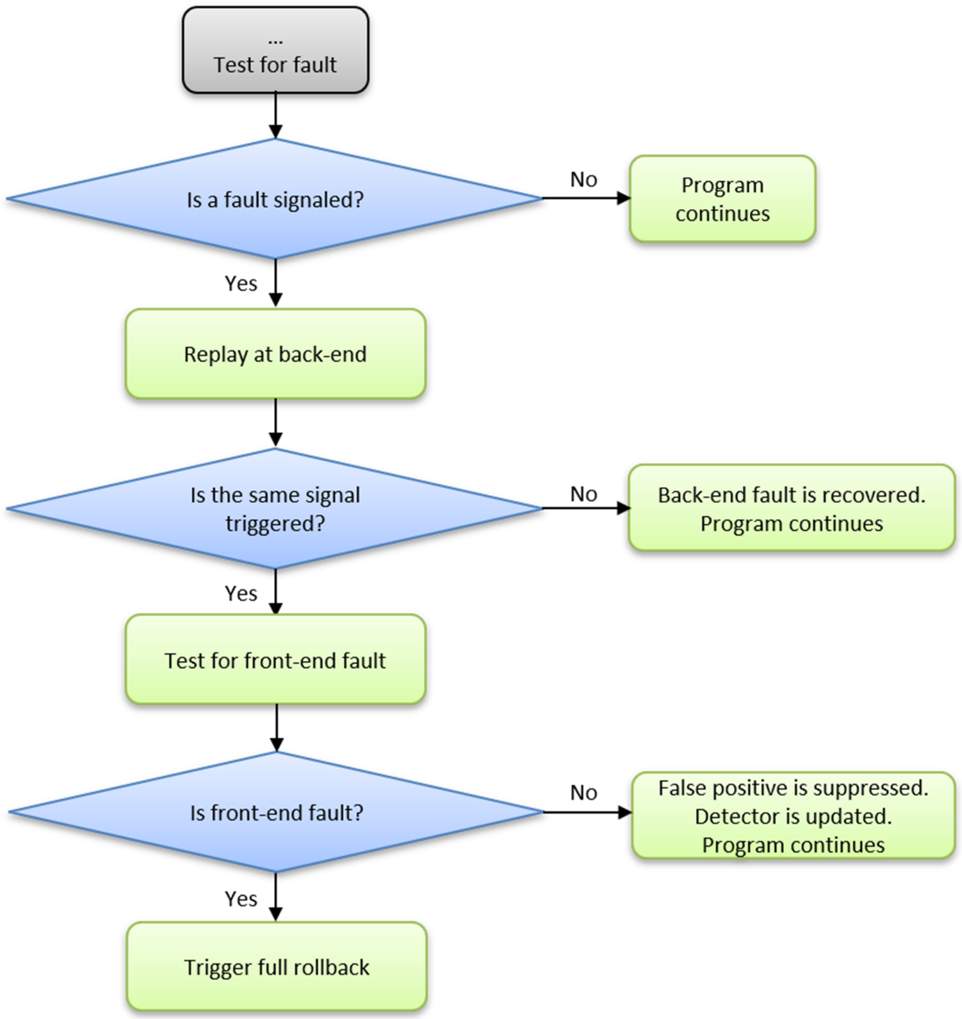


Figure 4-9: Workflow on analyzing fault categories and triggering corresponding recovery mechanisms

Our detector first raises signals in the back-end. Back-end includes all processor pipeline stages after the issue queue. Since the locality information we use includes memory access patterns and memory values, our detector can only raise signals when this information is available, which is after an instruction enters the pipeline's back end.

Our detector then determines whether the signaled fault is a back-end fault, or of a different type. As will be explained in Section 4.4, we use a back-end replay mechanism same as in FaultHound [17]. This replay is deemed final for back-end faults since most of them can be corrected. However, if the replay triggers a fault signal at the exact same instruction PC with the same memory data, we consider this fault not corrected by the replay, and thus not a back-end fault.

Our detector then distinguishes whether there is a front-end fault, or it just hit a false positive. We use a separate front-end fault detector, as will be described in Chapter 7, to determine this.

4.4. When a Fault is Confirmed: Handling a Fault

When a transient fault is predicted by the detector, we need to determine how the fault can be corrected.

Correcting a fault requires redundancy in hardware components, saved execution state, or repeated executions. To correct a transient fault, a common approach is to create periodical checkpoints and roll back to the previous saved checkpoint.

A full rollback incurs high cost on performance and power. FaultHound [17] avoided most of this overhead by utilizing the light-weight replay mechanism of the out-of-order-issue pipeline and adding a delay buffer to it. It argues that since most load and store instructions have dependencies only among nearby instructions, most of the faults can be corrected by squashing a short chain of instructions. This reduces the cost from a ~100 instruction rollback to a ~7 instruction replay. We use the same replay mechanism to recover back-end faults. If a fault is determined to be from the front-end, a full rollback will be triggered.

If a detector signal is analyzed to be neither a back-end fault nor a front-end fault, it is considered a false positive. In such a case, the signal will be suppressed, while the offending memory data is considered valid program locality information, which will be updated into the detector.

Chapter 5 The Input: Partial-Global Stride Sequence and Memory Value

This chapter describes the choice of partial-global memory stride sequences and values as the memory access pattern. We first discuss why the stride sequence can be more efficient than memory reference in capturing locality. Then, we introduce the concept of a partial-global stride, which has much fewer noises than a global stride. Next, we present a design and implementation of an instruction-partitioning mechanism that is critical in generating efficient partial-global strides. Finally, we explain the usage of both memory strides and values for fault detection which completes the picture of the detector input.

5.1. Reference versus Stride: Why is Stride Better?

There are many values from a memory access that can represent locality. Perturbation-Based Fault Screening [16] and FaultHound [17] both chose to use the memory address and memory data. PBFS first finds the range of each instruction's memory addresses or its accessed data. Then, it sees any out-of-range address or data as a perturbation. FaultHound, on the other hand, monitors only a selected number of bits of the memory address or data. It uses content-addressable memory to group them by value and uses a state machine on each group to detect anomalous values.

There is some temporal-spatial locality that has not been captured by these approaches, though. Memory stride, compared to a memory reference, can capture the memory access patterns in a sequential way. For example, in a loop, the memory references of all the instructions can be constantly changing, which can result in out-of-range addresses in PBFS, or unfitting values in FaultHound. In such cases, PBFS and FaultHound will either have high false positives overheads, or by marking these values as false positives, miss real faults since their databases have become more forgiving and insensitive.

In contrary, the memory access strides can stay the same from one iteration to the next, thus there are fewer permutations from the memory strides than from memory references. A stride-based detector will be more efficient than a reference-based detector.

A memory stride is defined as the difference of references or physical addresses between the current memory instruction and the previous memory instruction. Stride has been used by prefetchers [26-28] for predicting future references and prefetching them into caches before they are requested by the processor. Prefetcher designs have exploited locality in both local strides, meaning the strides generated by the same instruction, and global strides, meaning the strides generated by all load and store instructions.

Let a reference stream of n memory instructions be

$$\text{Equation 5-1: } \mathrm{RS_{global}} = \{\mathrm{R_0}, \mathrm{R_1}, \ldots, \mathrm{R_{n-1}}, \mathrm{R_n}\},$$

where R_0 is the predecessor of R_1 in time. A global memory stride S_k is defined as:

$$\text{Equation 5-2: } \mathrm{S_k} = \mathrm{R_k} - \mathrm{R_{k-1}}, 0 < \mathrm{k} \leq \mathrm{n}.$$

A sequence of global strides generated from the reference sequence can then be described as

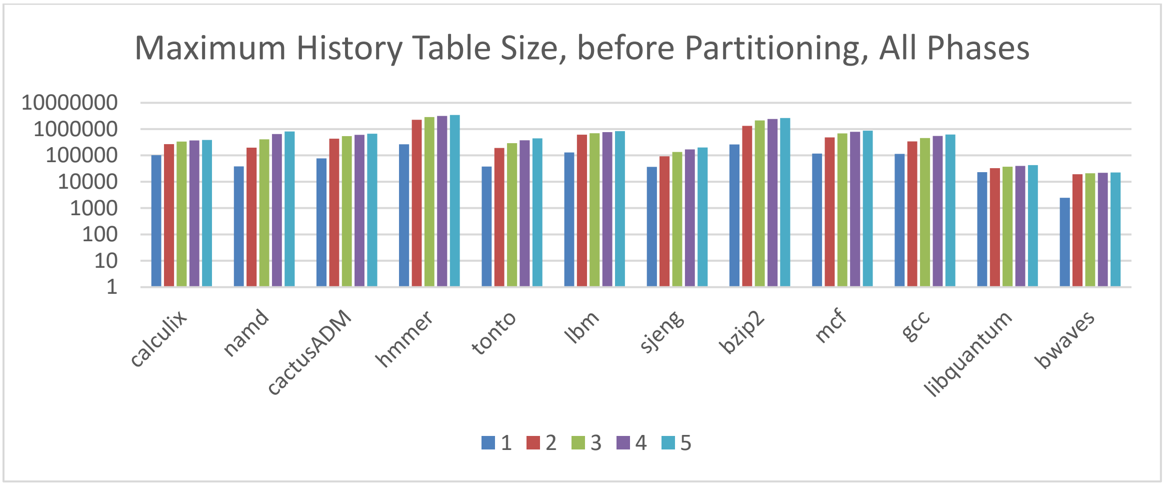
Equation 5-3: $SS_{global} = \{S_1, \ldots, S_{n-1}, S_n\}$.

To enhance the sensitivity of the detector, we also consider using a sequence of strides instead of one single stride as the memory patterns. A stride sequence can be more sensitive than a single stride.

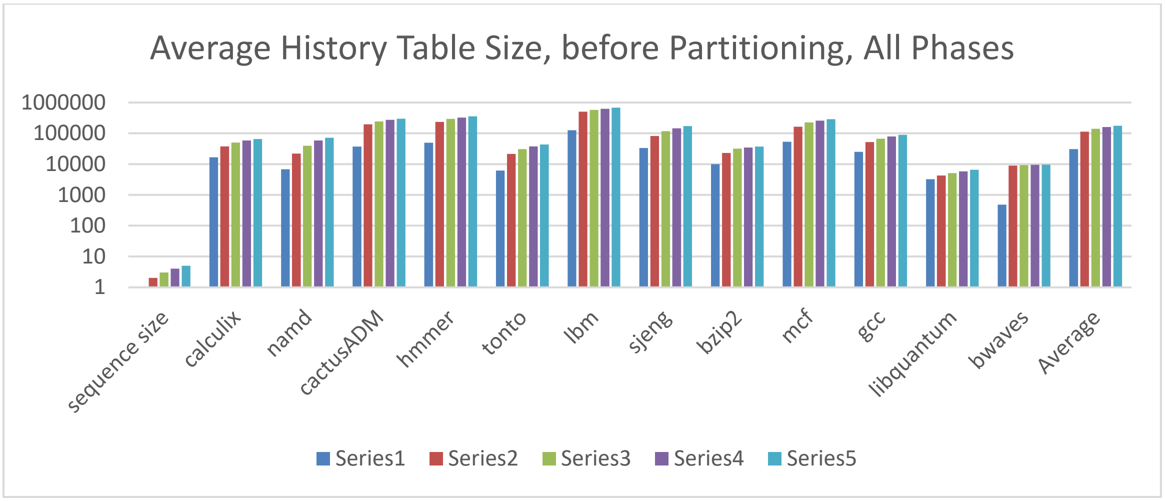
5.2. Partial-Global Stride Sequence and Instruction Partitioning

In this section, we discuss the limitations of global strides, and introduce partial-global strides as well as instruction partitioning in order to generate efficient partial-global stride streams.

5.2.1. Limitations of Global Strides



(a) Maximum history table sizes



(b) Average history table sizes

Figure 5-1: History table sizes for stride sequences, stride sequence length varying from 1 to 5

The raw global strides can form a large amount of stride sequence permutations. If the memory access stream is not partitioned, the global stride stream can lose conformity easily, and the number of unique global stride sequences, given a fixed sequence length, will significantly increase.

We first use a history table to study all unique global stride sequences, lengths varying from 1 to 5 strides. We wanted to find out how large the table need to be if cleared at each phase beginning in the length of one SimPoint[18] execution (100 million instructions). Our experiment shows that the number of unique global stride sequences is too large for a history table to be implemented. Shown in Figure 5-1, when the sequence length varies from 1 to 5 global strides, the maximum history table size across all phases and the average size per phase are provided for each benchmark. The maximum table sizes are on the order of 10^6, and the average table sizes are on the order of 10^5. Given that each entry, which is a sequence of global strides, contains at least 8 bytes, the area overhead and performance overhead from searching in the table are unacceptable. The sizes of the tables need to decrease drastically.

We observed the main issues that cause the global stride sequences to lose conformity:

- When a memory instruction outside a loop follows a memory instruction inside a loop, or vice versa, the stride between the memory access locations may vary from different iterations of the loop.
- When the interleaving memory instructions access data structures that have different element sizes, the memory instruction stream is mixed with different local strides, and the global stride will lose its pattern.
- Some instructions access random memory locations as opposed to an array, e.g. Pointer-chasing data structures. They have random effective addresses and strides, which corrupts the global stride patterns.

These issues indicate that the conformity depends on the static instructions that comes into the stream. A subset of the static instructions, when selected correctly, can provide a clear stride sequence pattern, while the pattern can be destroyed if one or more other static instructions are allowed into the subset.

5.2.2. Partial-Global Stride

To better describe the partitioning of the memory instructions and reference streams, we introduce the concept of partial-global stride. A partial-global stride falls in between local and global strides, meaning the strides generated by a subset of all loads and stores.

Let the reference stream in Equation 5-1 be divided into two sub-streams:

Equation 5-4: $\mathrm{RS_{global}}^0 = \{\mathrm{R_0, R_3, ..., R_{n-2}, R_n}\}$, $\mathrm{RS_{global}}^1 = \{\mathrm{R_1, R_2, ..., R_{n-3}, R_{n-1}}\}$,

We rename the sub-streams in Equation 4 as:

Equation 5-5: $\mathrm{RS^0_{partial-global}} = \{\mathrm{R^0_0, R^0_1, ..., R^0_{m-1}, R^0_m}\}$, and

$$RS^1_{partial-global} = \{R^1_0, R^1_1 ..., R^1_{n-m-1}, R^1_{n-m}\},$$

Then a partial-global stride can be generated from either RS^0 or RS^1. For example, from RS^0:

Equation 5-6: $\mathrm{S^0_k = R^0_{k-1} - R^0_k}, 0 < \mathrm{k} \le \mathrm{m}.$

The sequence of partial-global strides from RS^0 is:

Equation 5-7: $\mathrm{SS^0_{partial-global}} = \{\mathrm{S^0_1, ..., S^0_{m-1}, S^0_m}\}.$

To capture the patterns within memory strides, we want to divide the static instructions into partitions, and isolate the instructions that when put into one stream, disturb the patterns.

As an example, let us assume the static instructions that generated RS_{global} in Equation 5-1 is divided into two partitions, P_0 and P_1, which each contains a subset of all static instructions. Then, the reference stream RS_{global} is divided in Equation 5-4 and Equation 5-5. A reference goes into $RS^0_{partial-global}$ if its static instruction belongs to P_0, and $RS^1_{partial-global}$ if its static instruction belongs to P_1. Each reference stream generates its own partial-global stride stream, $SS^0_{partial-global}$ and $SS^1_{partial-global}$. Each partial-global stride stream has its own history table, and is free from the disturbance of references in the other instruction partition.

5.2.3. Instruction Partitioning

The goal for partitioned tables is to have fewer stride sequence permutations, which contributes to smaller area, less power consumption, and a lower false positive rate.

Among the 3 issues listed in 5.2.1, #1 and #2 are the most prevalent. We use an example in Figure 5-2 to explain how we can partition the static instructions to resolve these issues. Figure 5-2 shows a for loop that accesses two separate arrays of data structures, arc and $perm$. While arc is accessed in lines 3 and 6, line 6 is executed only when the condition in line 3 is met. Similarly, $perm$ is accessed in line 10-12 only when the condition in line 7 is met. If we isolate the memory instructions into three groups by their line numbers, $line$ 3, $line$ 6, and $lines\ 10 - 12$, the instruction stream within each group will have a consistent memory address shift, and thus a clear stride pattern.

```
1       for( ; arc < stop_arcs; arc += nr_group )
2       {
3           if( arc->ident > BASIC )
4           {
5               /* red_cost = bea_compute_red_cost( arc ); */
6                     red_cost  =  arc->cost  -  arc->tail->potential  +  arc->head-
>potential;
7               if( bea_is_dual_infeasible( arc, red_cost ) )
8               {
9                   basket_size++;
10                  perm[basket_size]->a = arc;
11                  perm[basket_size]->cost = red_cost;
12                  perm[basket_size]->abs_cost = ABS(red_cost);
13              }
14          }
15      }
```

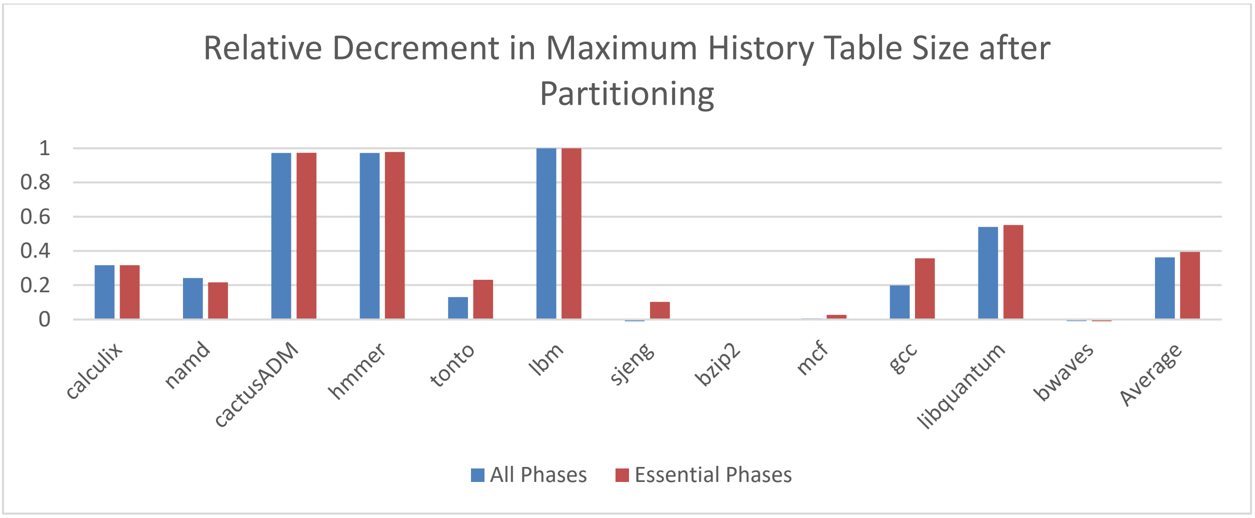
Figure 5-2: Code snippet from SPEC CPU2006 [29]

The key observation is that, the number of times that each of these lines are executed can help distinguish whether it should be put in the same group with another line. This characteristic can be represented by the number of unique memory references the corresponding instructions will make. For example, assuming that line 3 is tested m times and taken k times, while line 7 is taken l times, then line 3 will have m unique memory references, line 6 will have k unique memory references, and line 10-12 will have l unique memory references.

Thus, we use the number of unique memory references as the metric for partitioning static instructions. In each individual instruction working set phase, if some instructions appear to have the same number of static memory references, it's likely that they are accessing memory locations with similar size and similar stride, and therefore among them there is a better pattern.

The multi-table approach is compared against the single-table approach to show the benefit of partitioning. We collected data on the number of unique sequences and false positive rates before and after the static instructions are partitioned into groups. The program execution is divided into phases based on different instruction working sets, and at the beginning of each phase the instructions are re-partitioned, and the history tables are cleared for the new partitioning.

From Figure 5-3, the partitioned tables show much fewer entries of unique global stride sequences, i.e. the total size of history tables to contain all possible global stride sequences is much smaller than a single table. The following two figures show the decrement of maximum and average history table sizes after partitioning, in a period of a SimPoint, which consists of 100 million instructions. The *All Phases* data is from all instruction working set phases, while the *Essential Phases* data is from all the unique phases excluding their repetitions.



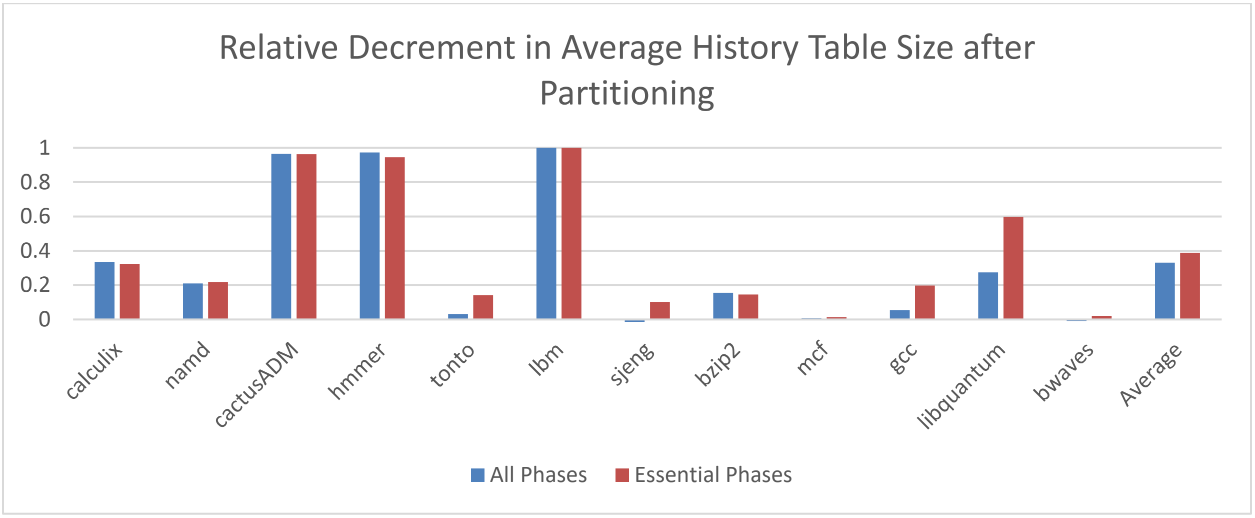


Figure 5-3: History table size change after instruction partitioning

Among the benchmarks, lbm, cactusADM, and hmmer benefit the most from the partitioning, while mcf, tonto, and bwaves have little improvement. The benchmarks receiving little benefits all have a low false positive rate to start with, i.e. the global stride patterns before partitioning are already well organized. One exception, however, is mcf which has more pointer-chasing than other benchmarks.

5.2.4. Hardware for Instruction Partitioning

The instruction partitioning depends on counting the number of unique memory references of each static instruction. We dedicate a certain length of learning period for this purpose. However, such functionality can be expensive to implement, since it is search-intensive and the number of unique references of a static instruction can be quite large. Implementation using SRAM is relatively inexpensive in hardware design but incurs high search latency and results in high performance penalty. Other approaches, mainly using ternary content addressable memory, achieves constant search time [30] but requires complex hardware and incurs high area and energy overhead [31]. In this section, we present an area and energy-efficient hardware design to achieve the instruction partitioning.

We make two important observations in order to create this design. First, the partitioning information can be used on a repeated phase. Thus, there does not have to be learning at every phase transition, but only at a new unique phase. Second, to test whether a reference is an unseen reference of a static instructions, we do not need a cache to store the original PC and reference but can instead use a Bloom filter.

Beside the search Bloom filter, a cache is needed to store static PCs and a counter associated with each PC. The design is illustrated in Figure 5-4.

To perform instruction partitioning, we dedicate a learning period specifically for learning the instruction partitions. This learning period needs to happen before the learning period needed by the stride and value Bloom filters described in Chapter 6, since the stride and value Bloom filters rely on the instruction partitions to get their content.

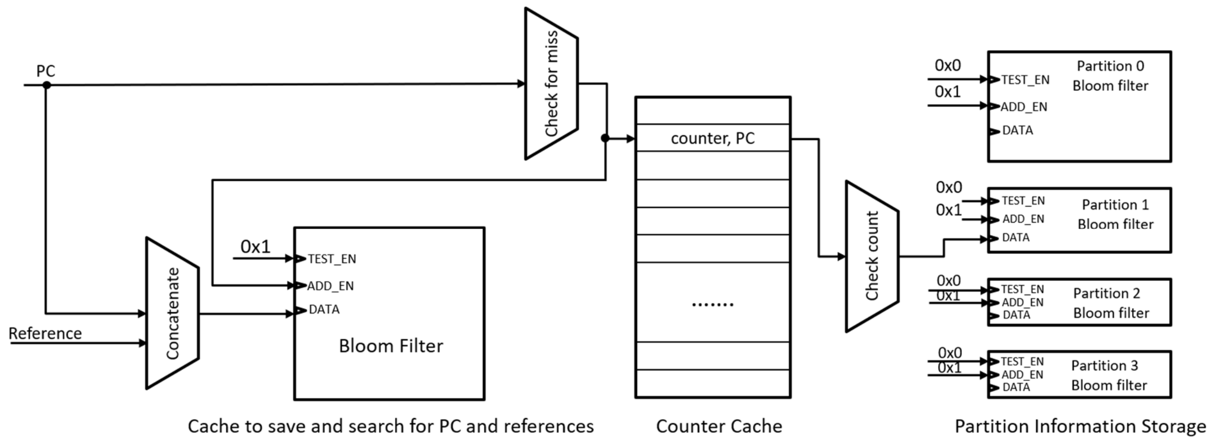


Figure 5-4: Hardware design for instruction partitioning and partition information storage

Search Bloom Filter

During the learning period, we concatenate the lowest 24 bits of the PC and the lowest 20 bits of the reference and checks whether this data is in a Bloom filter. If it incurs a miss, the counter for this PC is increased, and this data is added into the Bloom filter. At the end of the learning period, each counter within the cache is checked to a predefined group of ranges, and the associated PC is added to the cache storage for that corresponds to this counter's range. We use Bloom filters to store the partition information as well. To reduce the pollution of the search Bloom filter, we skip the insertion of a PC and reference pair if this PC's unique references have reached the upper limit of the highest range.

The search Bloom filter is 4KB in size. With a learning period length of 100K instructions for each unique phase, we observe 8% discrepancy in the partitioning outcome compared with perfect storage and perfect search. This discrepancy is due to the false positives when testing Bloom filter membership, thus is always an underestimation of the counters for the misplaced instructions. This is acceptable because even if some static instruction PC's unique reference count is slightly underestimated, the partitions on the higher ranges still have their patterns intact.

Counter Cache

The counter cache is implemented to have 1024 entries, which is far higher than the average number of unique static instructions per phase, which is 402 in our experiments. The cache is 8-way set associative and uses the static instruction PC's lowest 24 bits as the address for a constant time search and update for this PC's counter.

Partition Information Bloom Filters

Once the learning finishes, the static instruction partitions are stored into the partition information Bloom filters. We use four partition information Bloom filters. They have heterogeneous sizes to accommodate different sizes of the partitions, that respectively occupies 50%, 25%, 12.5%, and 12.5% of overall capacity. With a total size of 2KB and inserted with all learned static instruction PCs, the Bloom filters shows a false positive rate of 3% on average.

The above structures are placed near the L1 cache for quick access. At phase transition time, the content of the partition information Bloom filters is saved to a L3 cache. We save the content for up to 20 unique phases to limit the impact on L3. If a repeated phase occurs, the content of the unique phase that is being repeated gets loaded from L3 to the near cache Bloom filters.

The Bloom filters are implemented using a parallel Bloom filter approach [25] in SRAM, where each parallel Bloom filter has only one hash function and need only one read/write port. With 32 nm technology, the near cache structures, including the search Bloom filter, the counter cache, and the partition information Bloom filters, add 0.55% area overhead to a Sandy Bridge-like core. As will be discussed in Section 5.5, we have two separate detectors for stride and value. Therefore, the area overhead for instruction partitioning is doubled to be 1.1%.

5.3. Selecting the Effective Bits as Input

The efficiency of the detector can be further improved by using only a subset of bits from the strides. Our preliminary study shows that in more than 99% of the strides in a 64-bit program, bits 20 to 63 have no variation. Thus, we only consider the lowest 20 bits of a stride as the input to our detector.

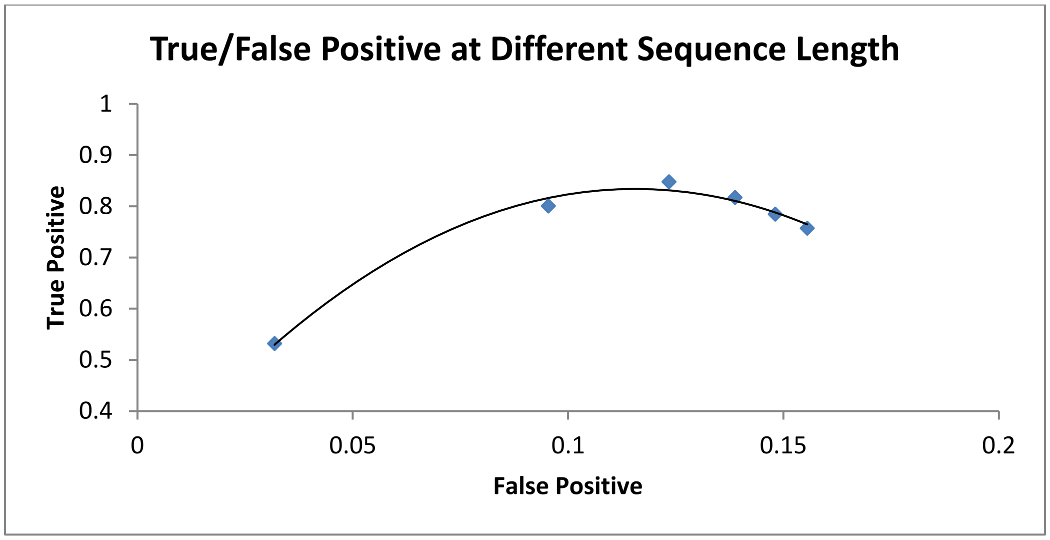


Figure 5-5: Impact of stride sequence length

To find out how many lowest bits we need for the fault detection, we did an exploratory study using the lowest 20 bits of each stride and compared the results to that of using the lowest 18 bits. In both cases, the sequence includes 3 strides. We experimented using a simple Bloom Filter. Given the same training length and Bloom Filter size, using the lower 20 bits could achieve an

83% detection rate and 6.4% false positive rate, while using the lower 18 bits could achieve a 45% detection rate and 2.3% false positive rate. These numbers indicate a trade-off between detection rate and false positive rate that can be adjusted through the number of bits chosen for the dataset. For the experiments in the rest of this chapter, we chose to use the lowest 18 bits.

5.4. Stride Sequence Length

Using a sequence of strides, instead of individual strides, as the input to the Bloom Filter, provides a view of pattern across strides. Since the number of unique stride sequences is exponential to the sequence length, the length must be carefully evaluated.

We studied the trend of the true vs. false positive rates as the sequence size changes from 1 to 6, shown as a ROC curve in Figure 5-5. The false positive rate increases as the sequence length increases, while the rue positive saturates itself and decreases when the sequence length is larger than 3. A sequence length of 3 can be used to achieve the highest detection rate. However, for an efficient fault detection design, the false positive rate should also be considered. When the sequence length increased from 2 to 3, the ROC curve has flattened compared to when the length increased from 1 to 2. Therefore, a sequence length of 2 is more cost efficient, and we will use this length for the subsequent studies.

When used as the input to a Bloom filter, the sequence of strides is concatenated into a single bit array. For example, when a sequence of two strides are used as an input, each containing the lowest 18 bits of a stride, the two 18-bit vector are concatenated into a single 36-bit vector. This 36-bit vector is the input to the Bloom filter.

5.5. Using Memory Value in Addition to Stride

The data value from a memory access should also be used to increase the fault coverage. Strides can be efficient to use when detecting faults, but not all faults result in a stride variance. In some cases, a data altered by a single-bit transient fault will not influence the effective addresses of any memory instructions. This type of fault can only be detected by examining the memory data value.

Thus, we use data values in addition to strides as our input source for fault detection. To capture the patterns within each stream, these two types of inputs are isolated into two detectors, the Stride Detector and the Value Detector.

The effective bits and sequence length were studied on values as well. Like strides, values have little variance in the higher bits, thus we only use the lowest 18 bits of a value. Also, like strides, value errors can be better detected when static instructions are partitioned into separate sections of the detector. Our study on sequence length shows that values are best used individually, thus a sequence length of 1 is used for values through the rest of this work.

Chapter 6 The Detector: Bloom Filters

This chapter describes a fault detection mechanism that uses Bloom Filters to exploit the temporal and spatial locality within the partial-global strides and memory data. We will discuss the design and implementation of the Bloom Filters, their effect on fault detection, and the cost on performance, area, and power consumption.

6.1. Overview

Bloom Filters are used to store the history of memory access patterns in their hashed form and check for incoming patterns' membership for fault detection. In the learning period, a Bloom filter is warmed up by inserting memory stride sequences or data values. In the detection phase, the incoming stride sequences or data values are tested against the filters to predict faults.

In Chapter 4, we explained how the different components of the detector are orchestrated on a high level, including identifying phase transition and phase repetition, storing and loading Bloom filter contents, and using the replay and rollback mechanisms to handle back-end and front-end faults. A more detailed architecture overview including the Bloom filter setup is shown in Figure 6-1. Note that the instruction partitioning information Bloom filter is not shown in detail but abstracted into the Partition Selector; only one set of Bloom filters, representing the stride detector or value detector, is shown in the figure.

Within each Bloom filter, a random XOR-folding hash function is used, which will be discussed in Section 6.2. The individual Bloom filters, 0 through 3, each handles a separate static instruction partition. The heterogeneous Bloom filter sizes for each partition will be discussed in Section 6.3. To reduce hardware overhead, a parallel Bloom filter implementation is used for all Bloom filters and is discussed in Section 6.4. The effect of the learning period length is explained in Section 6.5. Section 6.6 presents the experimental results, including fault coverage, false positive rate, and costs on performance and energy.

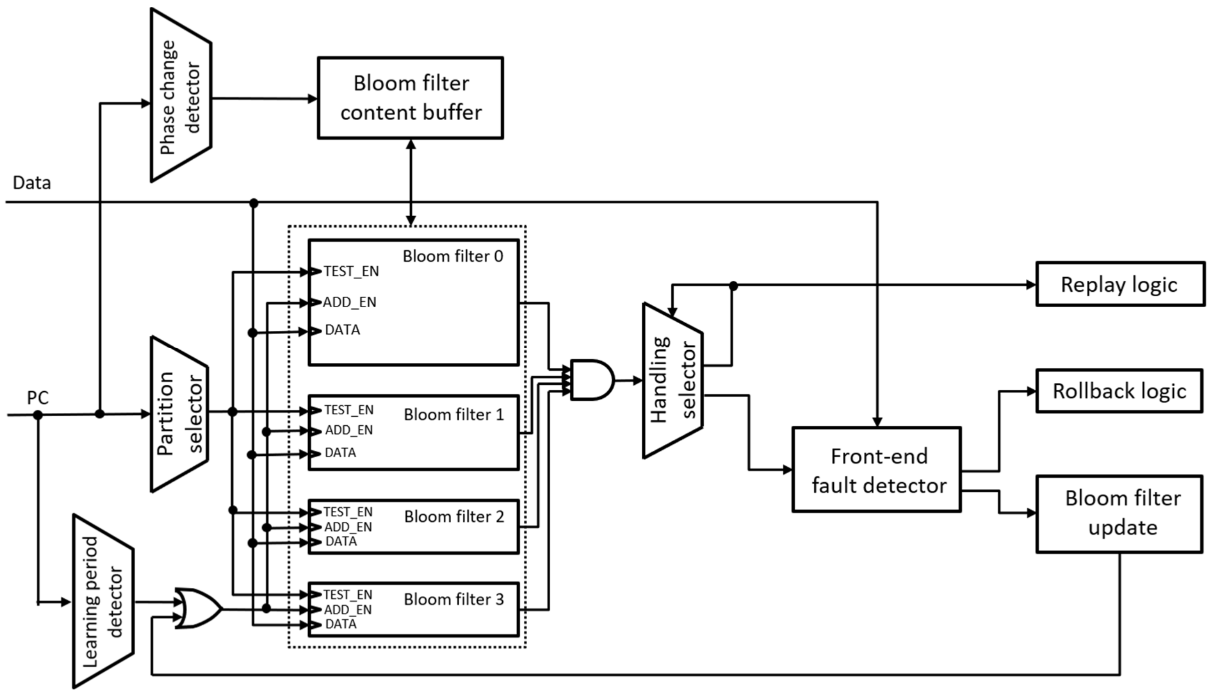


Figure 6-1: Design of the value-based fault detector

6.2. Hash Function

A Bloom filter is a bit array that can represent a set of values and test whether a value is a member of the set. Each Bloom filter has a set of independent hash functions associated with it.

Each hash function can transform an input value into a hashed value. For a Bloom filter with an m-bit array and k hash functions, when an input value is given, each hash function can generate a value in the range of $[0, m\text{-}1]$ to represent a bit location, and the locations within the bit array indicated by the k hashed values are set to insert the input value, or tested to see whether the input value is a member of the set.

For our usage, we want our Bloom filter to detect anomalies among all bits of the input value. For this purpose, we explored a few options of hash functions.

Shift-and-Add Hash Function

A simple, widely-used hash function is Shift-and-Add as shown in Figure 6-2.

```
def simple_hash(values, seed, cap):
    result = 0
    for i in range(len(values)):
        result = result*seed + values[i]
        result = result & (cap - 1)
return result
```

Figure 6-2: Python code for a shift-and-add hash

The input bit array is broken into segments, creating a value array $values$. With each seed, this hash function produces a bit location for the Bloom Filter to set. A set of different seeds are used to set multiple locations for the same input value.

This hash function has uneven detection efficiency among the bits of a global stride, since its approach is limited by its uneven usage of each bit in the stride.

Uniform XOR-Folding Hash Function

XOR-Folding hash functions create more equal detection opportunities for each bit within the strides. If a stride sequence has m bits, and the Bloom Filter has a capacity of 2^n, a Uniform XOR-Folding hash divides the stride sequence into segments of k bits, where $k = \frac{m}{n}$. Then, the segments are XORed, creating a result of n bits, and this n-bit number indicates the bit location in the Bloom Filter to be set. Figure 6-3 shows how the Uniform XOR-folding hash function works.

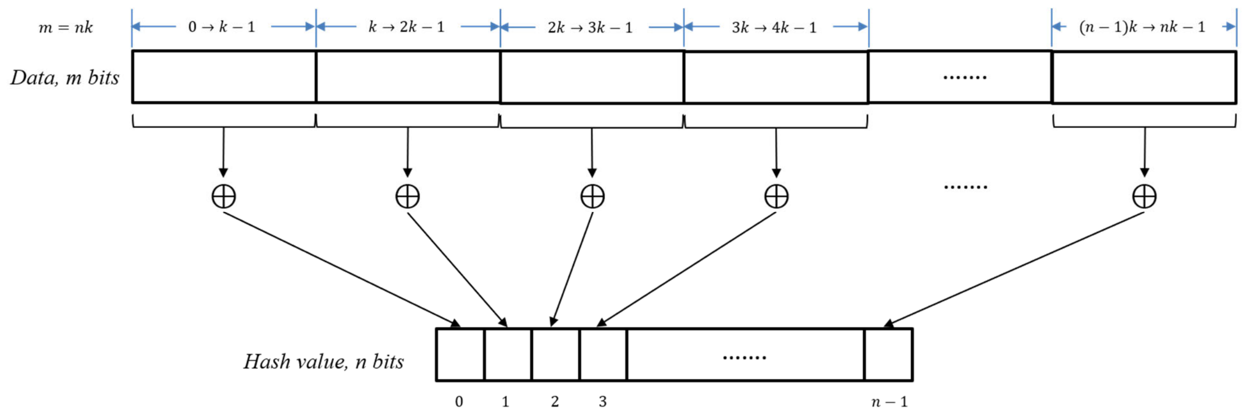


Figure 6-3: Uniform XOR-folding of an m-bit input into an n-bit value

To create multiple hash functions on the same input, each hash function can circularly shift the input bit array by a different offset, and then uniformly XOR it.

The uniform XOR-folding hash may work well with inputs where the bits values are uniformly-distributed. However, the different bit locations of memory stride sequences and values are not equal, since some bits may have more variance than others. Thus, we explored the random XOR-folding hash as explained next.

Random XOR-Folding Hash Function

Known as the *H3* hash function [23], Random XOR-Folding Hash uses the result of a random offset selector, selects bits semi-randomly from the global stride sequence without repetition, and form segments of k bits. Random XOR-Folding Hash mitigates the uneven utilization of the Bloom Filter bits [24] that compromises the Uniform XOR-Folding Hash fault coverage.

To create multiple hash functions on the same input, different random offset selectors are used.

The Number of Hash Functions

The number of hash functions, together with the size of the bit array, determines the probability of a false positive to a Bloom filter. A Bloom filter false positive means when an input value is tested to be a member of the set, while in fact it is not. When we use a Bloom filter for fault detection, this test false positive translates into an escaped anomaly, which then translates into loss of detection rate. In general, when the data set is small relative to the Bloom filter size, the number of Bloom filter false positives is reduced as the number of hash functions increases. However, as the data set gets larger, this trend can get reversed [25]. We compared using 4 hash functions against 8 hash functions. The result is shown in Figure 6-7, which indicates better outcome with 4 hash functions.

6.3. Heterogeneous Bloom Filter System

In Chapter 5, we explained the importance of instruction partitioning. The outcome of the instruction partitioning is a number of sets that each contains some static instruction PCs. These sets do not necessarily have the same size, and the memory access pattern they hold do not, either. Thus, it is important to assign a proper size for the Bloom filter of each partition.

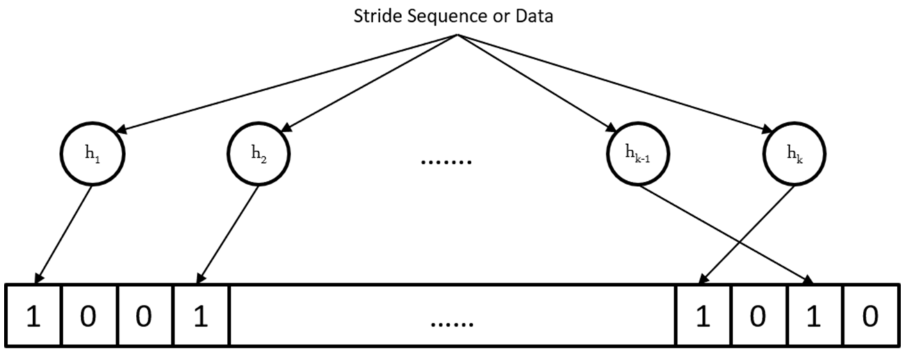


Figure 6-4: Bloom Filter without Partitioning

A Bloom filter implementation before partitioning is shown in Figure 6-4, which uses k hash functions. In the following sections, we will discuss how to partition the static instructions and assign sizes to the Bloom filters.

Identify the Instruction Partitions

In Chapter 5, we explained how the static instructions can be partitioned using number of unique memory references as a measurement. In our implementation, we perform a profiling of the static memory instructions and their number of unique references, u. The static instructions are then selected and divided into four partitions: $u = 1$, $2 \leq u \leq 10$, $11 \leq u \leq 40$, and $41 \leq u \leq 100$. All static instruction PCs within a partition are stored in dedicated cache. The Partition Selector in Figure 6-1 searches this cache for a PC and selects the corresponding Bloom filter partition. The stride sequence and memory value associated with this PC then goes into that Bloom filter partition.

Estimate Bloom Filter Size for Each Instruction Partition

Not all instruction partitions require the same Bloom Filter size. To achieve a high fault detection rate while keeping the false positive rate low, we need to consider factors including the number of unique PCs in the partition, the number of unique references per PC, the size of the stride sequence, etc. For the stride Bloom filter, when determining the Bloom Filter size, the instruction partitions are sorted by the number of unique PCs they hold. The largest Bloom filter partition is assigned to the static instruction partition with the greatest number of PCs. For the value Bloom filter, however, the number of unique values should also be considered. The product of the unique PC count and the counter lower limit is used to sort the instruction partitions. Considering the feasibility of implementation, we use four partitions, with their size proportional to 1:1:2:4., and so on.

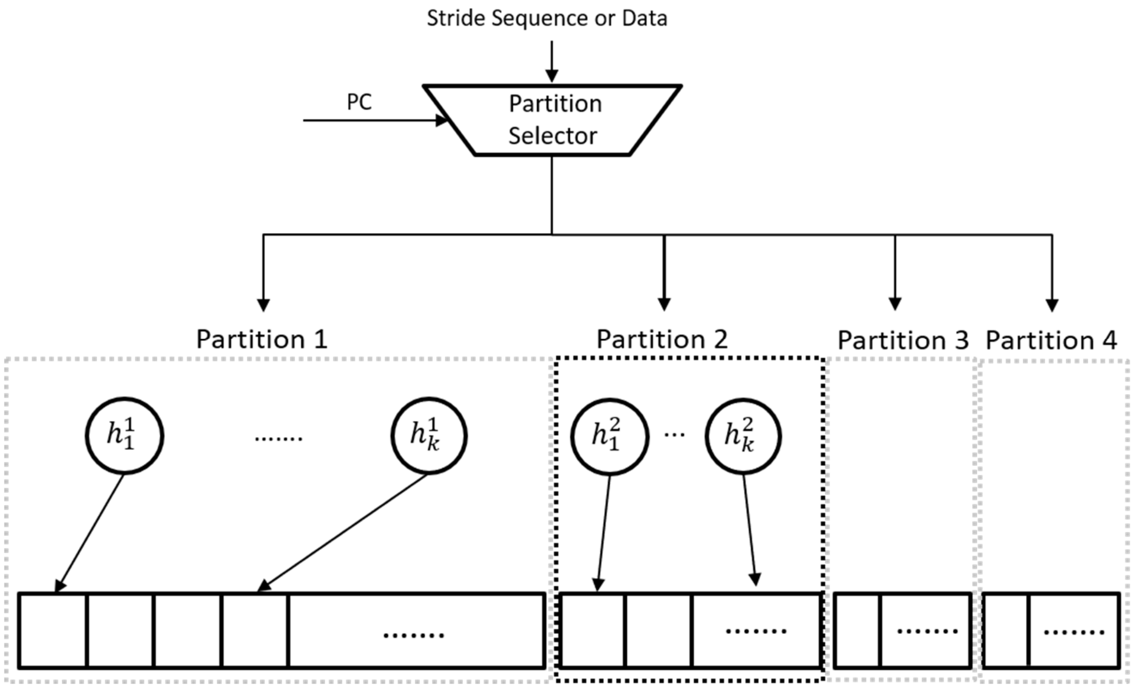


Figure 6-5: Bloom Filters after Partitioning

The Bloom filter after partitioning is shown in Figure 6-5. Each partition will still have k hash functions. In the example in Figure 6-5, Partition 2 is selected for the input stride sequence or data, since the Partition Selector designated Partition 2 for the current PC.

6.4. True versus Parallel Bloom Filters

To implement a Bloom filter with *k* hash functions in the SRAM, we'll need *k* read and write ports. Since the size of the SRAM increases quadratically with the number of ports, we will need an area-efficient implementation of the Bloom filter.

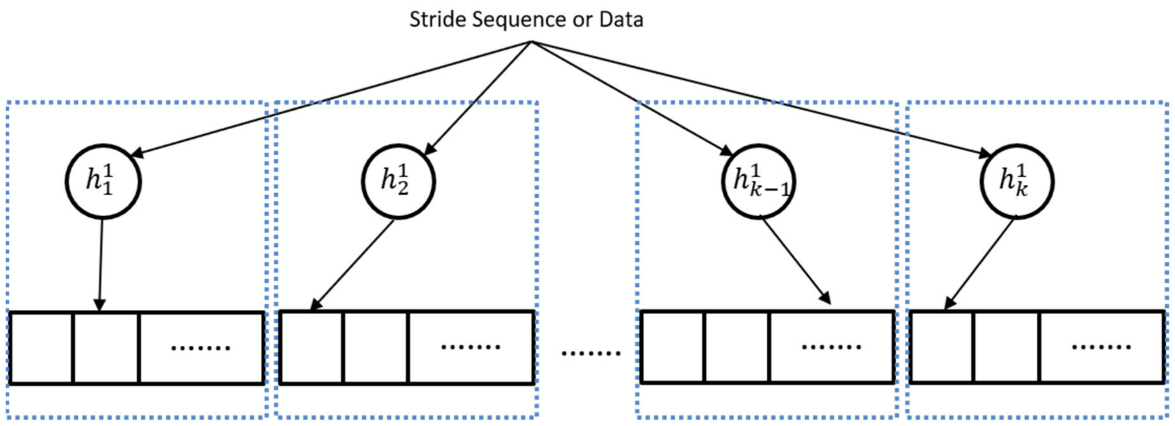


Figure 6-6: Parallel Bloom filter implementation for one partition

The concept of a true Bloom signature versus a parallel Bloom signature has been introduced by existing works [24]. By using the parallel Bloom filter implementation, the area overhead is quadratically reduced, while the detection rate is barely affected.

In our study, we use a similar definition. A true Bloom filter is when all *k* hash functions are used by a single Bloom filter. Its parallel Bloom filters counterpart is defined by dividing the bit field of the true Bloom filter into *k* Bloom filters, and each assigned one hash function. This implementation is illustrated in Figure 6-6.

Our experiment shows that the parallel Bloom filters achieves a fault detection rate for over 94% of the detection rate of a true Bloom filter, while the false positive rate is reduced by 36% compared to a true Bloom filter. The result is shown in Figure 6-7. In this experiment, we use a sequence of 2 strides for the stride Bloom filter, and a sequence of 1 data value for the value Bloom filter, while both filters are of 1KB size, with 10000 micro-ops training length. The results are extracted from the 12 SPEC CPU2006 benchmarks, and each data point represent the average value of all 12 benchmarks' results.

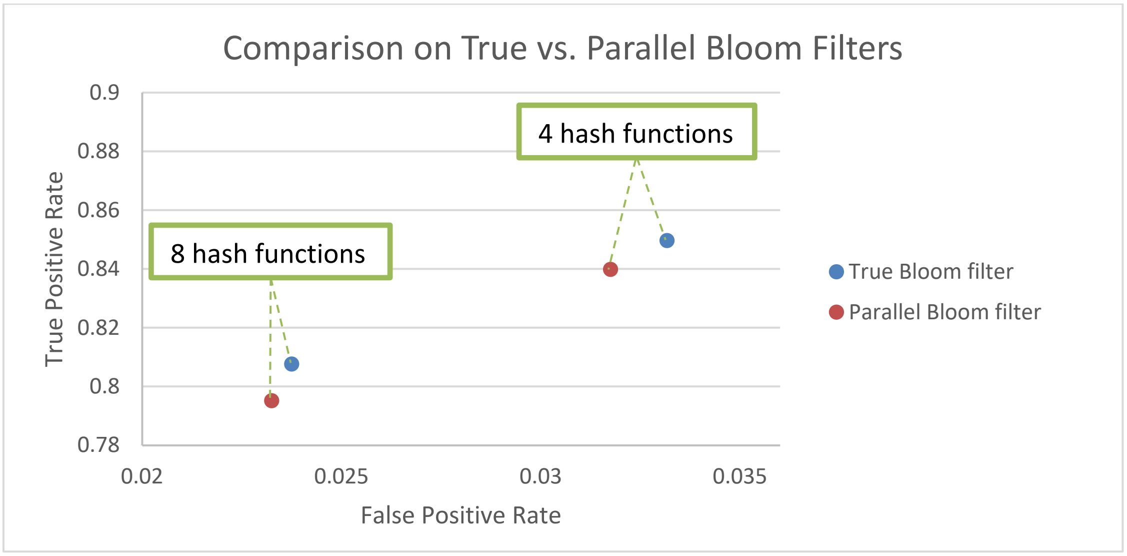


Figure 6-7: Fault coverage and false positive comparison between different Bloom filter implementations: True Bloom filter and parallel Bloom filter

6.5. Effect of Learning Period Length

A learning period is dedicated to the stride and value Bloom filters, after the instruction partitions are learned or re-loaded. The learning period length is a vital parameter that affects the effectiveness of the stride and value detector and the performance of the system.

Effect on Fault Detection Rate and False Positive rate

The length of the learning period has a direct impact on the fault detection rate and false positive rate. Since new stride sequence or data value can keep showing up, the learning period length is positively related to the size, *n*, of the data set inserted into the Bloom filters. In the meantime, the probability of a single bit still being 0 after *n* insertions is negatively related to *n* [25]. Therefore, the longer the learning period, the more bits get set in the Bloom filters, and the higher false positive rate when testing a new value against a Bloom filter. As a fault detector, the Bloom filters will report lower fault detection rate and lower false positive rate.

Effect on Energy Cost

During the learning period, the program thread needs to be protected using additional measures since the detector is not detecting any fault. The thread can be protected by protector threads that form a dual modular redundancy (DMR) or triple modular redundancy (TMR) [1] [69]. This result in energy overhead that is proportional to the learning period length.

Learning Period Study

To quantify the fraction of the learning period length relative to the program execution length, we first examine when the learning period is needed. To keep the fault detector's sensitivity, we divide the program execution in to working set phases [7], and each phase transition will trigger a refresh on the Bloom filters. We use a working set phase length at a granularity of 100,000 instructions, for a total execution length of 100 million instructions. We enable checking on repetitive phases. Within the 100 million instruction program execution, we see on average 13 unique phases across the benchmarks. For each working set phase, we either have an explicit learning period, or reuse the warmed Bloom filter from the previous occurrence of an identical phase.

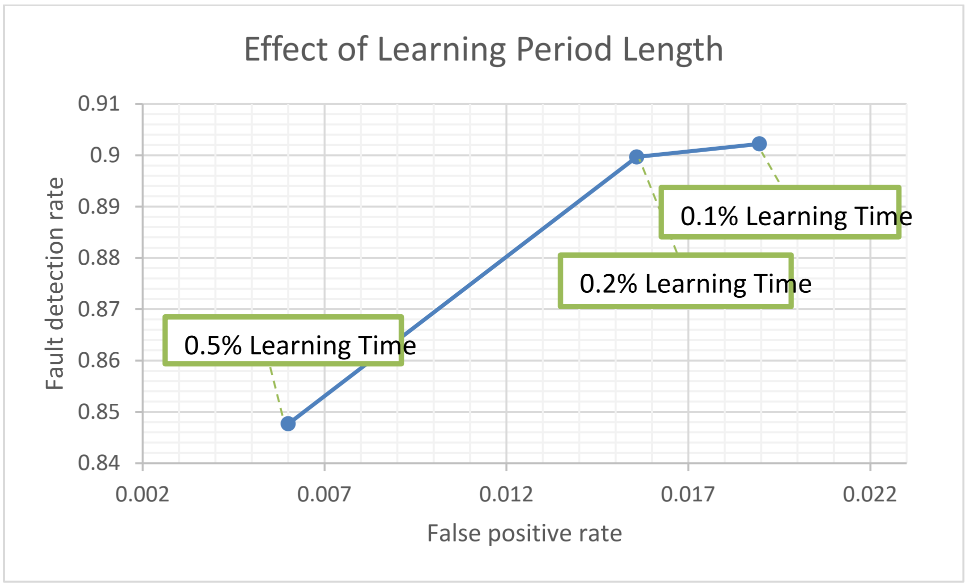


Figure 6-8: Impact of learning period length on fault coverage and false positive rate

To optimize the effect of the learning period, we experimented with 3 different learning period lengths: 4000, 16000, and 40000 instructions per phase. Let the learning period length be denoted as *m*, the execution length as *n*, and the number of unique phases as *k*. The fraction of the learning period relative to the program execution length can be calculated by:

$$m * k / n$$

Since k=13 (See Figure 4-7), the three learning period lengths project to 0.1%, 0.2%, and 0.5% of the program execution time.

Figure 6-8 shows the fault detection rate and false positive rate of each of the learning period lengths. Both rates are negatively related to the length of the learning period. The low learning times, 0.1% and 0.2%, guarantee high fault coverage at 90%, but report over 1.6% and 1.9% false positive rates respectively, which can create a drastic performance overhead. The 0.5% learning time, however, shows a fault coverage of 84% while incurring only 0.6% false positive rate. Thus, in our subsequent study, the 0.5% learning time is used.

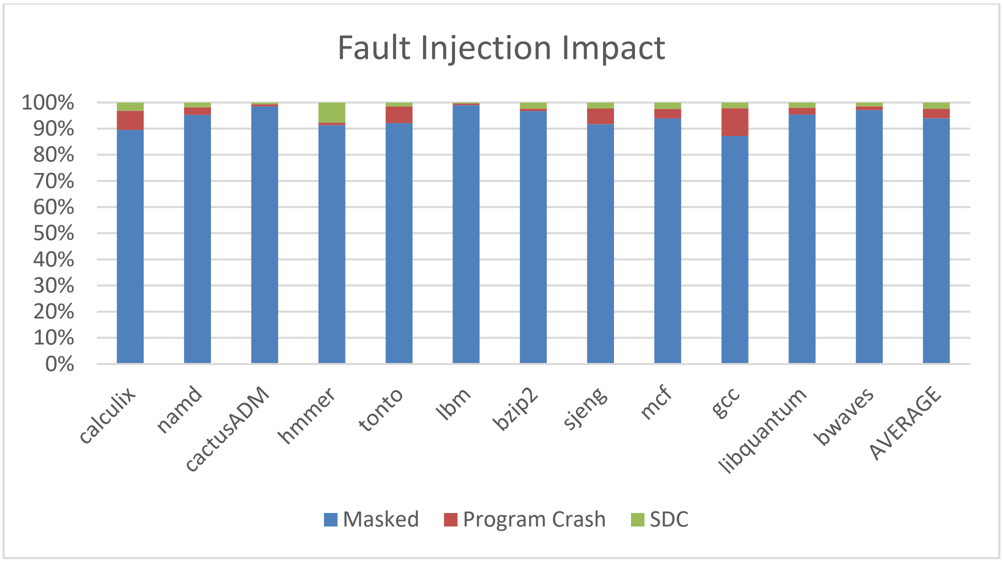
6.6. Evaluation

We evaluated the fault detector design on an out-of-order X64 architecture as described in Chapter 5. We will show both the fault detection rate and the false positive rate and discuss the impact on performance, area, and energy consumption.

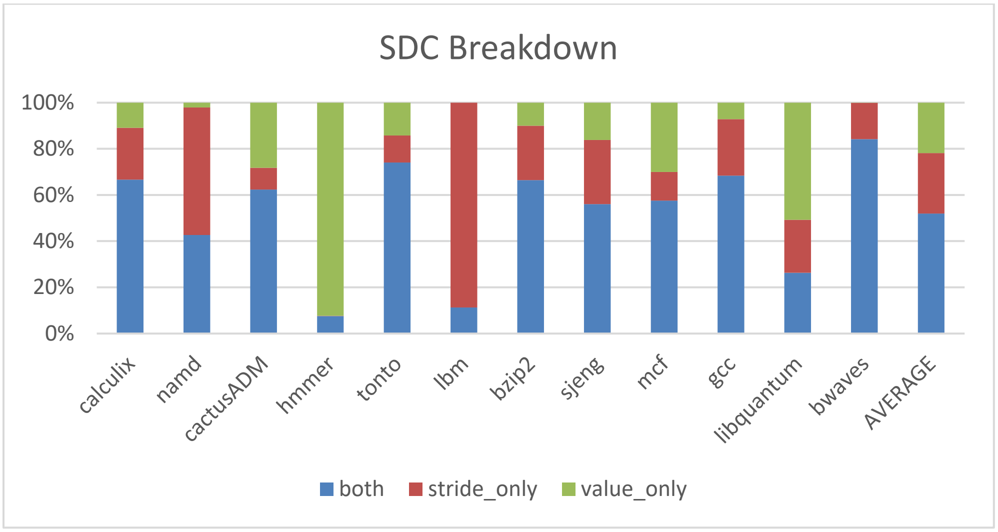
6.6.1. Fault Injection

The fault injection methodology we chose is flipping a register bit before its write-back. This method can account for an abundance of soft error behaviors [16, 17].

For each of the 12 SPEC CPU 2006 benchmarks we evaluated, we injected 50,000 faults. Out of all injected faults, 3.7% resulted in program crash. For the remaining faults, we distinguish SDC from a masked fault by monitoring the dynamic memory instructions, as well as memory reference and memory value traces, for at least 5000 instructions after the point of fault injection. Similar to past studies [16, 17], we assume that a fault is masked if no divergence is found after the monitored length. The fraction of masked faults, faults that resulted in program crash, and SDCs are shown in Figure 6-9 (a). On average, 1173 faults manifested themselves in the form of silent data corruption, and we will report our coverage on these faults only.



(a) Fault injection broken down by impact on program execution



(b) Silent data corruptions broken down by impact on stride and value

Figure 6-9: Fault Injection Outcome

A further breakdown of the SDCs are shown in Figure 6-9 (b) by whether they affected the memory value stream, or the memory stride stream, or both. On average, 82% of faults altered the memory value stream, while only 63% altered the memory stride stream. For 8 out of 12 benchmarks, >50% faults affected both stride and value streams. For *hmmer* and *libquantum*, the majority of the faults affected the value stream only. For *lbm* and *namd*, there is a significant portion of faults that affected stride only. The diversified characteristics are expected, since our fault injection is through the write-back register. If a memory instruction's reference does not go through any computation logic and never gets writen back to a register, the reference will not be modified, even though the value associated with the same instruction can be changed by a fault, which results in the value trace diverging from the fault-free trace while the stride trace stays unchanged. The same applies to when a fault results in stride trace divergence but not value trace changes.

6.6.2. Detector Configuration

Delay buffer	7 instructions
Instruction partition learning Bloom filter	4KB (reference-based partition), 4KB (value-based partition)
Instruction partition learning counter cache	1024 entries, 4 bytes per entry, 4-way set associative
Instruction partition storage Bloom filter	2KB (reference-based partition), 2KB (value-based partition)
Detector Bloom filters	1KB (stride), 1KB (value)
Number of saved unique phases per 100M instructions	20 (unique phase signatures, instruction partition Bloom filters, and detector Bloom filters in L3)

Table 6-1: Hardware configurations for the fault detection structures

Memory stride and memory data are both used for fault detection, but in two isolated detectors. Both detectors have the same total size of 1KB and are partitioned into 4 partitions using the same methodology mentioned in Chapter 5. The stride detector takes in stride sequences that consists two strides, while the data detector takes in a single memory value at one time. Both uses only the lowest 18 bits of the stride or data.

A detailed hardware configuration is shown in Table 6-1. We use the replay mechanism with a delay buffer same as FaultHound [17] to correct faults on the backend. Within the length of a 100M instruction SimPoint [18], a maximum of 20 unique phases' detector information can be saved and reloaded when repeated. This includes the unique phase signatures, the Bloom filters that stores the instruction partition information, and the Bloom filters for the stride and value patterns.

6.6.3. Detection using Memory Stride and Memory Value

In this section, we discuss the fault coverage and the false positive rate of the detector. The two detectors can have an overlap in their fault coverage, while they are also complimentary enough to boost the overall fault detection rate. The reasons are twofold. On one hand, the injected faults can have an impact on stride only or data only. On the other hand, the two detector's input data sets are fundamentally different even when no fault is injected. These conditions create a discrepancy in the two detector's behavior.

Fault Coverage

The fault coverage is shown in Figure 6-10. On average, we achieve 84% fault coverage on all faults that can be corrected on the backend. We use a processor core pipeline similar to the one used by FaultHound [17], and the backend, which is 80% of the pipeline area, is responsible for a proportion of 80% of all faults injected in the core, while the frontend is responsible for the other 20%. Thus, the backend fault detector covers 67.2% of all faults. As a comparison, PBFS [16] covers 30% backend faults, while FaultHound has a coverage of 60%.

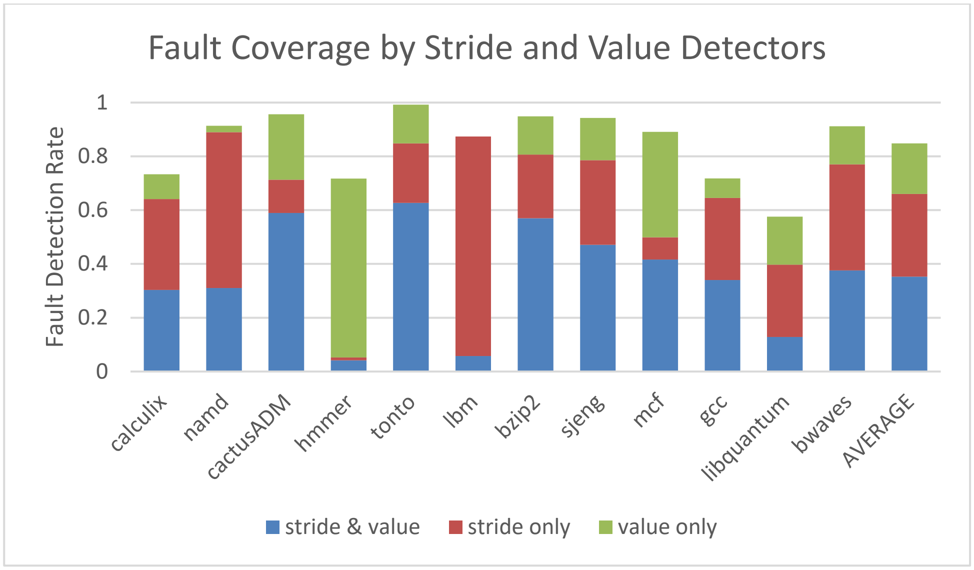


Figure 6-10: Fault coverage of the value-based detector

There are some dramatic differences among the benchmarks in fault coverage. For hmmer, the value detector is the main contributor in fault detection. This is because hmmer's reference/stride trace is insensitive to the injected faults, leaving little room for the stride detector to perform. For the same reason, lbm benefits largely from its stride detector. Lbm shows higher coverage than hmmer, which is most likely because lbm focuses on matrix computation and has a small memory instruction working set, and the stride patterns are quite strong as shown in Figure 5-3. Libquantum shows the lowest detection rate. One possible reason is the low memory instruction percentage: libquantum has only 12% dynamic memory instructions, compared with 24% average of all benchmarks. Other possible reasons include noises in the value patterns; its value detector detected fewer faults than the stride detector even though there are far more diverged value traces than stride traces.

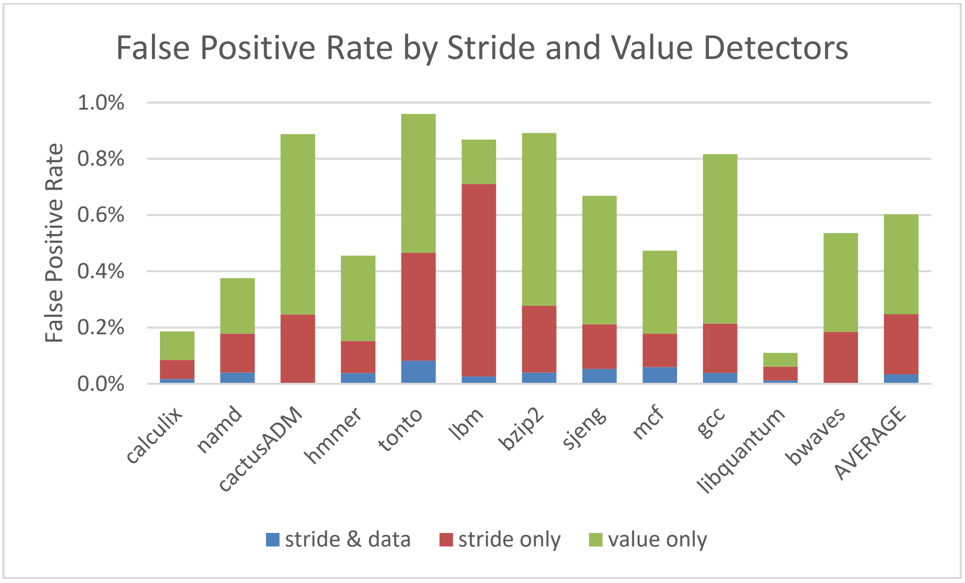
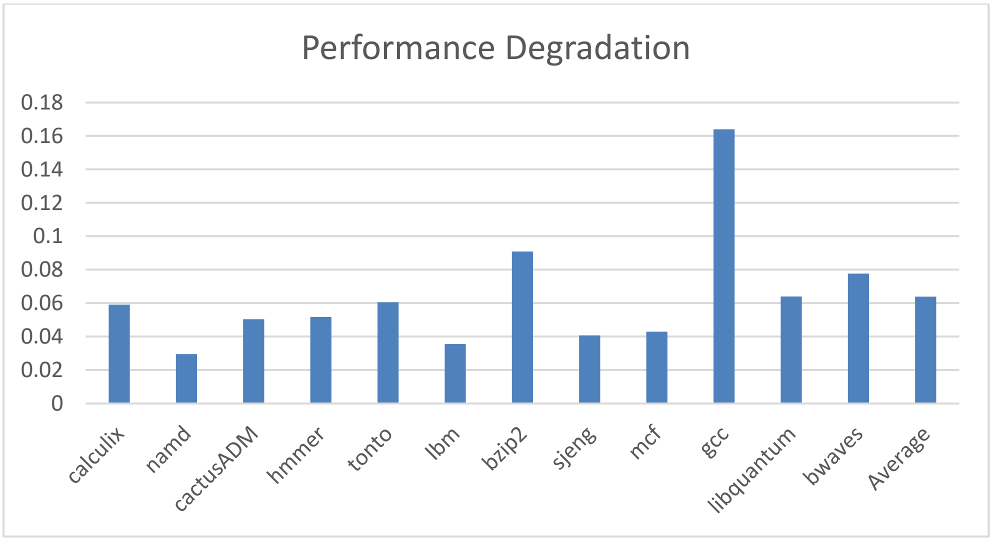


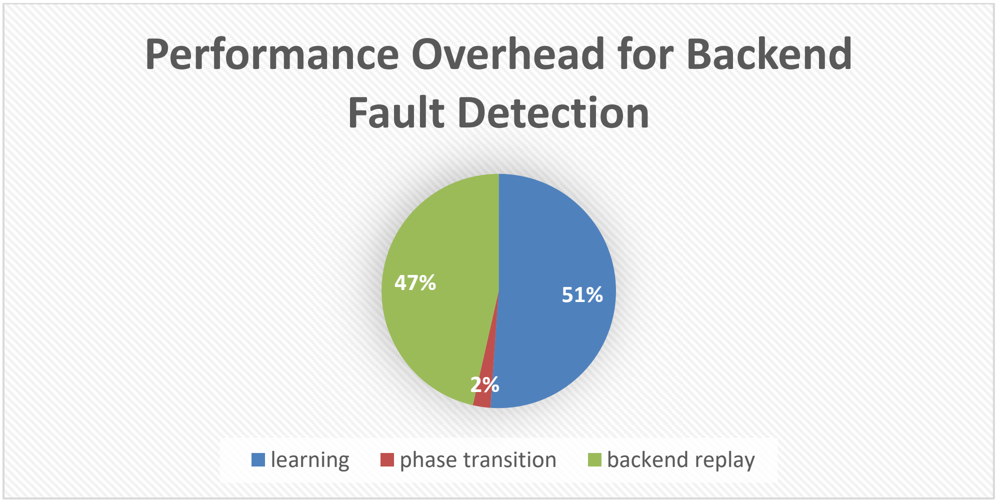
Figure 6-11: False positive rate of the value-based detector

False Positives

The false positive rates are shown in Figure 6-11 with an average of 0.6% of all instructions. This is comparable with PBFS [16]'s ~0.5% and much smaller than FaultHound [17]'s ~3%. This implies low energy overhead of our fault detection mechanism. Most of the benchmarks exhibit a false positive rate that is positively related to its fault coverage. Libquantum, which has the lowest fault detection rate due to value noises, also has the lowest false positive rate. This trend implies that the fault coverage and false positive rate could have been affected by the Bloom filter sizes, which might improve if Bloom filter sizes are customized for different benchmarks. In most benchmarks, the stride detector has a higher number of fault coverage per false positive, than that of the value detector. This is especially prominent in the results of gcc, which has relatively low fault coverage but high false positive rate. This phenomenon is most likely due to the instruction partitioning algorithm, which is optimized for the stride detector but not the value detector.



(a) Performance overhead in fault-free execution



(b) Performance overhead breakdown

Figure 6-12: Performance impact on the system

6.6.4. Performance, Area, and Energy Impact

With the evaluation setup described in Chapter 3, we simulated the performance in a fault-free program execution to evaluate the performance impact. We also use McPAT to calculate core power and area, and CACTI to calculate the power and area of the fault detector. The energy overhead is calculated based on the power of different components and the performance overhead.

Performance Impact

On average, the detector incurs 6.4% performance overhead, with the numbers of all benchmarks shown in Figure 6-12 (a). As a comparison, FaultHound [17]'s backend fault detection incurs higher performance degradation (e.g., 9%). The performance improvement mainly attributes to our low false rate.

To better explain the results, we break down the performance overhead in Figure 6-12 (b) into three categories.

- The first category shows the impact of the learning period, which is responsible for half of all performance overhead. As mentioned in Chapter 5, we dedicate 200K instructions for each unique phase to learn the instruction partitions. We dedicate another 40K instructions per unique phase for learning the stride and value patterns. The performance impact of this category is proportional to the number of unique phases, and as a result, gcc experiences the highest performance degradation since it has the greatest number of unique phases within the SimPoint.

- The second category is the overhead incurred during phase transition, which is merely 2% of the total overhead. At a phase transition, the partition information Bloom filters, as well as the stride and value Bloom filters, need to be refreshed. If the ending phase is a newly discovered unique phase, these Bloom filters' content needs to be stored into L3 cache. If the next phase is a repeating phase, the Bloom filter content of the unique phase that is being repeated on needs to be reloaded from L3 into the respective Bloom filters. These Bloom filters has a total size of 6KB per phase. Given an average of 13 unique phases and 74 phase repetition transitions throughout the SimPoint, the overhead of storing and loading these structures are quite small compared to the overall execution time.
- The last category is the performance penalty due to the replay upon a detector signal, which is responsible for nearly half of the overhead. Same as FaultHound [17], we use the lightweight backend replay to recover from faults by adding a delay buffer to the issue queue. Upon a detector signal, the instructions in the delay buffer get replayed in the backend pipeline stages. Thus, the performance overhead of this category is mostly proportional to the false positive rate.

Area Cost

The structures introduced by our fault detection mechanism include near-L1 dedicated cache for unique phase signatures, the search Bloom filter and the counter cache for instruction partition learning, the partition information Bloom filters for storing and searching the partitions, and the stride and value Bloom filters for fault detection. These structures together incur an area cost of 1.28% of processor core area.

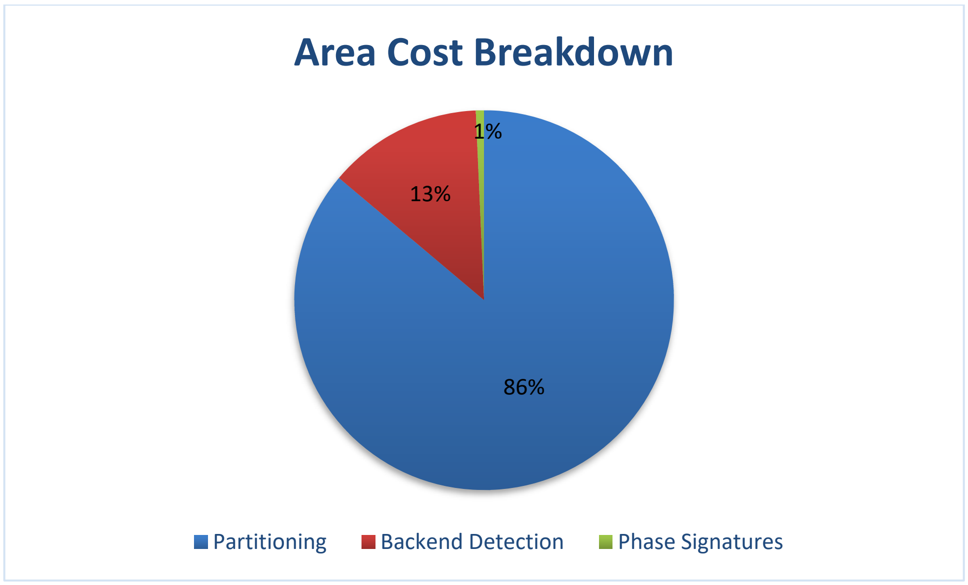


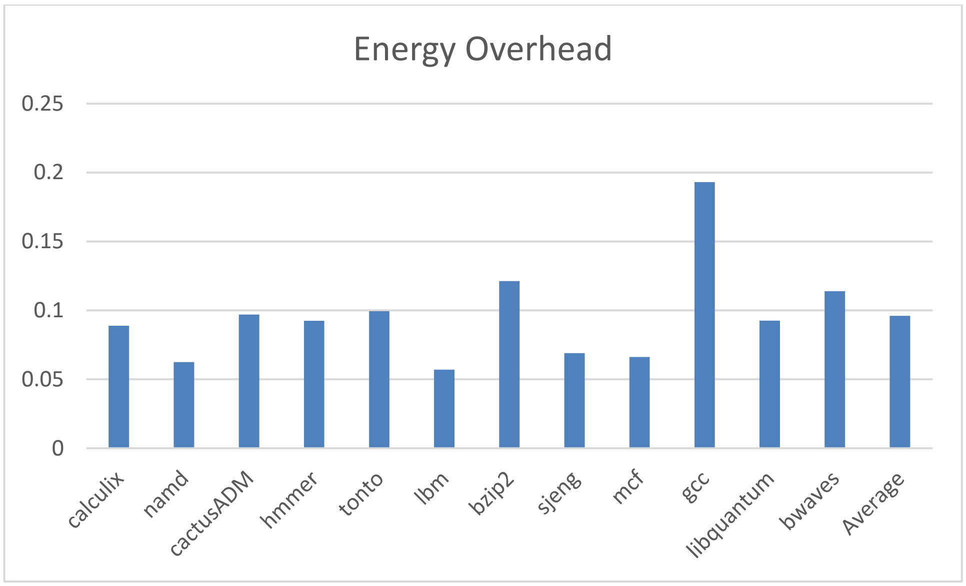
Figure 6-13: Area cost breakdown by purpose

Figure 6-13 shows a breakdown of all structures. The search Bloom filter and counter cache takes up 86% of all newly added area cost. These structures are intentionally designed to be large so that they have as few conflicts as possible, in order to accurately learn the static instruction partitions. This area cost can be justified since accurate instruction partitioning is the foundation of the fault detector's effectiveness.

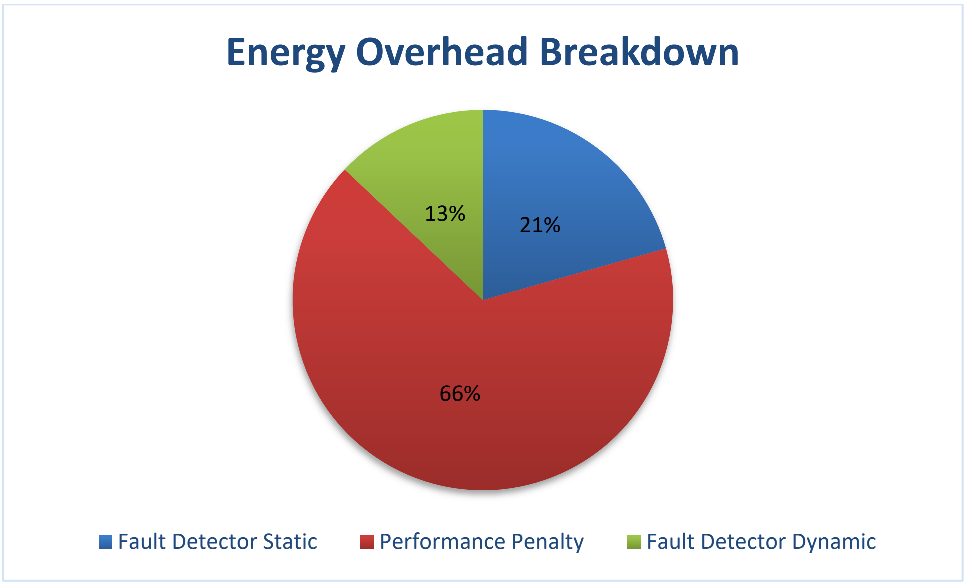
Energy Overhead

Overall, in a fault-free execution, the system experiences 9.8% energy overhead. In comparison, FaultHound [17] incurs ~11% energy overhead for its backend fault detection.

As shown in Figure 6-14 (a), the energy overhead of each benchmark is mostly proportional to the performance overhead shown in Figure 6-12 (a). This is explained by a breakdown by component in Figure 6-14 (b). The energy overhead consists of three major components.



(a) Energy overhead in fault-free execution



(b) Energy overhead breakdown into three major components

Figure 6-14: Energy overhead

- Static power of the newly added structures, which is responsible for 21% of all energy overhead.

- System energy overhead due to performance penalty, which is 66% of the total overhead. This explains why the energy overhead is almost proportional to the performance overhead.
- Dynamic power consumption by the backend fault detection mechanism, including power usage during the learning periods, phase transitions, and Bloom filter lookups for fault detection. This category incurs 13% of all energy overhead since our structures are small and the dynamic accesses

Chapter 7 Front-End Fault Filtering

In this chapter we discuss how front-end faults can be filtered. Front-end faults are differentiated from back-end faults in that they occur in the front-end stages of the processor pipeline, which include fetch, decode, rename, and dispatch. These faults can only be corrected by a full pipeline rollback, but not the back-end replay as described in Section 4.3. Since a rollback incurs significant performance and energy overhead, we want to have a dedicated filtering mechanism for signaling front-end faults, and more specifically, distinguishing a front-end fault from a false positive to avoid unnecessary penalties.

This chapter is organized as follows:

1. We first discuss in Section 7.1 how a front-end fault uniquely impacts program locality. Its unique characteristics are the basis of distinguishing a front-end fault from a false positive.
2. In 7.2, we incorporate the findings into how program locality is represented in our fault detector and presents a detailed filter design.
3. We present the experimental results of our front-end fault filter in Section 7.3. We also discuss the trade-offs between fault coverage and performance/power penalty under different parameter settings.

7.1. Front-End Fault vs. False Positive

Modern processors are vulnerable to transient faults at all pipeline stages. Our value-based fault detector, described in Chapter 6, signals faults when the known program locality is disrupted. For back-end faults, they can be corrected by a back-end replay. For a signal caused by a front-end fault, the replay is not able to correct it given the limited size of the replay buffer (see Section 4.4). For a signal caused by a false positive of the detector, it cannot be "corrected" by the replay, either, since the microarchitectural states will remain the same after replay. Thus, a filtering mechanism should be in place for distinguishing front-end faults from false positives.

Within the front-end pipeline stages, register renaming unit is usually the most area and energy consuming [20-22], and thus is the most susceptive to faults. We focus on detecting rename faults which represents most front-end faults.

Register renaming is a technique that maps architectural registers to physical registers, in order to eliminate false dependencies among instructions and increase instruction-level parallelism. The result of a fault that hits the renaming unit is usually that a physical register lookup gets disrupted and it returns an incorrect physical register index. The incorrect physical register index, if invalid or out of boundary, will cause a program to crash. Otherwise, in a silent data corruption scenario, it results in a read or write to an unintended physical register, and consequently causes a later instruction to use an unintended value.

To detect a transient fault on the renaming unit, we need to observe the anomalies that the rename fault may cause in the behavior of program locality and the outcome of our fault detector. A similar approach to think about front-end fault filtering was used in previous work [17]. As described in Chapter 6, we use parallel Bloom filters to identify known memory stride and value patterns. A rename fault results in an unintended value to be used, and thus has an unintended memory value and stride pattern to be sent to the Bloom filters for fault detection. Same as a false positive from the detector, a detector signal is raised in the case of a rename fault. However, different from a false positive, an unintended value from a rename fault hits the detector at cold locations, i.e. locations that are not recently hit by other detector signals. In contrast, a false positive usually indicates a shift in program execution phases, and the detector location which raises a signal by a false positive can raise a signal repeatedly during the phase shift.

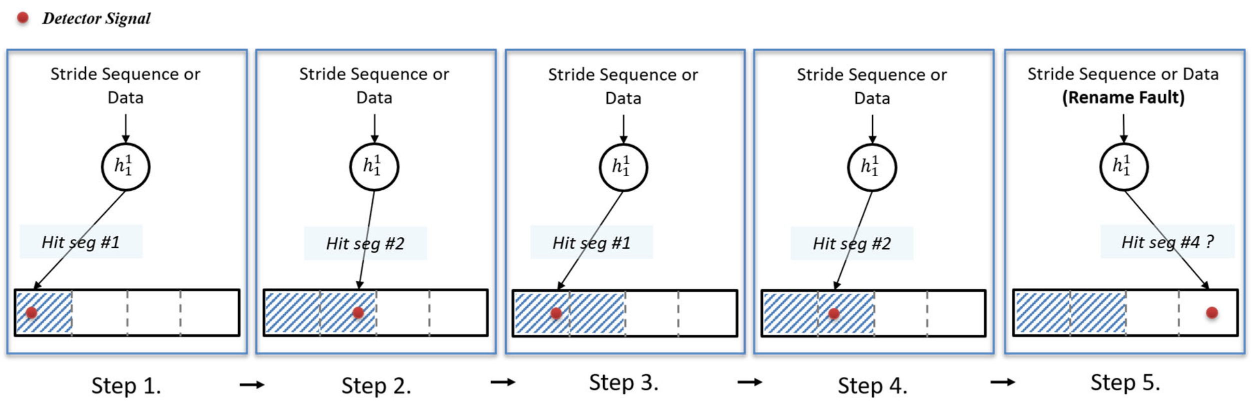


Figure 7-1: A rename fault's impact on the outcome of our detector

The detector behavior in the case of a rename fault is shown in Figure 7-1. The parallel Bloom filter's bit vector is divided into 4 segments to indicate the locations where a signal is raised. In Step 1 through 4, detector signals were raised in segment #1 and #2 repeatedly, which indicates a phase transition. In Step 5, an unintended data pattern resulted from a rename fault causes a detector signal in segment #4.

With this observation, we use a biased squash state machine to monitor segments of the Bloom filter. If a segment has not seen a detector signal in the past th times of any detector signals, and sees a signal in the next time, this signal will be filtered as front-end fault. Repeated signals on a segment will be filtered as false positives and suppressed.

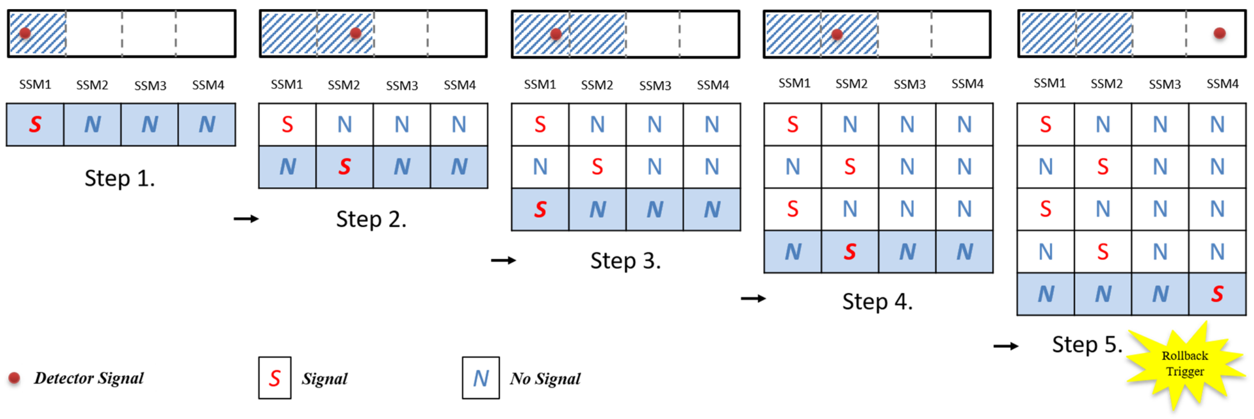


Figure 7-2: Using biased squash state machines for rename fault detection

An example of how a rename fault is filtered, and thus a rollback is triggered, using the previously described approach is shown in Figure 7-2. Each step corresponds to its counterpart in Figure 7-1. The parallel Bloom filter is divided into 4 segments. For each segment, a squash state machine (SSM) is installed. The SSM monitors whether a signal has been raised by the detector against this segment. Assuming the trigger threshold $th = 4$, segment #4 triggers a rollback when the 5th global signal is raised against it, since there has been 4 consecutive no-signals before a signal. Segment #1 and segment #2 does not trigger a rollback at the detector signals raised against them since their consecutive no-signals do not meet the threshold. Instead, they suppress the signals as false positives.

The state transition and rollback-triggering can be represented by the state machine design in Figure 7-3. This is a biased state machine which triggers a rollback or pipeline squash when a detector signal is filtered to be a front-end fault. Each Bloom filter segment is assigned one such squash state machine (SSM), as shown in Figure 7-3.

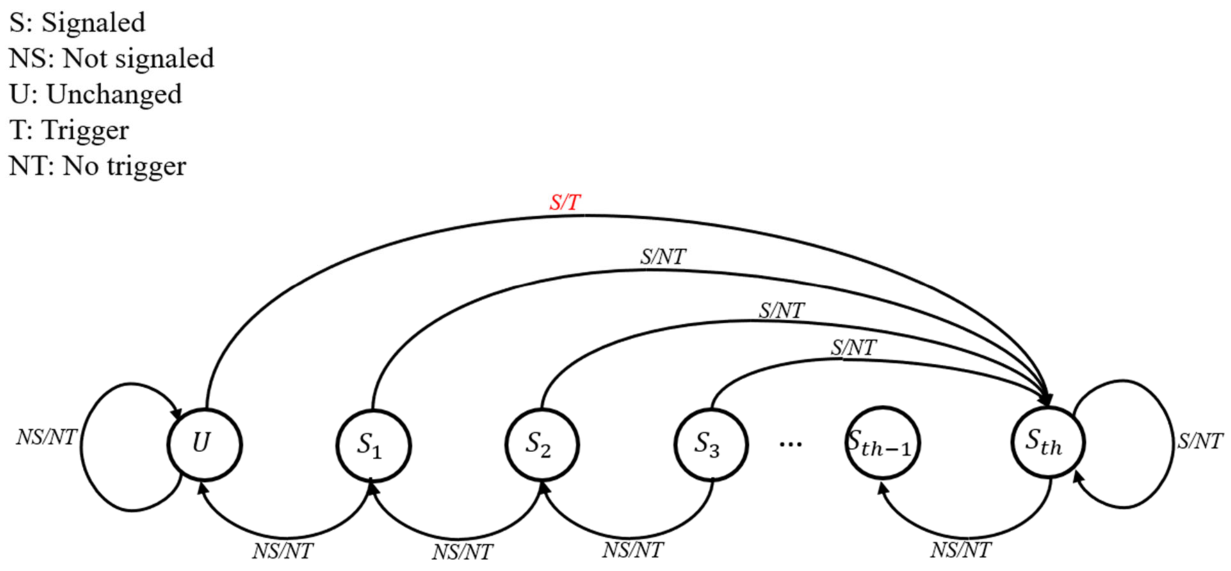


Figure 7-3: The state machine that filters detector signals and triggers rollback for front-end faults

7.2. Front-End Fault Filter Design

Due to the high performance and energy penalty of a rollback, the key design constraint for the front-end fault filter is to generate as few false positive as possible.

The false positive rate of the front-end fault filter is determined by the probability of each single SSM, and thus each parallel Bloom filter, issuing a trigger. We use the following parameters and equations for calculating a projected form of the false positive rate.

The parallel Bloom filter, the SSMs, and some parameters are illustrated in Figure 7-4. Let the parallel Bloom filter length, in bits, be L, the number of squash state machines per parallel Bloom filter be S, and the threshold of consecutive detector signals before a trigger be th. The number of bits covered by one squash state machine will thus be $\frac{L}{S}$.

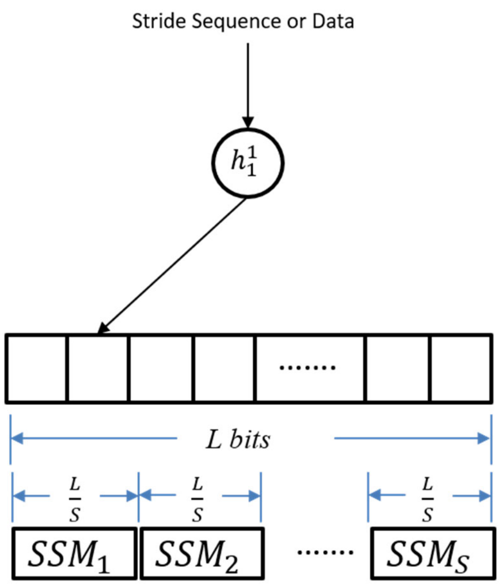


Figure 7-4: Biased squash state machine for one parallel Bloom filter

Assuming the probability of signal is uniformly distributed among all bits of a parallel Bloom filter, the probability of one bit within the parallel Bloom filter to be signaled by the detector is:

$$P_b = \frac{1}{L}$$

The probability for one SSM segment within the parallel Bloom filter to be signaled is:

$$P_s = 1 - (1 - P_b)^{\frac{L}{S}}$$

Based on our design, the probability for one squash state machine to trigger a rollback is

$$\text{Equation 7-1: } \mathrm{P_{tr}} = (1 - \mathrm{P_s})^{\mathrm{th}} \cdot \mathrm{P_s}$$

Or:

$$P_{tr} = \left(1 - \frac{1}{L}\right)^{\frac{L}{S} \cdot th} \cdot (1 - \left(1 - \frac{1}{L}\right)^{\frac{L}{S}})$$

The probability for one parallel Bloom filter to get a rollback trigger from any of its associated SSMs is:

$$\text{Equation 7-2: } \mathrm{P_{tr\_pbf}} = 1 - (1 - \mathrm{P_{tr}})^{\mathrm{S}}$$

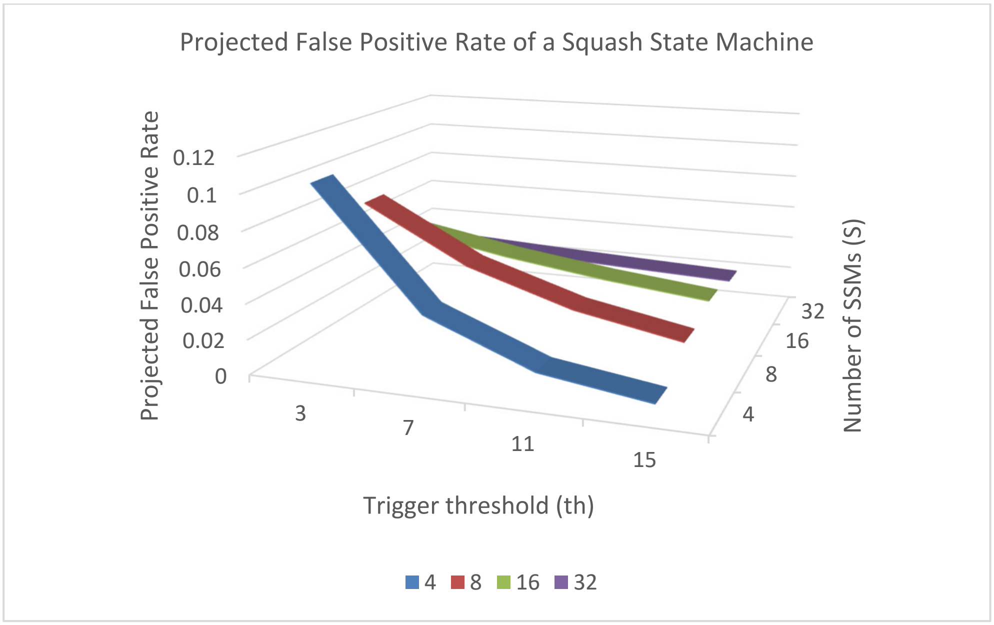


Figure 7-5: False positive rate of a squash state machine as a function of the number of SSMs and the trigger threshold.

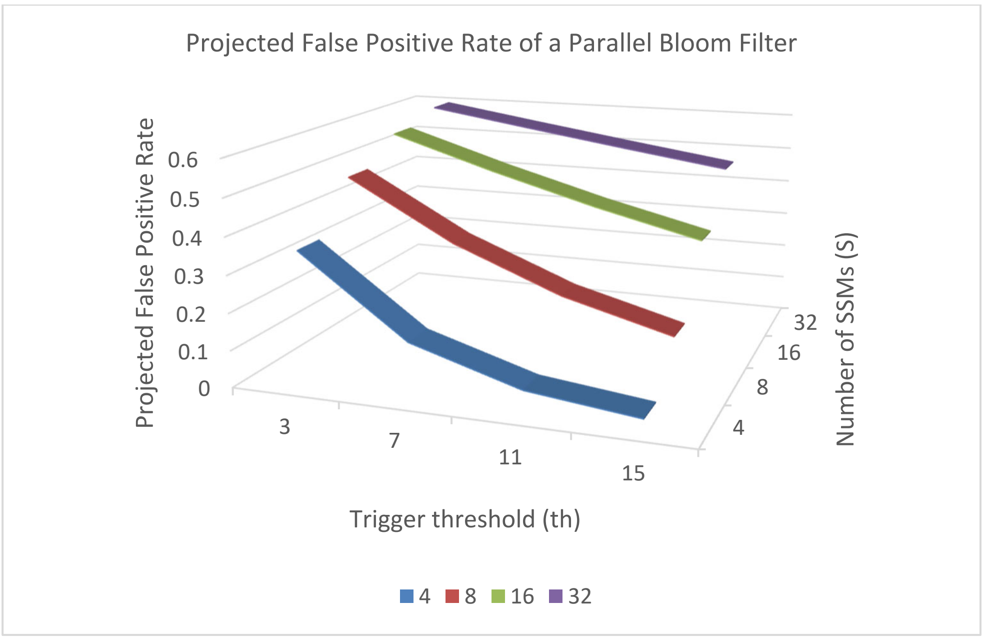


Figure 7-6: False positive rate of a parallel Bloom filter as a function of the number of SSMs and the trigger threshold.

Equation 7-2 gives us a guideline on choosing our parameters for the front-end fault filter. The expanded form of this formula is rather complex, so we do not provide it here. However, within the range of possible values for our implementation, Figure 7-5 and Figure 6 show how the projected false positive rate changes over the two main parameters, the number of SSMs per parallel Bloom filter and the trigger threshold, i.e. the number of no-signals before a signal to be filtered as a trigger. The parallel Bloom filter size L is set to be 1024 bits in the figures, although we did the same calculation with L being 2048 and 4096 respectively and found that the trends are almost identical.

From Figure 7-5, the projected false positive rate of one squash state machine decreases as the trigger threshold increases, no matter what value S, i.e. the number of SSMs used for one parallel Bloom filter (PBF), is. This is expected since in Equation 7-1, $(1 - P_s)^{th}$ decreases as th

increases. With 4 SSMs per PBF, the false positive rate decreases faster, from 10.5% when $th = 3$ to 0.5% when $th = 15$. With 32 SSMs per PBF, the false positive rate decreases slower, from 2.8% when $th = 3$ to 1.9% when $th = 15$. It may seem that for the different S values, we can find a combination of parameters to achieve a low false positive rate. However, we cannot solely depend on the FP rate of one single SSM for choosing the parameter values of the filter, since one parallel Bloom filter is consisted of multiple SSMs and the probability needs to be calculated using Equation 7-2.

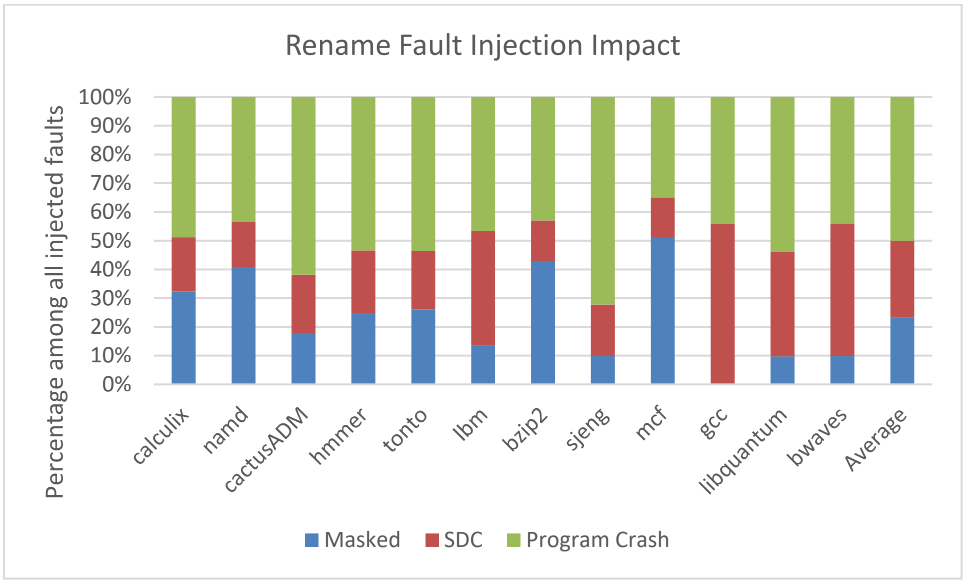
From Figure 7-6, the projected false positive rate of one parallel Bloom filter still decreases as the trigger threshold increases. However, as the number of SSMs per PBF increases, the false positive rate increases significantly. Thus, to achieve a low false positive rate for one parallel Bloom filter, we need to have a small S value and a high th value.

7.3. Experimental Results

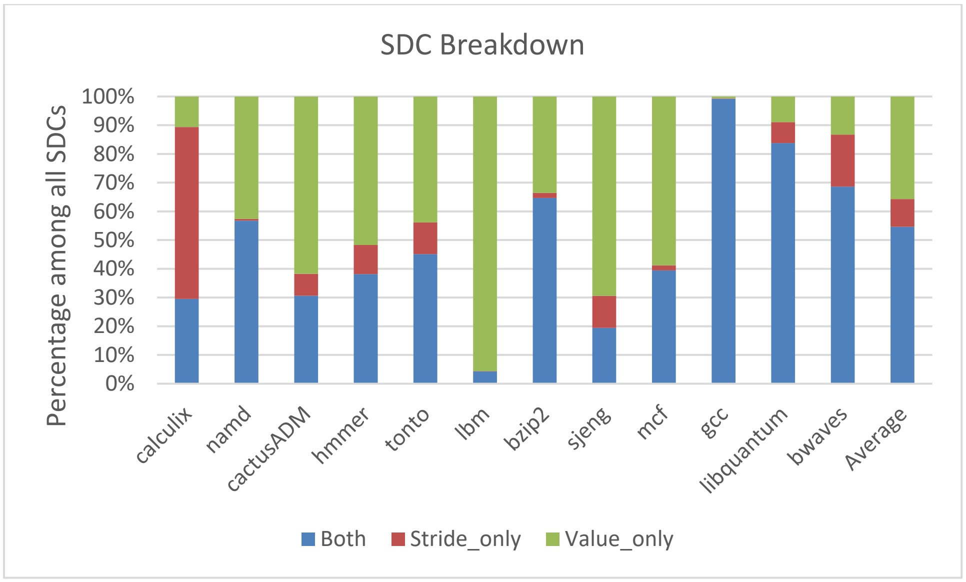
In this section we present the experimental results fault coverage, false positive rate, and the incurred performance and energy overhead.

7.3.1. Fault injection

In our experimental setup, we injected 2000 single-bit faults into the rename stage of an out-of-order CPU. 12 of the SPEC CPU2006 benchmarks were run to evaluate the fault injection impact. Out of all the injected faults, we categorize the outcome of an injection into three categories: Silent data corruption (SDC), program crash, and unchanged behavior. Figure 7-7 shows the breakdown of the outcome, as well as its specific impact on memory address stride and memory access value.



(a) Rename fault injection broken down by impact on program execution



(b) Rename fault injection broken down by impact on stride and value

Figure 7-7: Rename fault injection outcome

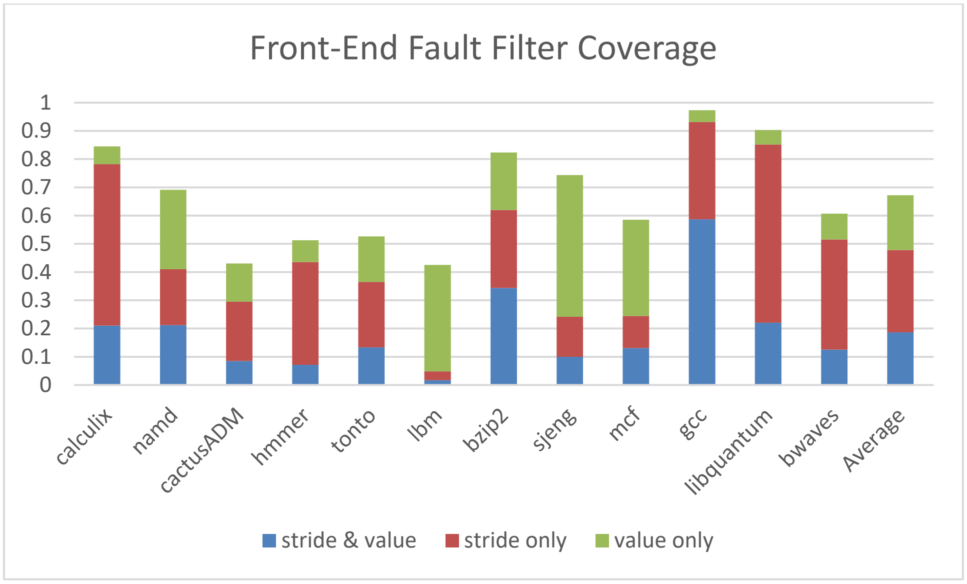
As shown in Figure 7-7 (a), about 50% of the rename faults resulted in a program crash. For the remaining faults, we need to distinguish SDCs from masked ones. Same as the backend fault injection, we monitor at least 5000 instructions following the fault injection. The dynamic memory instructions, memory strides, and memory values are compared to those of a fault-free execution. If a divergence is not found after the monitored length, the fault is considered masked. Out of all injected faults, 27% are masked. The remaining 23% faults are silent data corruptions (SDC), for which we will report fault coverage and false positives.

The stride trace and the value trace show slightly different sensitivity to the rename faults. From Figure 7-7 (b) which gives a breakdown of all the SDCs by their impact on strides and values. About 10% has changed only the stride trace, 36% has changed the value trace, and the remaining 54% has changed both the stride and the value traces. Since either of stride detector's or the value detector's filters can trigger a rollback, this breakdown gives us a fertile ground to capture front-end faults.

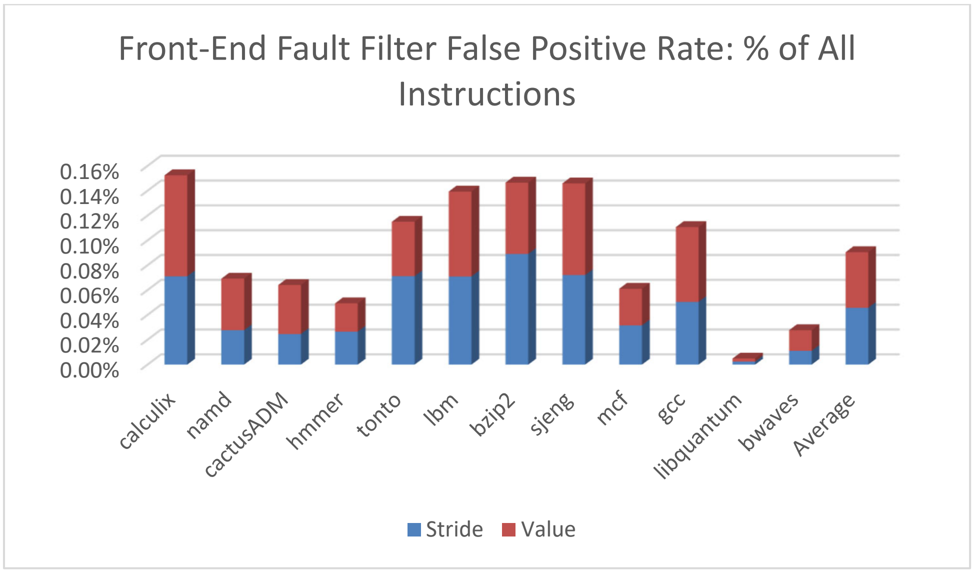
7.3.2. Fault Filter Outcome

Our front-end fault filter is evaluated using 4 SSMs per parallel Bloom filter, and a trigger threshold of 15. For all SDCs resulted from the rename faults injection, the fault coverage, as well as the false positive rate from the stride-based detector's filter, the value-based detector's filter, and both combined, are studied.

Fault Coverage



(a) Fault coverage of the front-end fault filter, $S = 4$, $th = 15$



(b) False positive rate of the front-end fault filter, $S = 4$, $th = 15$

Figure 7-8: Fault coverage and false positive of the front-end fault filter

The front-end fault filter's coverage rate for all benchmarks are shown in Figure 7-8 (a). On average, the fault coverage is 67% of all rename SDCs.

The front-end fault filter yielded relatively high coverage on most of the benchmarks. Especially, gcc and libquantum shows >90% fault coverage. We observed one main reason for this phenomenon. A significant portion of the coverage comes from the filter associated with the stride-based detector. These benchmarks have a larger number of faults that altered the stride trace, and as discussed in Chapter 6, the stride detector is more optimized than the value detector for fault detection. The same applies to calculix.

On the other hand, cactusADM and lbm exhibits relatively low fault coverage. This could attribute to one or a combination of two reasons:

- Within the parallel Bloom filter, all segments are frequently hit, thus an error can be left unfiltered.
- The fault detector uses only 1 value (compared to 2 strides) as the data input for the Bloom filters. This may result in additional difficulty for an unintended value to be filtered.

Since the frontend is responsible for 20% of processor core pipeline area, this translates into 13.6% coverage of all faults. Combined with the backend fault detector, our fault detection mechanism shows a total fault coverage of 80.8%. In comparison, FaultHound [17] has a total fault coverage of 75% on SDCs.

False Positives

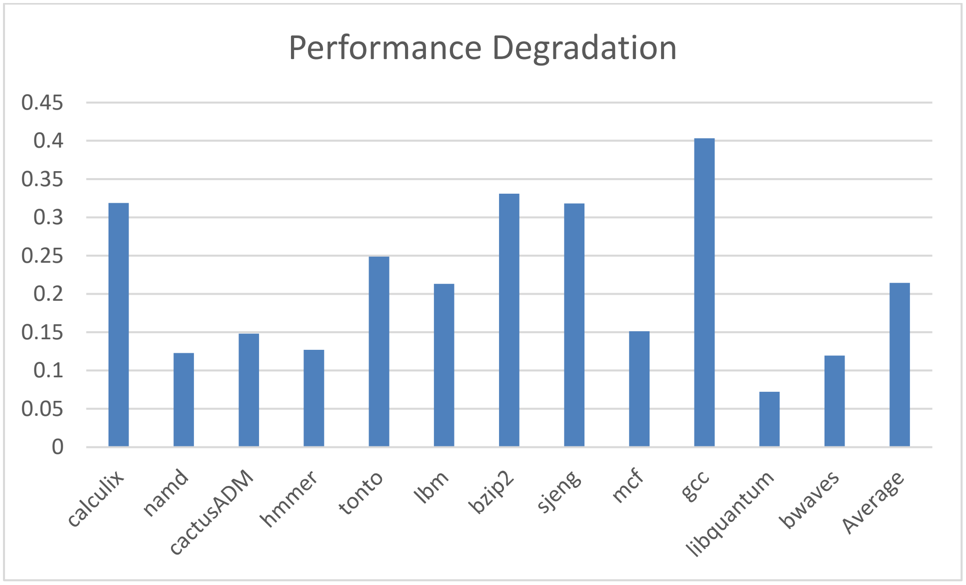
The false positive rates of all benchmarks are shown in Figure 7-8 (b). On average, the front-end fault filter incurs a false positive of 0.09% among all instructions. The stride and value filter each are responsible for about half of the false positives. Benchmarks like calculix and bzip2 show relatively high false positive rate. Others, like libquantum, incurs extremely few false positives. The false positive rate is affected by a combination of factors, mainly the working set size of memory strides and values, as well as the occupancy rate of the Bloom filters, i.e. the percentage of bits set to 1 within the Bloom filters. One additional observation with libquantum is that, with the results shown in Chapter 6, libquantum was among the ones with lower backend coverage, which could be due to the Bloom filter size being smaller than ideal for its backend detector to signal a fault. This becomes an advantage for front-end fault detection. Since there are fewer backend detector signals, the state machines restore to the initial state less often, creating less chance for front-end fault filter false positives.

7.3.3. Performance and Energy Impact

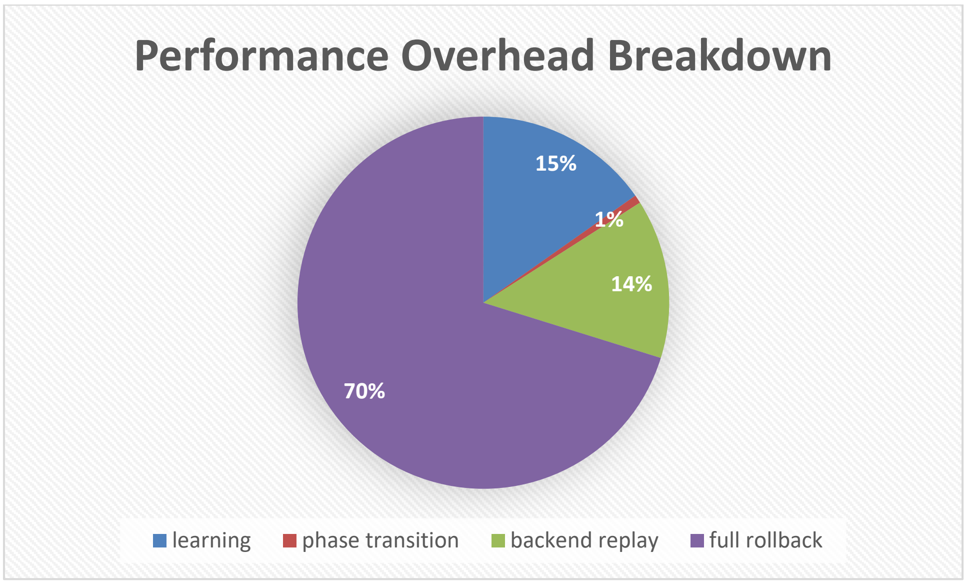
Recovery penalty	Full pipeline rollback (~100 instructions)
Front-end squash state machines	128 (stride), 128 (value), 4 bits each

Table 7-1: Hardware configuration for the front-end fault filter

With the evaluation setup described in Chapter 3, we simulated the performance in a fault-free program execution. The hardware parameters are shown in Table 7-1. Since the 4-bit state machines are much smaller than the caches and Bloom filters, we did not model the area or power for these state machines but focus the discussion on the performance and energy overhead incurred during rollbacks.



(a) Performance overhead in fault-free execution



(b) Performance overhead breakdown

Figure 7-9: Performance overhead relative to baseline system

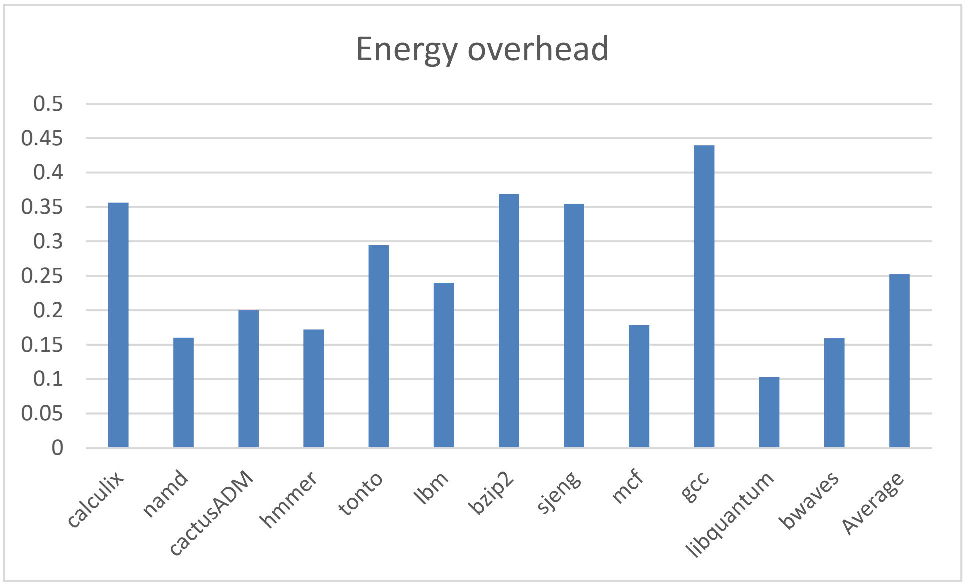
Performance Impact

Overall, the detector incurs 21% performance overhead when detecting both backend and frontend faults, as shown in Figure 7-9 (a). This more than doubled the backend detector-only performance overhead discussed in Chapter 6. The increase is mainly due to the high rollback penalty of false positives, as shown in the breakdown in Figure 7-9 (b). When the front-end filter detects a front-end fault or a false positive, a full pipeline rollback is triggered to recover from the rename fault. The rollback squashes all previous instructions in the pipeline, which usually results in the re-execution of ~100 instructions in an out-of-order issue processor [16, 17].

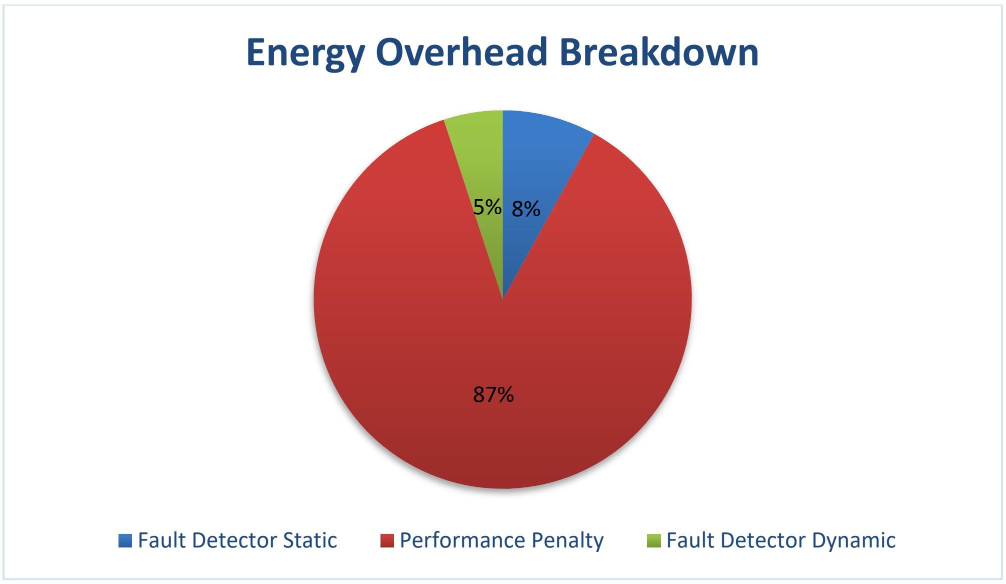
The performance overhead far less than what PBFS [16] has (e.g., 100%) since PBFS triggers a full pipeline rollback for every perturbation. However, our overhead is higher than what FaultHound [17] reported (e.g., 10%). By comparison, we do have the learning periods that FaultHound does not have, which is responsible for 3% of overall performance degradation. The rest of the difference is mainly due to discrepancies in simulation techniques and workloads. Beside the SPEC2006 benchmarks, FaultHound used additional workloads that are memory-intensive, whose high cache miss rate helps hide part of performance penalty incurred by the full pipeline rollbacks and results in lower performance overhead.

Energy Overhead

Overall, in a fault-free execution, the system experiences 25% energy overhead as shown in Figure 7-10 (a). Compared to the backend-only fault detector's 9.8% energy overhead, the front-end fault filter added another 15% overhead in energy consumption. The energy overhead is proportional to the performance overhead, since the majority of the overhead comes from full pipeline rollbacks incurred during false positives of the front-end fault filter as detailed in Figure 7-10 (b).



(a) Energy overhead in fault-free execution



(b) Energy overhead breakdown

Figure 7-10: Energy overhead relative to baseline system

Our energy overhead is far less than that of PBFS (e.g., 100%) and is the same as what FaultHound reported (e.g., 25%). FaultHound’s energy overhead could not be hidden even though the performance cost was reduced by workload characteristics.

Chapter 8 Efficient Parameter Variation Sampling for Architecture Simulations

8.1. Introduction

In both architecture and design automation communities, large scale Monte Carlo simulations are widely used to investigate the probabilistic impacts of manufacturing variation [44]. These variations follow complex, random behavior and influence the behavior of circuits and architectures in profound manner, limiting the applicability of analytical models and steering researchers toward Monte Carlo simulation. Typically, a single Monte Carlo experiment consists of generating hundreds or thousands of random parameter variation scenarios and simulating either a circuit or processor design under each of those scenarios. However, the total simulation cost for many parameter variation studies can be enormous. In the realm of architecture, each Monte Carlo simulation would require running a detailed architecture simulator for anywhere from one hundred million to one billion instructions – a task which may take hours. Given that most recent studies in the architecture community may incorporate ten or more individual benchmark programs [45]– [47], the full set of Monte Carlo simulations may require thousands of compute-hours.

If left unaddressed, the burdensome architectural simulation time associated with parameter variation studies may have adverse impacts. At the risk of reducing simulation accuracy, researchers may choose to use fewer Monte Carlo samples, simulate a smaller window of program execution (e.g. 10 million instead of 100 million instructions), select a faster but cruder and less detailed simulator model, or subset the benchmark suite. As previous work has shown (e.g., [48], [49]), corner-cutting in the name of reducing simulation time can have disastrous effects on accuracy of architecture studies and in some extreme cases may draw researchers to incorrect conclusions.

While there have been successful attempts to reduce Monte Carlo simulation time in the circuit domain, these approaches cannot directly be applied to architecture [50]. The circuit approaches attempt to reduce the simulated samples while retaining the same statistical properties. In particular, circuit-level studies assume knowledge of circuit structure and model variation at the gate level while architecture studies are at a much higher semantic level and investigate designs with billions of gates. We address this problem by bridging this semantic gap and making the approach scalable to architecture.

At the heart of the motivation for this work is the observation that significant gains in the efficiency of variation-aware architecture simulation can be achieved if better sampling methodology for parameter variation is accommodated. Specifically, we postulate that we can reduce the number of samples needed to achieve statistically sound results if we use sequences that are guaranteed to give faster convergence than Monte Carlo. To do this we must bridge a gap in understanding between circuit and architecture. We adapt several existing circuit-level techniques to make them suitable for this domain and introduce several novel approaches that further improve simulation efficacy. The main contributions of this paper can be summarized as follows:

A. Adapting Low-Discrepancy Sampling Methods to Architectural Simulation: Low-Discrepancy (LD) sampling techniques generate quasi-random samples defined to have lower integration errors than true Monte Carlo sequences [51]. By implementing low-discrepancy techniques into variation map generation, the sample space of parameter variation can be covered by fewer samples relative to Monte Carlo sampling approaches. This efficient sampling methodology leads to large reduction in architectural simulation time.

B. Introducing Multi-Resolution Grid Maps: To better represent sensitive geographic regions of the chip, we divide it into a non- uniform grid. For processor components that are more sensitive to the parameter variation, we assign a finer grid resolution, and apply coarser granularity to those components which are less sensitive. In total, we improve the effectiveness and efficiency of the parameter variation representation, while maintaining the same overall complexity of representation.

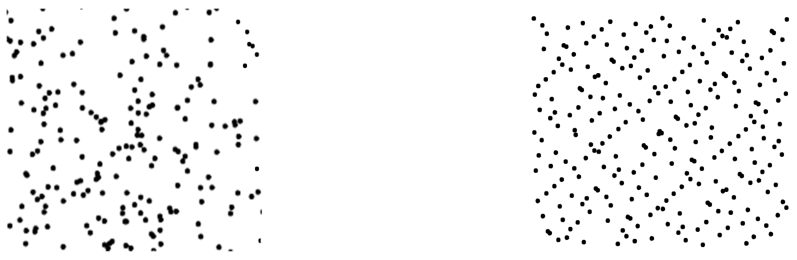
C. Comprehensive Experimental Evaluation: We implement and evaluate the proposed methodologies. Our results demonstrate that for the selected microarchitecture level timing error and leakage power estimation, the low-discrepancy sampling and multi-resolution grid model give at least 3.3× faster convergence than Monte Carlo sampling.

The rest of this paper is structured as follows. In Section II we recollect some necessary background for understanding the proposed techniques, which we elaborate upon in Section

III. Section IV details the quantitative benefits of our work by presenting the results of our experimental evaluations. Section V concludes the article and outlines directions for future work.

8.2. Background

Predicting the impact of manufacturing variation on circuit and architecture designs has become a challenging and increasingly important task for several reasons [44], [52]. The fabrication process introduces prominent variations to the threshold voltage, V_{th}, and the effective gate length, L_{eff} of transistors [44], [53], [54]. These parameter variations include both true random components which are independent and systematic components that are a function of the chip geometry and exhibit complex correlation patterns [55]. Modeling and simulation approaches must correctly account for the way that the parameter variation impacts circuit delay while capturing the spatial correlations.



(a) Standard Monte Carlo (b) Quasi-Monte Carlo

Figure 8-1: Comparing 2D sequences generated with standard Monte Carlo and Low-Discrepancy techniques. The two examples have an equal number of points

Due to the probabilistic nature of manufacturing variations, and the complex interactions between transistor parameters, stochastic methods including Monte Carlo(MC) experiments based on repeated trials have become powerful tools for studying the consequences of parameter variation and developing architectural and circuit innovations to counter them [46], [56]. At a high-level, the approach consists of generating two-dimensional fields which represent random parameter variation which obey the before mentioned statistical properties and then running detailed simulation for each one of these scenarios. For gate- level Statistical Static Timing Analysis (SSTA), a natural way to model spatially correlated parameter variation is with a correlation matrix which captures the statistical relationship between every pair of transistors in the circuit [50]. Many random parameter fields can then be generated using this correlation matrix as a starting point. In the case of SSTA, SPICE simulations are run with each field sample. Since the total number of samples needed to guarantee convergence can be quite large, the number of MC trials becomes the biggest factor in simulation run time. At the circuit-level, some innovative sampling techniques have been able to drastically reduce this factor and improve simulation runtime. Singhee *et al.* [50], [57] recognized that with conventional random field generation, Monte Carlo techniques require many samples to guarantee convergence because its accuracy obeys a $O(n^{-0.5})$ proportionality with sample set size n [58]. They further noticed that comparing to true random sequences, some classes of Quasi-Monte Carlo sequences with the same number of samples have better coverage for the sample space, hence give faster convergence. In particular, *Low-Discrepancy* (LD) methods are known to generate high quality deterministic patterns that are guaranteed to give approximately $O(n^{-1})$ [51] convergence, a significant improvement over random sequence. Figure 8-1 illustrates the difference between the coverage natures of conventional Monte Carlo points and low- discrepancy points in a two-dimensional space. The conventional Monte Carlo samples show both clusters and sparse regions while the low-discrepancy samples give much better coverage of the space. One can imagine extending this concept to higher dimensional spaces where each dimension might represent a physical factor (e.g. V_{th} for each transistor in a circuit).

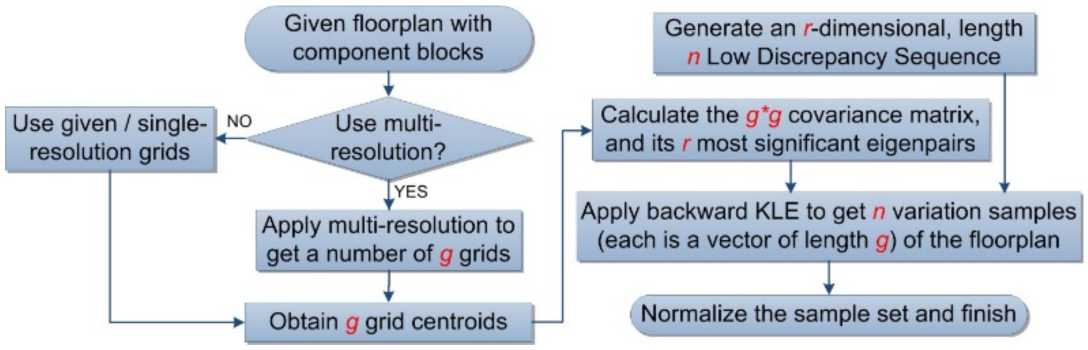

Given floorplan with component blocks
Use multi-resolution?
NO
Use given / single-resolution grids
YES
Apply multi-resolution to get a number of g grids
Obtain g grid centroids
Generate an r-dimensional, length n Low Discrepancy Sequence
Calculate the g*g covariance matrix, and its r most significant eigenpairs
Apply backward KLE to get n variation samples (each is a vector of length g) of the floorplan
Normalize the sample set and finish

Figure 8-2: Process flow for generating n LD variation maps

However, low-discrepancy sequences alone cannot replace Monte Carlo sequences in generating parameter variation samples for even small circuits. For a design with n gates, one would need to generate a low-discrepancy sequence with dimensionality of n. Current best low-discrepancy sequence generators offer practical advantage over standard Monte Carlo sequences only in the early r dimensions ($r \leq 12$ [59]). Consequently, for efficient parameter variation modeling of circuits, we apply the Karhunen-Loeve Expansion (KLE) [60], a model simplification technique similar to Principle Component Analysis (PCA) [50]. Recall that a correlation matrix can be used to represent gate- level variations across the chip. This serves as a very precise description for high-dimension model of the chip. The first r ($r \sim 25$) components of KLE, composed of the r-dimensional random (or quasi-random) sequence and the r most significant eigenpairs of the correlation matrix, is an accurate estimate of KLE [50]. This effectively reduces a large number of correlated variables – in this case transistor parameters which are geometrically correlated –

into a much smaller number of values and hence lower dimensionality. With a drastically reduced representation of the parameter variation, low-discrepancy techniques can be safely applied to reduce the number of required samples.

8.3. Variation Modeling and Sampling

As described in the previous section, Quasi-Monte Carlo sampling methods have been applied to accelerate gate-level SSTA simulations under parameter variation [50], [57] in the circuit domain. Designs with $\Omega(10^4)$ gates are evaluated for these studies, where spatial correlations of these gates can be captured in a correlation matrix of tractable size. However, these techniques do not directly scale to architectural simulations for a few reasons. First, gate-level descriptions of most modern processors are unavailable for academic researches where they are obviously available for circuit-level designs. Second, even if a complete processor could be modeled at the gate-level, the netlist of the design, which may contain hundreds of millions to billions of gates, exceeds the capacity of existing gate- level algorithms which have $O(n^2)$ spatial complexity for n gates. Finally, most computer architects work on a higher and more abstract level, and architectural simulations aim for more complex and comprehensive evaluations for the system. For example, recent work at the architecture level has examined whole-chip leakage power and timing error rates as functions of parameter variations [44]. These studies must include program state and microarchitecture-level models that are fundamentally different from transistor-level simulations in SSTA.

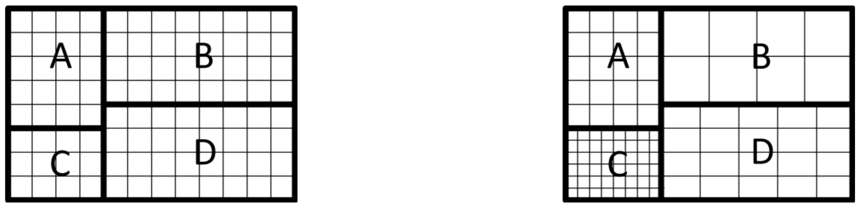
To address these challenges, our proposed techniques have special considerations for architectural variation simulations. First, instead of gate-level, we model parameter variations at block/grid level. Grid size here poses a tradeoff between the computational complexity and the modeling accuracy. Second, within the processor each structural block will have its own susceptibility to and distinct behavior under parameter variation. We leverage the fact that some components may have a greater overall impact on the system than others and introduce *multi-resolution modeling* of parameter variation. Figure 8-2 gives a process flow for generating *n* LD samples. In the following section, we first demonstrate how to model a block/grid variation map with Quasi-Monte Carlo methods. Then, we discuss algorithms that generate grid structures with the best accuracy- complexity tradeoffs.

A. Compact Systematic Variation Representations

Our parameter variation modeling approach assumes a high- level physical model for microarchitectural components nominally described via a floorplan. Depending on the application, one may choose to model structures within a single processor pipeline, or cores and caches in a many-core chip. Given this floorplan, we represent the physical variation of parameters such as L_{eff} and V_{th} for diverse usages and abstractions. Either block-based variation model is applied, where we assume the parameter within each component is a constant and use its centroid for correlation calculation, or we further decompose the blocks into regular grid regions and generate variation samples with finer granularity. Note that, although block level models may lose some accuracy comparing to grid level models, they are still acceptable models for certain architectural study [44].

B. Implementing Quasi-Random Samples

Figure 8-2 shows how to generate our Quasi-Monte Carlo samples. First, given the block floorplan and grid resolution, the correlation matrix is calculated for KLE decomposition. To maintain consistency with [44], here the matrix concerns purely the covariance factor between grid regions. This differs from the circuit-level approach in [50] where the correlation factors are normalized by grid area. Second, there are many possible methods for constructing LD sequences. We select Niederreiter's sequence, which has been proved to have less integration error [61] than Sobol's sequence which was used by [50], [57]. This LD sequence is then combined with KLE to generate an original set of systematic parameter variation samples. Finally, before "publishing" the sample set, we adjust the set to improve the sample space coverage. Systematic variation is supposed to have a statistical mean μ_{var} of zero and a specified standard deviation σ_{var} (according to this parameter's given μ and σ/μ [44]). For each block/grid region i, we apply linear normalization to its variation values across all samples, so that $\mu_{var,i} = 0$ and $\sigma_{var,i} = \sigma_{var}$. After doing this, the KLE-based LD variation sample set are well positioned within the targeted statistical range.



(a) Single-Resolution grids (b) Multi-Resolution grids

Figure 8-3: The illustration of SR and MR grids distributed over a 4-block floorplan. Both figures are with the same number of grids

C. Enhancing Localization with Multi-Resolution Analysis

We make another observation relevant to microarchitectural parameter variation studies, namely that some components of the processor are known to be more sensitive to variation than others.

In this paper, we apply this to evaluate two important architecturally relevant component properties that are strong functions of parameter variation: timing error rate P_e and leakage power P_{leak}. In an era where architects are considering timing speculation as a way to improve performance and efficiency, timing error rates are important properties of a design [45], [46], [56]. In deep submicron technology, leakage power comprises a significant portion of total chip power and therefore serves as an essential design characteristic.

We first consider P_e, modeling an n-stage pipeline as a series failure system. The total P_e can be represented as a weighted summation of the error rate of each pipeline stage i:

Equation 8-1: $P_e = (\sum_{i=1}^{n} \alpha_i \times P_{e_i})$

□$_i$ is the activity factor of block i. Intuitively, pipeline stages which have either high activity factors or error rates P_{e_i} are more likely to produce timing errors and will have a greater impact on total error rate. Activity factors are a strong function of program characteristics (e.g. floating-point applications with have high activity factors for their FP execution units while integer programs will not) and in many cases activity magnitudes can be predicted before simulation. We now consider P_{leak}. Chip- wide leakage power can be seen as the integration of the leakage power of each component i:

Equation 8-2: $P_{leak} = \sum_{i=1}^{n} P_{leak_i}$

Leakage for a component depends on both the temperature of that block and its area. Since area is known a priori and temperature is dependent on activity, we can reasonably ascertain which component blocks are likely to be dominant. As two of the more important characteristics of a processor under parameter variation, both of timing error rate and leakage power are in the form of $f = \sum_{i=1}^{n} f_i$. Let f_0 and f_{i0} denote the true values of f and f_i to optimize the estimation of f, we need to minimize the estimate error ε:

$$\text{Equation 8-3: } \varepsilon = \frac{|f - f_0|}{f_0} = |\sum_{i=1}^{n}(f_i - f_0)/f_0| = |\sum_{i=1}^{n}(\varepsilon_i \times f_{i_0}/f_0|$$

Equation 8-3 implies that for blocks with larger f_{i_0}, the estimate error ε_i needs to be smaller to minimize the total error. Hence in this work, we introduce *Multi-Resolution* (MR) variation sampling, in which the on-chip parameter map is composed of blocks with varying grid density. The total number of grids points G are distributed to each block i following the rule

$$\text{Equation 8-4: } \mathrm{G_i} = (\frac{\mathrm{f_{i_0}}}{\mathrm{f_0}}) \times \mathrm{G},$$

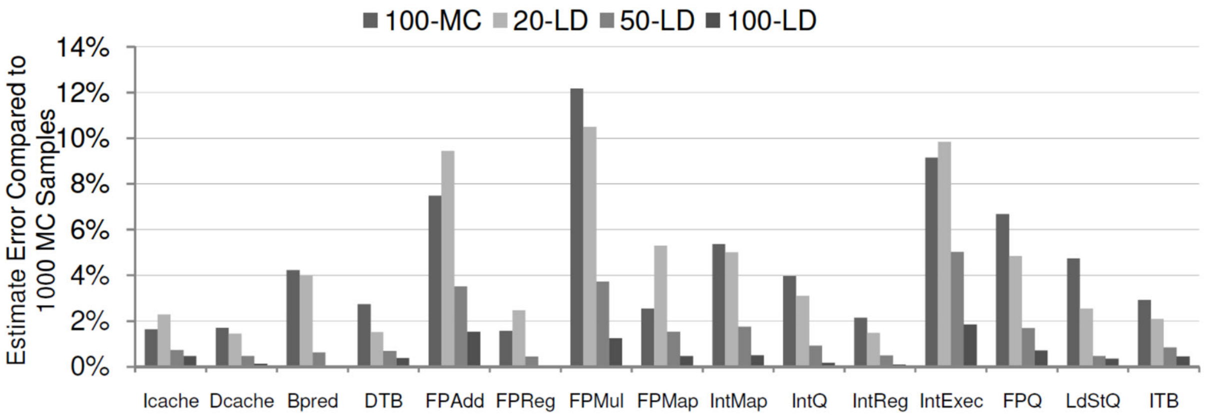


Figure 8-4: The estimate error of P_e relative to 10,000 MC samples: for 15 cpu blocks, 100 MC samples, 20 LD samples, 50 LD samples and 100 LD samples

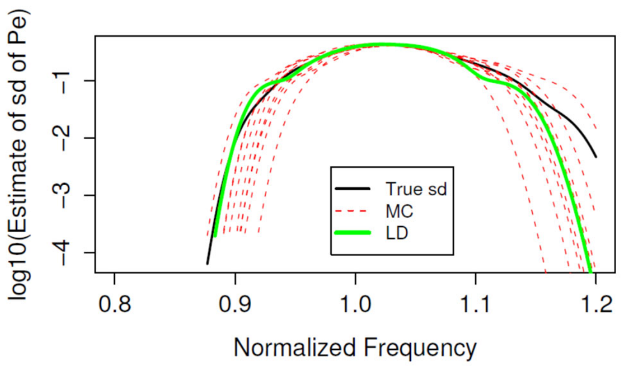
which intuitively means that the grid density within one block is proportional to the "function" density within it, which we can obtain from nominal empirical results. Figure 8-3 illustrates this idea with both Single-Resolution (SR) and MR grids, where block C has the greatest density of the targeting function and block B has the least. As experimental results show, for identically sized parameter maps, MR samples converge faster than SR samples. We conclude this section with a note that, combining the LD and MR techniques, generating 1,000 samples typically takes several seconds to a few minutes on a standard Linux desktop system. The sample generation time is therefore negligible when compared to the detailed simulation time which follows.

8.4. Evaluation

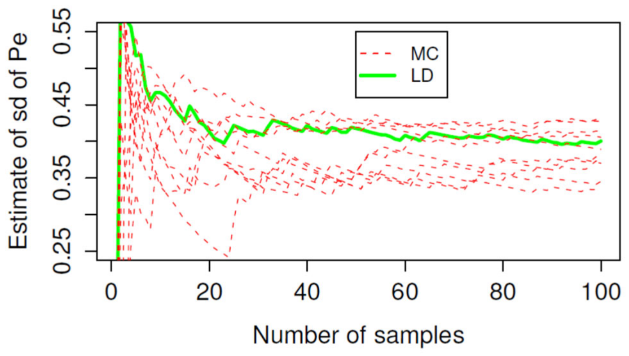
Our Quasi-Monte Carlo and Multi-Resolution variation models are suitable for examining the impact of parameter variation on many aspects of a microarchitecture. In this section, we evaluate our variation model and sampling methodology by applying it to two aspects of high-performance processor design which are extremely sensitive to parameter variation: (1) timing errors associated with timing speculative architectures [45] and (2) chip leakage power. Our first application examines trade-offs in observed timing errors versus clock frequency and compares convergence rates of timing error rates under low-discrepancy sequences versus standard Monte Carlo samples. In the second application, we examine the on-die leakage power variations with both SR-LD sampling and MR-LD sampling comparing to MC. Both applications are compared against VARIUS [44] Monte Carlo samples as a baseline case, which has been widely adaped for architectural parameter variation sampling [45], [47].

For the timing error estimation, we use the VARIUS timing model. It adopts the Alpha-Power Law [62] to relate threshold voltage V_{th} and effective gate length L_{eff} to gate delay:

Equation 8-5: $T_g \propto \frac{L_{eff}V}{\mu(V-V_{th})\alpha}$



(a) The estimate of sd



(b) The estimate of sd converges with increasing run size

Figure 8-5: The estimate of Pe's standard deviation and its convergence for Icache: Comparing 1 LD to 10 MC runs with (a) Fixed run size of 100 samples. (b) Fixed clock frequency at 1.0.

where V is the supply variation, the is carrier mobility and is an empirically derived constant. The gate delay is then used to estimate the timing error rate for logic and memory structures under process, voltage, and temperature variations. For leakage, we apply the HotLeakage [63] model which suggests that

$$\text{Equation 8-6: } I_{leak} \propto (\frac{kT}{q})^2 e^{q\frac{-V_{th}-V_{off}}{\eta kT}},$$

and the leakage power is proportional to I_{leak}. A factor of the total leakage power across the chip can be obtained by an integration of Equation 8-6, where k is the Boltzmann Constant, q is the electron charge, and V_{off} and □ are empirically determined parameters. We adopt these parameters from [44], [64] and [65] and scale them to 32nm technology.

We model a single core design featuring an Alpha 21264 processor scaled to a 32nm technology and use a floorplan detailing the microarchitectural structures of this design. In our experiments, we model random and systematic variation. A spherical correlation model [44] is used for all the variation samples. We assume V_{th} and L_{eff} are highly correlated [44] and use identical systematic variation samples for the two parameters. Our models apply □/□ of parameter variation, nominal supply and threshold voltage, and the decomposition of systematic and random components which follow that of [44]. These parameters are suitable for modeling high-performance designs in a 32nm technology.

A. Low-Discrepancy Variation Samples

To evaluate the effect of low-discrepancy sampling, we apply block-based LD variation samples to the VARIUS [44] timing error model, and estimate the distribution of the resulting timing error rates P_e for all the pipeline stages of a processor floorplan under a sequence of clock frequencies.

For comparison, the process is repeated with several sets of VARIUS Monte Carlo samples. The results of a large Monte Carlo set with 10,000 samples are used as a gold standard. This is a sample set size sufficiently large that sample mean and variance are very close to true distribution mean and variance. Note, that these sample sizes are prohibitively large for most simulation studies – they represent a best-case result.

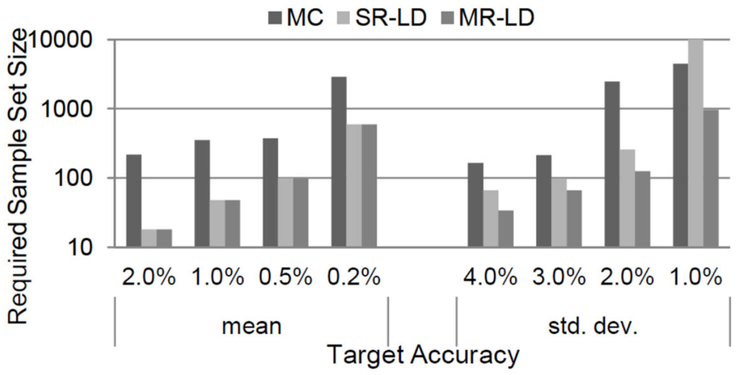
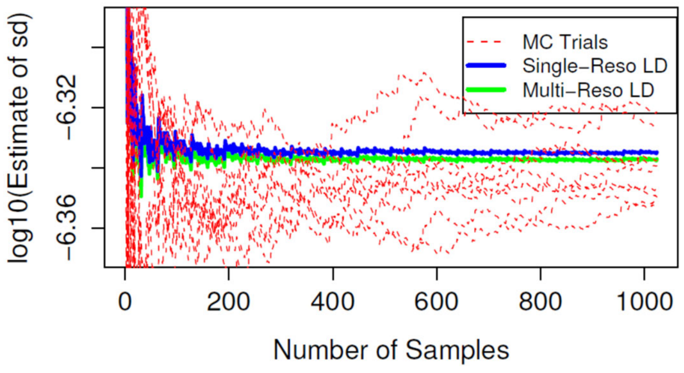
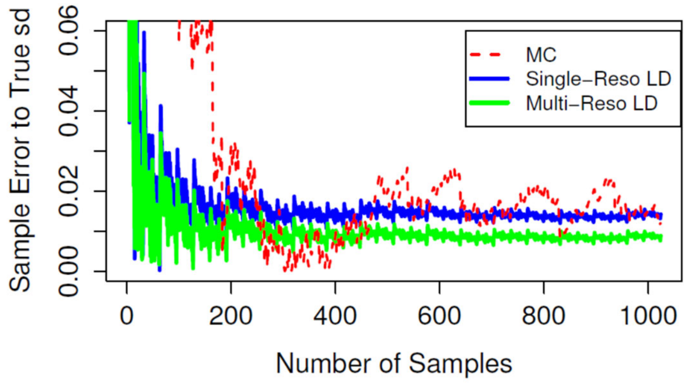


Figure 8-6: The number of samples needed for targeting accuracy when estimating chip leakage power: MC, SR-LD and MR-LD.

Although [47] suggests that 100 Monte Carlo samples show enough convergence when applying to VARIUS timing error model, our experiments show that on average any group of 100 MC samples still have considerable error when compared to the gold standard. On the contrary, Low-discrepancy samples produce high fidelity results. Figure 8-4 presents the error of 100 MC, 20 LD, 50 LD and 100 LD samples relative to 10,000 MC samples when estimating the mean of P_e of each processor pipeline component. For 10 out of 15 components, 20 LD samples have better accuracy than 100 MC samples, and 50 LD samples outperforms 100 MC on all components. One can view this in an alternative way. With the same number of samples, 100 LD gives an accuracy at least 75% better than 100 MC. This experiment proves that LD samples converge much faster than MC, which translates to either significant reduction of samples needed or better accuracy with the same number of samples. Since generating LD samples is a deterministic process, the shown results are repeatable and consistent. MC trials in contrast produce dramatic fluctuations for different runs and hence do not guarantee fast convergence.



(a) The estimate of *sd*



(b) The sample error to the true *sd*

Figure 8-7: The estimate and the sample error of the standard deviation of the leakage factor distribution with increasing sample set size: comparing MR-LD, SR-LD and standard MC.

The LD estimate of the standard deviation (*sd*) also shows faster convergence. Due to space limit, we only show in Figure 8-5 the estimate and convergence of standard deviation for the Icache block. The two sets of curves intuitively show the difference between the natures of LD and MC sampling and are consistent with our expectations. In summary, low-discrepancy techniques allow much faster convergence, resulting in large reductions in sample set size.

B. Low Discrepancy, Multi-Resolution Variation Samples

Now we evaluate the sampling of low discrepancy and multi- resolution grids, and we do this by estimating the deviations in chip leakage power. Multi-resolution analysis allows us to configure grid granularity within a component block according to its importance. For this study, we focus on leakage power and make grid densities proportional to power densities for all blocks, as explained in Section III. Power density of a block is determined by its temperature when with nominal V_{th0}, and we use the temperature distribution from [66] for the processor floorplan. After distributing the grid resolution, we generate the MR-LD variation samples with KLE-based LD methods.

We generate a set of MC, SR-LD and MR-LD samples for comparison. For the three different modeling methodologies, all samples are with the same sized parameter map (25 × 25), and the resulting leakage estimates are compared to that of a gold standard, 10,000 MC samples of resolution 50 × 50.

Figure 8-6 shows the number of samples needed to achieve the targeting accuracies. For the mean, the LD samples show at least 4× faster convergence than MC. However, SR-LD and MR-LD do not have significant difference themselves, which could be possible because the estimate errors of the mean are already low. We note that, although not shown in the figure, the average error of MR-LD is 0.4% smaller than SR-LD. For the standard deviation, LDs still converge faster than MC, and at the same time, MR-LD outperforms SR-LD, with speedup of at least 3.3×

and 2.2× respectively. Figure 8-7 presents more intuition for the estimate of *sd* as the number of samples grows (for clarity only until 1000 samples are shown), which leads to the observation that MR-LD converges to a better accuracy than SR-LD. Considering the fact that the difference between the computational efforts of implementing single-resolution and multi-resolution models is only distributing the grids with different density, the potential of the multi-resolution model is attractive, especially when accuracy is critical.

8.5. Conclusions and Future Work

In this work, we introduced a collection of techniques to help computer architects rethink the parameter variation model and improve sampling methodology when applying Monte Carlo simulations. Our key contributions were: (1) to develop spatial variation representations that could be applied to study architectural components while leveraging properties of the low-discrepancy and (2) to introduce multi-resolution models that adapt grid resolution to suit the relative importance of a component. We evaluated our techniques using a series of Monte Carlo experiments and found that in most cases our improved modeling and sampling methodology can dramatically reduce the number of samples needed to achieve convergence.

As one of the most straightforward ways to decompose parameter variation, Karhunen-Loeve Expansion (KLE) is adapted for the spatially correlated parameter variation model. However, KLE is still Fourier-like, meaning that each orthogonal term in the decomposition captures the information across the whole spatial domain. Considering the target of the entire processor where the pipeline stages' characteristics differ from each other, there might be other ways to decompose the parameter variation while taking the differences between different stages into consideration. One possible way is *wavelet decomposition*, in which each term localizes one specific part of the domain, and hopefully this could lead to a better approach to represent the different variation scenarios in different pipeline stages.

We evaluated our Multi-Resolution approach by distributing the grid densities proportional to the target function densities. While sharing a similar motivation as the multi-level grid files from database research [67] used for selectivity estimation, in this work we have a slightly different context. Namely, the coarseness of the resolution is varied based on the sensitivity to variations. For future work, we would like to further investigate the problem of dynamic fine-tuning of the grid-map and sample generation, in reaction to some (observed) changes in the parameters variation and component activity factors that may affect the validity of the experiments. Towards that, we will try to apply some of the techniques for streaming data management [68] in our context.

Chapter 9 Conclusions

Fault-tolerant architectures utilizing program localities have shown promising results in detecting silent data corruptions (SDC), while keeping a lower cost in area, performance, and energy consumption compared with other approaches (e.g., TMR [69]). Multiple techniques have been proposed to capture program locality in load/store references and values, and check for locality violation for fault detection. However, these approaches either achieves low fault coverage or incurs complex hardware design and energy overhead.

In this work, we have explored new dimensions in program locality and proposed techniques for using these types of locality for fault detection.

First, we have studied the longer-term program locality in program phases. We showed that program phase information can be reused when a phase repeats itself. We also found that within a 100 million instruction SimPoint [18], there are only 13 unique phases on average, which provides a fertile ground for learning the program phases during these phases and reuse the information for repetitive phases.

Second, we have studied shorter-term locality in memory strides. We showed that a partial-global memory stride is more efficient than a reference in representing program locality. We also designed an algorithm for static instruction partitioning to create effective partial-global memory strides.

Additionally, we have proposed to use Bloom filters to store and lookup program locality patterns. We presented a design of heterogeneous Bloom filter system for the instruction partitions, as well as for strides and memory values, for efficient storage and fast lookup.

Lastly, we designed, implemented, and evaluated a learn-and-detect fault detection framework. We described how the framework weaves the program phases and memory information together to detect transient faults. At the beginning of each unique phase, the instruction partitions are learned during a dedicated learning period, followed by a learning period to warm up the stride and value Bloom filters. The instruction partitions and memory information Bloom filter content is stored into L3 cache and if a unique phase is repeated upon, the content can be reloaded into the designated hardware for the repeating phase to use. This framework allowed detector customization based on program phases with low learning overhead. We have shown that with the proposed framework, a total of 80.8% of SDCs are covered with less than 1% false positive rate.

Limitations

The presented fault detection mechanism has its limitations in the following two aspects.

First, our fault detection mechanism does not distinguish application faults from hardware faults. Like a hardware fault, an application bug can trigger a detector signal, but unlike hardware faults which can be corrected by the replay or rollback, the signal raised by an application bug will eventually be suppressed as a false positive. Therefore, running programs that are more likely to have software bugs on such a platform can trigger more false positives *and result in higher performance and energy overheads than expected*. Also, since the stride or value pattern from a false positive will be inserted into the Bloom filters, the pollution can cause loss in fault coverage. As a result of this limitation, we would recommend that only well-tested applications be run on these platforms. However, in a context where fault tolerance is important, it is unlikely for a user to run beta software on such a platform. Therefore, this restriction should not have any significant impact in practice.

Second, during the learning periods, since the fault detector is not able to detect faults, the system needs extra fault-tolerant measures temporarily to protect the program execution. This can be achieved by running a protector thread through either dual-modular redundancy (DMR) or simultaneous multi-threading (SMT) [73, 74]. Our learning periods take 3% of all dynamic instructions and protecting this portion of the program execution will incur performance and energy overheads. A DMR approach executes the protector thread on a neighboring core, which has little performance impact for the same program, but using the other core as a mirroring hardware module during 3% of all instructions can incur ~3% overall energy overhead. An SMT approach executes the protector thread on the same core during the learning periods. By utilizing vacant functional units within the same core, SMT-based fault tolerant mechanisms incur a moderate performance overhead (e.g., 20% [73, 74]) and slightly significant energy cost (e.g., 50% [17]). For our 3% learning periods, this translates into 0.6% total performance impact and 1.5% overall energy overhead. In our evaluations in Chapter 6 and 7, we protect the learning periods with temporal redundancy by re-executing all instructions (without SMT), thus the performance and energy costs during learning periods are both ~3%. This can be improved with a DMR or SMT approach.

Future Work

While this work tries to comprehensively cover silent data corruptions and keep the cost low, there are still aspects that can be the object of future work. First, even though the benchmarks used are diversified in behavior, they may not be representative of the workloads that are most susceptible to silent data corruptions. A set of richer benchmarks can be used for future evaluation. Second, the system emulation mode is insightful for a timing-accurate simulation, but it skips the detailed simulation of kernel instructions. Future evaluations can use full system mode to get more comprehensive results on the kernel code execution. Third, the front-end fault filter itself still has room for improvement in fault coverage and false positives. Future work can explore other ways of detecting front-end faults.

References

[1] Von Neumann, John. "Probabilistic logics and the synthesis of reliable organisms from unreliable components." *Automata studies* 34 (1956): 43-98.

[2] Meixner, Albert, Michael E. Bauer, and Daniel Sorin. "Argus: Low-cost, comprehensive error detection in simple cores." *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007.

[3] Li, Man-Lap, et al. "Understanding the propagation of hard errors to software and implications for resilient system design." *ACM Sigplan Notices* 43.3 (2008): 265-276.

[4] Hari, Siva Kumar Sastry. *Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems running Multithreaded Workloads*. Diss. University of Illinois at Urbana-Champaign, 2009.

[5] Reddy, Vimal, and Eric Rotenberg. "Inherent time redundancy (itr): Using program repetition for low-overhead fault tolerance." *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. IEEE, 2007.

[6] Smolens, Jared C., et al. "Fingerprinting: bounding soft-error detection latency and bandwidth." *ACM SIGOPS Operating Systems Review* 38.5 (2004): 224-234.

[7] Dhodapkar, Ashutosh S., and James E. Smith. "Managing multi-configuration hardware via dynamic working set analysis." *Proceedings 29th Annual International Symposium on Computer Architecture*. IEEE, 2002.

[8] Venkatasubramanian, Rajesh, John P. Hayes, and Brian T. Murray. "Low-cost on-line fault detection using control flow assertions." *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.*. IEEE, 2003.

[9] Sahoo, Swarup Kumar, et al. "Using likely program invariants to detect hardware errors." *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008.

[10] Nistor, Adrian, et al. "Toddler: Detecting performance problems via similar memory-access patterns." *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[11] Wang, Nicholas J., and Sanjay J. Patel. "ReStore: Symptom-based soft error detection in microprocessors." *IEEE Transactions on Dependable and Secure Computing* 3.3 (2006): 188-201.

[12] Pattabiraman, Karthik, et al. "Automated derivation of application-specific error detectors using dynamic analysis." *IEEE Transactions on Dependable and Secure Computing* 8.5 (2010): 640-655.

[13] Li, Man-Lap, et al. "Trace-based microarchitecture-level diagnosis of permanent hardware faults." *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008.

[14] Nomura, Shuou, et al. "Sampling+ dmr: practical and low-overhead permanent fault detection." *ACM SIGARCH Computer Architecture News* 39.3 (2011): 201-212..

[15] Narayanasamy, Satish, Ayse K. Coskun, and Brad Calder. "Transient fault

prediction based on anomalies in processor events." *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007.

[16] Racunas, Paul, et al. "Perturbation-based fault screening." *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007.

[17] Pomeranz, Irith, and T. N. Vijaykumar. "FaultHound: value-locality-based soft-fault tolerance." *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015.

[18] Hamerly, Greg, et al. "Simpoint 3.0: Faster and more flexible program phase analysis." *Journal of Instruction Level Parallelism* 7.4 (2005): 1-28.

[19] Sherwood, Timothy, and Brad Calder. "Time varying behavior of programs." *In UC San Diego* (1999).

[20] Moshovos, Andreas. "Power-aware register renaming." *Computer Engineering group technical report* (2002): 01-08.

[21] Ayala, José L., Marisa López-Vallejo, and Alex Veidenbaum. "Energy-efficient register renaming in high-performance processors." *Proceedings of WASP*. 2003.

[22] Anjam, Fakhar, Stephan Wong, and Faisal Nadeem. "A multiported register file with register renaming for configurable softcore VLIW processors." *2010 International Conference on Field-Programmable Technology*. IEEE, 2010.

[23] Ramakrishna, M. V., E. Fu, and E. Bahcekapili. "Efficient hardware hashing functions for high performance computers." *IEEE Transactions on Computers* 46.12 (1997): 1378-1381.

[24] Sanchez, Daniel, et al. "Implementing signatures for transactional memory." *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 2007.

[25] Sanchez, Daniel. *Design and implementation of signatures for transactional memory systems*. University of Wisconsin-Madison Department of Computer Sciences, 2007.

[26] Jouppi, Norman P. "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers." *ACM SIGARCH Computer Architecture News* 18.2SI (1990): 364-373.

[27] Fu, John WC, Janak H. Patel, and Bob L. Janssens. "Stride directed prefetching in scalar processors." *ACM SIGMICRO Newsletter* 23.1-2 (1992): 102-110.

[28] Sherwood, Timothy, Suleyman Sair, and Brad Calder. "Predictor-directed stream buffers." *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 2000.

[29] Henning, John L. "SPEC CPU2006 benchmark descriptions." *ACM SIGARCH Computer Architecture News* 34.4 (2006): 1-17.

[30] Mohan, Nitin, et al. "Design techniques and test methodology for low-power TCAMs." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 14.6 (2006): 573-586.

[31] Taylor, David E. "Survey and taxonomy of packet classification techniques." *ACM Computing Surveys (CSUR)* 37.3 (2005): 238-275.

[32] Saini, Subhash, Johnny Chang, and Haoqiang Jin. "Performance evaluation of the

intel sandy bridge based nasa pleiades using scientific and engineering applications." *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, Cham, 2013.

[33] Intel® 64 and IA-32 Architectures Optimization Reference Manual, https://www.intel.com, 2016

[34] Binkert, Nathan, et al. "The gem5 simulator." *ACM SIGARCH computer architecture news* 39.2 (2011): 1-7.

[35] Li, Sheng, et al. "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures." *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2009.

[36] Balasubramonian, Rajeev, et al. "CACTI 7: New tools for interconnect exploration in innovative off-chip memories." *ACM Transactions on Architecture and Code Optimization (TACO)* 14.2 (2017): 1-25.

[37] Mukherjee, Shubhendu S., Joel Emer, and Steven K. Reinhardt. "The soft error problem: An architectural perspective." *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005.

[38] Schroeder, Bianca, Eduardo Pinheiro, and Wolf-Dietrich Weber. "DRAM errors in the wild: a large-scale field study." *ACM SIGMETRICS Performance Evaluation Review* 37.1 (2009): 193-204.

[39] Lakshminarayanan, Karthik, Anand Rangarajan, and Srinivasan Venkatachary. "Algorithms for advanced packet classification with ternary CAMs." *ACM SIGCOMM Computer Communication Review* 35.4 (2005): 193-204.

[40] Shah, Devavrat, and Pankaj Gupta. "Fast incremental updates on Ternary-CAMs for routing lookups and packet classification." *Proceedings of Hot Interconnects*. 2000.

[41] Shinde, Rajendra, et al. "Similarity search and locality sensitive hashing using ternary content addressable memories." *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010.

[42] Sorin, Daniel J. "Fault tolerant computer architecture." *Synthesis Lectures on Computer Architecture* 4.1 (2009): 1-104.

[43] Gizopoulos, Dimitris, et al. "Architectures for online error detection and recovery in multicore processors." *2011 Design, Automation & Test in Europe*. IEEE, 2011.

[44] R. Teodorescu, B. Greskamp, J. Nakano, S. Sarangi, A. Tiwari, and J. Torrellas, “Varius: A model of parameter variation and resulting timing errors for microarchitects,” *IEEE Trans on Semiconductor Manufacturing*, 2008.

[45] S. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas, “EVAL: Utilizing processors with variation-induced timing errors,” in *IEEE/ACM Int. Symp. on Microarchitecture*. IEEE Computer Society, 2008.

[46] B. Greskamp, L. Wan, U. R. Karpuzcu, J. J. Cook, J. Torrellas, D. Chen, and C. Zilles, “Blueshift: Designing processors for timing speculation from the ground up,” in *Int. Symp. on High-Performance Computer Architecture*, 2009.

[47] R. Teodorescu, J. Nakano, A. Tiwari, and J. Torrellas, “Mitigating pa- rameter variation with dynamic fine-grain body biasing,” in *Int. Symp. on Microarchitecture*, 2007.

[48] D. Citron, “MisSPECulation: partial and misleading use of SPEC CPU2000 in computer architecture conferences,” in *Int. Symp. on Com- puter Architecture*. ACM, 2003.

[49] J. J. Yi, R. Sendag, D. J. Lilja, and D. M. Hawkins, “Speed and accuracy trade-offs in microarchitectural simulations,” *IEEE Trans. on Computers*, 2007.

[50] A. Singhee, S. Singhal, and R. Rutenbar, “Practical, fast monte carlo statistical static timing analysis: Why and how,” in *Int. Conf. on Computer- Aided Design*, 2008.

[51] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, 1992.

[52] C. S. Amin, N. Menezes, K. Killpack, F. Dartu, U. Choudhury, and N. H. andYehea I. Ismail, “Statistical static timing analysis: How simple can we get?” in *Design Automation Conference*, 2005.

[53] T. Karnik, S. Borkar, and V. De, “Probabilistic and variation-tolerant design: Key to continued moore’s law,” 2004.

[54] E. Humenay, D. Tarjan, and K. Skadron, “Impact of parameter variations on multi-core chips,” in *Workshop on Architectural Support for Gigascale Integration*, 2006.

[55] P. Friedberg, Y. Cao, J. Cain, R. Wang, J. Rabaey, and C. Spanos, “Modeling within-die spatial correlation effects for process-design co- optimization,” in *Int. Symp. on Quality Electronic Design*, 2005.

[56] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner1, and T. Mudge, “Razor: A low- power pipeline based on

circuit-level timing speculation," in *Int. Symp. on Microarchitecture*, 2003.

[57] A. Singhee and R. Rutenbar, "From finance to flip flops: A study of fast quasi-monte carlo methods from computational finance applied to statistical circuit analysis," in *Int. Symp. on Quality Electronic Design*, 2007.

[58] J. Kiefer, "On large deviations of the empirical d. f. of vector chance vari- ables and a law of the iterated logarithm," *Pacific Journal of Mathematics*, 1961.

[59] P. Bratley, B. Fox, and H. Niederreiter, "Implementation and tests of low-discrepancy sequences," *ACM Trans. on Modeling and Compurter Simulation*, 1992.

[60] M. Loeve, "Probability theory," 1977.

[61] T. Davies and R. Martin, "Low-discrepancy sequences for volume prop- erties in solid modelling," in *CSG '98 Conference*, 1998.

[62] T. Sakurai and A. Newton, "Alpha-power law MOSFET model and its applications to CMOS inverterdelay and other formulas," *IEEE Journal of Solid-State Circuits*, 1990.

[63] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," University of Virginia, 2003.

[64] L. Zhang, L. S. Bai, R. P. Dick, L. Shang, and R. Joseph, "Process variation characterization of chip-level multiprocessors," in *Design Automation Conference*, 2009.

[65] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45nm design exploration," in *Int. Symp. on Quality Electronic Design*, 2006.

[66] Y. Han, I. Koren, and C. A. Moritz, “Temperature aware floorplanning,” in *Workshop on Temperature-Aware Computer Systems*, 2005.

[67] K.-Y. Whang, S. Kim, and G. Wietherhold, “Dynamic maintenance of data distribution for selectivity estimation,” *VLDB Journal*, 1994.

[68] Sharfman, A. Schuster, and D. Keren, “A geometric approach to monitoring threshold functions over distributed data streams,” in *SIGMOD Conference*, 2006.

[69] Fiala, David, et al. "Detection and correction of silent data corruption for large-scale high-performance computing." *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012.

[70] Ganesan, Karthik, Deepak Panwar, and Lizy K. John. "Generation, validation and analysis of SPEC CPU2006 simulation points based on branch, memory and TLB characteristics." *SPEC Benchmark Workshop*. Springer, Berlin, Heidelberg, 2009.

[71] Chung, Jae Yoon, et al. "An effective similarity metric for application traffic classification." *2010 IEEE Network Operations and Management Symposium-NOMS 2010*. IEEE, 2010.

[72] Strehl, Alexander, Joydeep Ghosh, and Raymond Mooney. "Impact of similarity measures on web-page clustering." *Workshop on artificial intelligence for web search (AAAI 2000)*. Vol. 58. 2000.

[73] Vijaykumar, T. N., Irith Pomeranz, and Karl Cheng. "Transient-Fault Recovery Using Simultaneous Multithreading." *Computer Architecture, International Symposium on*. 2002.

[74] Rotenberg, Eric. "AR-SMT: A microarchitectural approach to fault tolerance in

microprocessors." *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*. IEEE, 1999.