NATIONAL INSTITUTE OF TECHNOLOGY MEGHALAYA

DOCTORAL THESIS

Architecture-aware Program Analysis Techniques for Approximate Computing

Author:
Bernard Nongpoh

Supervisor:
Dr. Rajarshi Ray

Co-Supervisor:
Dr. Ansuman Banerjee

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Department of Computer Science & Engineering
National Institute of Technology Meghalaya

February 2020

Architecture-aware Program Analysis Techniques for Approximate Computing

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

in

Computer Science & Engineering

Submitted by
Bernard Nongpoh
Registration No. P15CS001

Under the supervision of
Dr. Rajarshi Ray (Supervisor)
Assistant Professor, NIT Meghalaya
Dr. Ansuman Banerjee (Co-Supervisor)
Associate Professor, ACMU, ISI-Kolkata

Department of Computer Science & Engineering
National Institute of Technology Meghalaya
Shillong-793003, Meghalaya, India

February 2020

National Institute of Technology Meghalaya

Abstract

Department of Computer Science & Engineering

Doctor of Philosophy

Architecture-aware Program Analysis Techniques for Approximate Computing

by Bernard Nongpoh

Rising concerns on energy efficiency and lack of a clear roadmap for performance scalability of modern semiconductor systems have motivated researchers in recent times to look for alternate computing paradigms that are fundamentally different from the classical ones. The quest for a scalable computing solution that can ensure performance, energy efficiency, low latency and acceptable reliability has been a topic of wide research interest both in academia and industrial research, and several proposals across the computation stack have been proposed for quite some time. In the last decade, the paradigm of approximate computing has shown significant promise and prominence in a number of application domains, and is thus being increasingly adopted across the computing stack, from algorithms to circuits. The main philosophy driving approximate computing is to derive energy and performance efficiency by trading accuracy within acceptable limits. An arsenal of approximation techniques have been proposed in literature, from algorithms, programming languages to circuits. While approximate computing has made inroads into a number of application domains like computer vision, machine learning, planning etc., a wider adoption is on the anvil. Researchers have identified an arsenal of challenges to be addressed to make approximate computing the pervasive computing paradigm. This thesis aims to address some of these challenges with an objective to make this evolving paradigm more widespread and mainstream.

One of the fundamental challenges in approximate computing is in identifying components or elements that are suitable candidates for approximation. While on one hand, manual annotations are unreliable and error-prone, on the other hand, automated tools that can automatically derive approximable points are yet to evolve as generic solutions, and are mostly limited to specific application domains. Classifying elements in a given application as precise or approximable is a key enabler for approximate computing to succeed. A primary focus of our research is in devising a principled generic automated workflow for identifying components of a given application that are amenable to approximation. We propose a framework to automatically classify data variables and structures in a program as either approximable or inapproximable, with probabilistic reliability guarantees. Extending the approximability analysis technique further to handle program instructions, load and branch instructions in particular, we connect the results of our analysis to the processor runtime, whereby we setup a coupling with speculative execution to derive more performance benefits by deviating from the classical philosophy of rollbacks and pipeline flushes when a data value deemed approximable is mis-speculated or an approximable branch is mis-predicted. Our results indicate that some approximable loads and branches can be executed without any execution roll-back in the pipeline and yet can assert a certain user-specified quality of service with a probabilistic guarantee. Finally, we revisit the coherence requirement in the multi-core computation stack, wherein the computational load is shared between cores, but the load / store operations made by one core on the shared data elements are usually made visible to the other cores at some point to maintain consistency and coherence of computation. Embracing the approximate computing philosophy, we propose to selectively stop inter-core propagation of updates made by write instructions on shared data deemed approximable by our analysis. The goal of this analysis is to look for write instructions on shared data, whose updates if not communicated to the other cores, do not cause the computation results to deviate beyond acceptable limits. Using the knowledge of approximate writes on shared data, we propose modifications on a cache-coherence protocol and show performance benefits in the execution of multithreaded programs on shared memory multicore architectures.

The key results outlined in this thesis are backed up with extensive experiments on public domain workloads. Further, we provide comparisons with existing methods in literature to demonstrate our novelty and performance benefits. We believe that our solutions can have important ramifications going forward in addressing some of the important challenges in approximate computing.

Keywords: Program Analysis, Computer Architecture, Approximate Computing.

National Institute of Technology Meghalaya

Declaration of Authorship

I, **Bernard Nongpoh**, Registration No. **P15CS001** declare that this thesis titled, **“Architecture-aware Program Analysis Techniques for Approximate Computing”** and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

February 11, 2020

Bernard Nongpoh

National Institute of Technology Meghalaya

Certificate of the Supervisor (s)

This is to certify that the thesis titled, **"Architecture-aware Program Analysis Techniques for Approximate Computing"** submitted by **Mr. Bernard Nongpoh**, Registration No. **P15CS001**, in partial fulfillment of the requirements for the award of the degree of Doctor of Philosophy in the Department of Computer Science & Engineering is a record of research work carried out by him under our supervision and guidance.

All help received by him from various sources have been duly acknowledged.

No part of this thesis has been submitted elsewhere for award of any other degree or diploma.

(Co-Supervisor)	(Supervisor)
Dr. Ansuman Banerjee	Dr. Rajarshi Ray
Associate Professor	Assistant Professor
Advanced Computing & Microelectronics Unit	Dept. of Computer Science & Engineering
Indian Statistical Institute Kolkata	National Institute of Technology Meghalaya

National Institute of Technology Meghalaya

Approval of the Thesis

This is to certify that the thesis titled **"Architecture-aware Program Analysis Techniques for Approximate Computing"** has been submitted by **Bernard Nongpoh**, Registration No. **P15CS001**, in fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science & Engineering of National Institute of Technology Meghalaya.

Supervisor	Co-Supervisor
DC members	External Examiner(s)
Chairman DC/DRC	

National Institute of Technology Meghalaya

February 11, 2020

Bernard Nongpoh

Acknowledgements

A I am grateful to God almighty for his gracious blessings and granting me good health, wellbeing and paving the way to pursue and complete this thesis. First, and foremost I would like to express my deepest gratitude to my Mother (Bei) Mrs. Belinda Nongpoh and to my Father (Pa) Mr. Martin Lyngdoh for their love, support, and prayer.

I would like to express my sincere gratitude to my supervisor Dr. Rajarshi Ray for his kindness, patience, motivation, constant effort and support. He has taught me to conduct quality research with perseverance. My sincere gratitude to my co-supervisor Dr. Ansuman Banerjee for his generosity, kindness, constant motivation, tremendous support, and effort throughout the course of my Ph.D. It has been an honor working under two kind-hearted persons, Dr. Rajarshi Ray, and Dr. Ansuman Banerjee. Thank you Sirs for believing in me.

I would like to thank Mrs. Moumita Das, a Ph.D. student at ISI-Kolkata for her contribution and collaboration who helped me kick-start the working with an architectural simulator. My gratitude to my collaborator Mr. Saikat Dutta, a Ph.D. student at the University of Illinois, Urbana-Champaign, USA for his contribution and especially spending valuable time in showing me how to work with byte-code instrumentation.

I thank Dr. Himadri Sekhar Paul (External DC member) for his insightful comments and suggestions. I would also like to thank all DC/DRC members for their feedback and suggestions.

I thank my fellow batchmates and friends: Phrangboklang, Amit, Mercy, Pynbianglut, Anirban, C Lalengmawia, Farhana, Swamy, Mahindra, Wasmir, Mir and Carl for their constructive discussions and the fun we have had in the last four years.

I take this opportunity to thank the National Institute of Technology Meghalaya for giving me this golden opportunity, infrastructure, and support in these four years of my study. I would like to express my gratitude to the Director, Registrar, Deans, HoD (CSE), faculties and non-teaching staff for their services and help I have received during my Ph.D.

I am grateful and thankful to the Ministry of Electronics and Information Technology, GoI for their generosity in providing financial assistance through the Visvesvaraya Ph.D. Scheme without which I would not have been able to pursue my research. I thank Google India and ACM SIGSOFT for their generosity in providing travel grant for presenting our paper at ESEC/FSE 2017 conference, Germany.

I am forever grateful to my wife Mrs. Balumlang Mary Kurkalang for believing in me and her continuous support. I thank all my brothers and my sister for their prayers.

Dedicated to my father, mother, wife and to all my ever encouraging family members. . .

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

In recent times, it is widely acknowledged that a major transformation is under way in computer hardware as processors strive to extend and assert their sustenance beyond the speculated end of Moore's Law. This has fostered the growth of new forms of heterogeneous processors, heterogeneous memories, near-memory computation structures, and, in some cases, computing elements that break free from the Von Neumann philosophy. Additionally, with energy rapidly becoming a first class citizen of utmost criticality and concern, alternative computation models that are more energy efficient are being envisioned, and a significant effort is being harnessed to develop computation paradigms and frameworks that can cater to the required compute needs, while being cognizant to the energy dissipation metric.

Indeed, computing devices such as mobiles and Internet of Things (IoT) devices have become ubiquitous today, these devices are required to run under low power constraints. With the ubiquity of the Internet and explosion in digital data, and proliferation of modern applications such as computer vision, machine learning, data mining, recognition and search, modern computing systems are confronted with a sustenance challenge, with an imminent need to scale in computation power on one hand, and be more energy efficient on the other. This has spearheaded research on alternate computation models, and approximate computing is one such effort that has shown significant promise in recent times.

Approximate computing is driven by the observation that most modern applications in machine learning, computer vision, and multimedia processing do not always require exact or precise results to be useful; a good enough answer is sufficient and this leaves room for optimization. Studies [1] reveal that on an average, 83% of the runtime is spent in precise computations that are tolerant to errors in their outputs. Examples include myriads of application domains ranging from computer vision, machine learning to robot motion planning. The motivation driving the approximate compute paradigm is the fact that we can exploit the inherent resilience of an application for potential performance gain and reduce power consumption.

Many application domains today deal with computation requirements that have a tolerance towards errors and approximation. They exhibit a somewhat inherent resilience towards errors, the property of an application in which parts of its computation or data or both could entail minor inaccuracies without disturbing the output beyond an acceptable limit. Approximate computing is a computing paradigm where inaccuracy is allowed in computations deliberately in controlled measures with the motivation of gaining on energy and performance efficiency. In the approximate computing paradigm, an application output is usually a set of values (maybe a set of discrete ones or a continuous band), also known as the Quality of Service (QoS) band. An inaccuracy tolerant application is thus allowed to produce outputs within this QoS band in the presence of inaccuracies in its computation or data or both. The inaccuracy tolerance in the applications can arise due to factors like the inability of humans to perceive noise within limits, inherent nature of approximation in the output etc.

The focus of this thesis is to study and improve on various approximate computing techniques at the language and micro-architecture level. The main focus is on developing automated methods for analyzing portions of an application fit to be approximable and approximation techniques at the micro-architecture to improve the performance of an application. The following discussion outlines our motivation followed by the contributions and organization of the thesis.

1.1 Motivation and Objectives

The paradigm of approximate computing is poised at an interesting juncture today, replete with several open challenges to be addressed. We mention some of them below, that serve as the primary motivation behind this work.

One of the fundamental challenges in approximate computing today is in identifying components or elements that are suitable candidates for approximation. While on one hand, manual annotations are unreliable and error-prone, on the other hand, automated tools that can automatically derive approximable points are yet to evolve as generic solutions, and are mostly limited to specific application domains. Classifying elements in a given application as precise or approximable is a key enabler for approximate computing to succeed.

Designing the appropriate hardware and software abstractions and interfaces for approximate computing techniques that can cut across the entire compute stack constitutes another important challenge. While there has been several proposals for embracing approximate computing at the programming language level, thereby allowing programmers to mark data elements, instructions and components that are approximable, proposals for design of approximate hardware blocks are also abundant in literature. However, an end-to-end solution that can compound the benefits of the findings derived at the program level, techniques for specifying and ensuring quality with an efficient means to compose approximate hardware and software [2] that can cut through the operating system and architecture down to the devices is still not in place, to the best of our knowledge. While there has been isolated proposals for embracing approximate computing at each level, we believe connecting the different layers of the stack requires a different viewpoint altogether. This is a non-trivial challenge considering that the computation models and principles have to be properly amalgamated to derive the compound benefits. This serves as another motivation towards our contributions in the later chapters.

Concurrency has been an useful aid for performance enrichment for scalable computation. The study of approximate computing in the terrain of concurrent computation, more specifically, concurrent programs, multi-processing and multi-core architectures has not progressed significantly in literature, to the best of our knowledge. This, in our view, poses an interesting challenge, considering the fact that ensuring correctness of computation within concurrency is in itself a major challenge, and requires specific techniques to address factors like races, coherence and memory consistency. Putting approximate computing on top of concurrency to ensure a further tolerance in terms of correctness has to be properly defined and dealt with, and requires careful introspection. Additionally, tying the benefits of approximation in a concurrency setting inside concurrent compute architectures poses another level of complexity, and this serves as the motivation for the last contribution of our thesis.

In the following discussion, we mention briefly the contributions of this thesis.

1.2 Thesis contributions

The main contributions of this dissertation are outlined below.

- An automated framework for approximability analysis of program data: In this work, we propose a method to automatically identify error resilient program variables in an application with a probabilistic reliability guarantee. Our proposal implements a combination of dynamic and static analysis methods for sensitivity analysis. The dynamic analysis is based on statistical hypothesis testing, while the static analysis is based on classical data flow analysis. Experimental results compare our automated data classification with reported manual annotations on popular benchmarks used in approximate computing literature. Our framework achieves promising reliability results compared to manual annotations and earlier methods, as evident from the experimental results.

- Combining approximate computing with speculative execution: At the architecture level, a method for enhancing speculative execution with selective approximate computing is proposed. Speculative execution is an optimization technique used in modern processors by which predicted instructions are executed in advance with an objective of overlapping the latencies of slow operations. Branch prediction and load value speculation are examples of speculative execution used in modern pipelined processors to avoid execution stalls. However, speculative executions incur a performance penalty as an execution rollback when there is a misprediction. In this work, we propose to aid speculative execution with approximate computing by relaxing the execution rollback penalty associated with a misprediction. Extending on the above framework, we propose a sensitivity analysis method for data and branches in a program to identify the data load and branch instructions that can be executed without any rollback in the pipeline and yet can ensure a certain user-specified quality of service of the application with a probabilistic reliability. Our analysis is based on statistical methods, particularly hypothesis testing and Bayesian analysis.

- Approximate computing for multi-threaded programs on multi-cores: To improve the performance of multithreaded programs running on shared-memory multicore processors, we propose to embrace approximate computing by selectively relaxing the coherence requirement in order to reduce the cost associated with a cache-coherence protocol. In multicore and multicached architectures, cache coherence is ensured with a coherence protocol. However, the performance benefits of caching diminishes due to the cost associated with the protocol implementation. In this work, we propose a novel technique to improve the performance of multithreaded programs running on shared-memory multicore processors by embracing approximate computing. Our idea is to relax the coherence requirement selectively in order to reduce the cost associated with a cache-coherence protocol, and at the same time, ensure a bounded QoS degradation with probabilistic reliability. In particular, we detect write instructions in a multithreaded program that write to shared data, and propose an automated statistical analysis to identify those which can tolerate coherence faults. To leverage this observation, we propose an adapted cache-coherence protocol that relaxes the coherence requirement on stores for these approximable writes. Additionally, our protocol uses stale values for load misses due to coherence, the stale value being the version at the time of invalidation.

1.3 Thesis organization

This thesis is organized into 6 different chapters. Chapter 2 presents the background and related work. This chapter discusses existing works and methodologies in applying hardware/software approximate computing techniques. In Chapter 3, we present a technique to classify parts of the program that can be approximated using the dynamic analysis method. Chapter 4 presents a static analysis technique using classical data flow analysis. Chapter 5 presents our proposal for enhanced speculative execution for approximate computing. This chapter discusses a technique for sensitivity analysis of load and branch instructions followed by a Bayesian analysis technique to determine the cumulative effects of load/branch instructions. Chapter 6 presents approximate computing for multithreaded programs. Finally, Chapter 7 ends with a conclusion and discussion on future directions.

Chapter 2

Background and Related Work

In this chapter, we present a brief overview of the background topics needed for the forthcoming chapters. Further, we survey some of the relevant research literature related to the contributions of this thesis. The layout of this chapter is as follows: Section 2.1 presents an overview of the statistical methods used in the algorithms for approximate computing proposed in this thesis. Section 2.2 introduces program analysis, Section 2.3 discusses in brief speculative execution, and we conclude with Section 2.4 which discusses some related work. We begin with a discussion on the statistical methods used in this thesis.

2.1 Statistical Methods

One of the key contribution of this thesis is the connection that it builds between statistical methods and program analysis. Particularly, the concepts of acceptance sampling, hypothesis testing, probability distributions, joint probability distribution and Bayesian networks will be useful to discuss in order to appreciate some of the contributory algorithms presented in the thesis. We briefly present an exposition of these topics in the discussion below.

Acceptance Sampling: Acceptance sampling is a statistical technique to control quality of products in any production system. A collection of sample products are taken from the produced lot and the quality of these samples are tested. Based on the quality of the sampled lot only, the quality of the entire production system is either accepted for rejected [3]. Acceptance sampling is the choice of quality control mechanism in industry when testing a product destroys or degrades the product, or when testing each and every product is prohibitively expensive in terms of time and effort, or the number of products to test is too large. There are a number of *sampling plans* that could be employed in acceptance sampling. Though the sampling is made randomly, the sampling plan fixes the number of samples to test in order to ensure reasonable confidence of the decision. Acceptance sampling is relevant to our work because in our context, we consider a sample to be an execution of a program on a test input, and we intend to accept or reject the quality of a program based on finitely many test-cases, i.e., samples. We explain in detail the meaning of *quality of a program* in Section 3.3.2. Acceptance sampling technique is a good choice for us because for most of the programs of our interest, there are infinitely many test executions and therefore, exhaustive testing is not a choice. We now discuss the method of hypothesis testing.

Hypothesis Testing: It is a procedure to test the truth of a hypothesis over a population, by acceptance sampling. The procedure either accepts the claimed hypothesis or rejects it, based on the observations made on a limited number of samples from the population. As an example, we may consider the population to be the collection of products from a factory. We may then have a hypothesis saying that the quality of any product is *acceptable* with a probability at least 0.95. Now, the hypothesis requires testing by one of the testing procedures, to either accept or reject it. Generally, a hypothesis on a population is denoted by H and is called the null hypothesis, and H' denotes the contradictory hypothesis and is called the contrary hypothesis. For example, in the factory example discussed above, H will refer to the hypothesis that any factory product is *acceptable* with a probability at least 0.95. On the other-hand, the contrary hypothesis H' will refer to the claim that any factory product is *acceptable* with a probability less than 0.95. Clearly, both H and H' cannot hold together and one of them must be true. Our proposed algorithms in the thesis for approximate computing heavily relies on making hypotheses and testing them. We now briefly discuss one hypothesis testing procedure briefly, namely, the *Sequential Probability Ratio Test*, abbreviated as SPRT in the rest of the thesis. For the details on SPRT, the reader may refer to Section 3.3.3.

Sequential Probability Ratio Test (SPRT): SPRT is one out of the many hypothesis testing procedures that is proposed by Abraham Wald [4]. Unlike in other testing procedures where the number of samples are fixed a-priori, the principle behind SPRT is to decide whether additional samples need to be taken to accept or reject a hypothesis, on the basis of the previously observed outcomes. It has been shown to use the optimal number of samples to test a hypothesis under certain conditions. During execution of the procedure, it can take one of the following decisions:

- Reject the *null hypothesis H* in favor of the *contrary hypothesis H'* and then stop,
- Accept the *null hypothesis H* and stop,

- Fail to reach any conclusion and continue with the next sampling.

As the procedure samples, it keeps a ratio of two probabilities, p and q at the end of observing k samples. Out of the k samples, let k_1 denote the number of acceptable samples. Then, p is the probability of seeing k_1 acceptable samples and $k - k_1$ rejected samples, given that the probability of observing a random sample to be acceptable is known to be p_1. Similarly, q is the probability of seeing k_1 acceptable samples and $k - k_1$ rejected samples, given that the probability of observing a random sample to be acceptable is known to be p_0. The probabilities p_0 and p_1 define what is called the indifference region and is discussed in detail in Section 3.3.4.

$$\frac{p}{q} = \frac{{p_1}^{k_1}(1-p_1)^{k-k_1}}{p_0^{k_1}(1-p_0)^{k-k_1}} \tag{2.1}$$

The procedure compares this ratio with a predefined constant and terminates by accepting the hypothesis when this ratio is less than or equal to the constant. This constant is dependent on what is called the strength of the test. The strength of the test derives its value from the probability of making a type-I and type-II error by the test procedure. The reader may refer to Section 3.3.4 for further details of types of errors and strength of test.

2.2 Program Analysis

Program analysis is the process of automatically finding useful facts about programs [5]. It aims to improve software quality in term of reliability, security, and performance. It provides the tools and algorithms that can analyze other programs. Some of the applications of program analysis are [6]:

- *Security*: To check if a program leaks private data of a user [7].
- *Bug finding*: To expose as many assertion failures as possible [8].
- *Verification*: To check whether a program is behaving according to its specifications [9].
- *Compiler optimizations*: To improve the performance of programs by code transformations [10].
- *Automated parallelization*: To automatically convert sequential programs to parallel versions [11].
- *Integrated Development Environment (IDE) support*: To assist programmers during the time of development - assistance such as code suggestion, code completion, code refactoring and program understanding [12].

We now discuss two key properties of a program analysis procedure - *soundness* and *completeness*. In *sound analysis*, whenever the analysis infers a program to have a property (say $\mathcal{R}$), $\mathcal{R}$ indeed holds for the program. In *complete analysis*, for every program that has the property $\mathcal{R}$, the analysis infers $\mathcal{R}$ to hold on the program. An unsound analysis can infer a program to have property $\mathcal{R}$ when actually the program does not have the property. An incomplete analysis may fail to infer a program to have the property $\mathcal{R}$ when the program actually has the property.

Program analysis techniques are broadly classified into three kinds: *static*, *dynamic* and *hybrid* analyses [5]. In the following sections, we will briefly discuss static and dynamic analyses.

2.2.1 Static Analysis

Programmers often use software testing techniques to gain confidence on the correctness and functionality of a program with respect to its specifications. However, as famously quoted by Edsger W. Dijkstra, *"Program testing can be used to show the presence of bugs, but never to show their absence"* [13]. This is because the semantic behavior of a program is observed only for a limited number of test-inputs. How would a program behave if other test-inputs remain unknown. On the other hand, static analysis aims to automatically infer semantic properties of a given program at compilation time that holds true for all executions of the program [14], and are therefore strong claims. Many of the static analyses problems are undecidable (e.g. The *may alias* problem is undecidable [15]). To have decision procedures for static analysis problems, analysis procedures generally resort to approximations. Some of the static analysis tools that have proven useful in both industry and academia are:-

- Syntactic and logic errors: Lint [16], FindBugs [17] and Coverity [18]
- Detection of memory leak i.e., failure to deallocate memory when no longer required: Facebook Infer [19]
- Verifying that software meets the critical behavioral properties of the interfaces it uses, for example checking API usage rules: Microsoft SLAM [20]
- Verifying invariants: An invariant is a property that is always true in all possible executions for example ESC/Java [21]

Static analysis usually operates on a suitable intermediate representation of the program. A common and useful one is the control flow graph [22]. It is a graph that summarizes the flow of control in all possible runs of the program. Each unique statement in a program is represented by a node in the graph. Each outgoing edge from a node denotes a possible successor of that node in some possible execution. The control flow graph is a possible abstraction of a program. Figure 2.1 shows a sample *C* program and Figure 2.2 shows the corresponding control flow graph. The program consists of three integer variables x, y, z with the assertion *assert(y==14)*. This assertion is to check for all possible program executions, whether the assertion is valid or not. Figure 2.2 shows the control flow graph with each unique statement represented by a node in the graph. For example, the statement *int z=10* in the program is represented by a node with label 2 i.e., node *z=10* in the graph. Generally, a node is labeled by a positive integer numbered from 0 to N, where N is the number of nodes in the graph. For simplicity, we label the nodes of the graph with program statements. Static analysis techniques generally work with abstract states, each of which summarizes a set of concrete states. For instance, if an analysis requires to track the value of variables x, y and z of a program in Figure 2.1, the analysis keeps track of the set of values that the variables x, y and z may attain at each program point. These values are called the abstract states, in contrast to the concrete values in an actual run. The analysis may fail to accurately represent some values of the variables in the abstract state. For this kind of approximation, a static analysis can be *incomplete*. However, in a sound analysis, the values of the variable represented in the abstract state implies that they are indeed attainable for all runs of the program. Let us look into a simple static analysis discussed in Example 2.1.

```
1   void main()
2   {
3      int z=10;
4      int y=0;
5      while(true)
6      {
7         if(x==1) // here x is user input
8            y=z+4;
9         else
10           y=14;
11        assert(y==14);
12     }
13  }
14
```

Figure 2.1: An assertion to check that the value of y is constant for all runs.

Example 2.1 Consider an analysis that requires to check if a given variable always has a constant value. In the example code in Figure 2.1, the analysis needs to guarantee that for all possible executions, the assertion (y==14) in line 11 is indeed true. We now examine how the analysis discovers invariants [1] of the form (y==14), even for the programs that have an unbounded number of paths. We will use a common static analysis technique called iterative

[1]An invariant is a property that is always true in all possible executions.

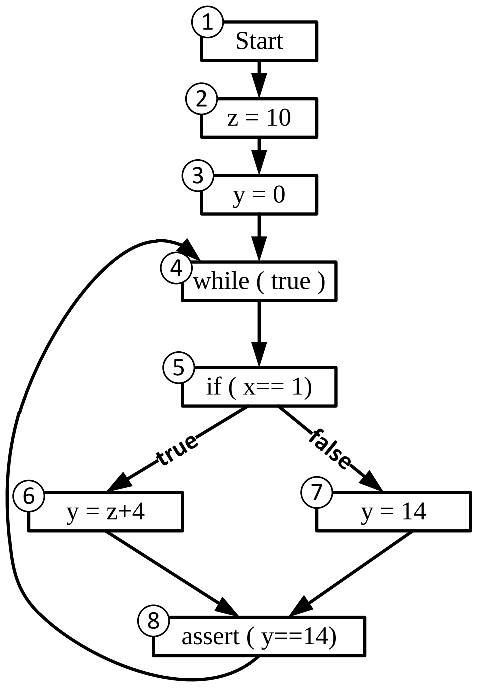


Figure 2.2: Control flow graph of the C program in Figure 2.1 with each node assigned with a distinct label.

approximation where at each program point, the analysis updates its knowledge of the value of x, y, and z. This update is based upon the information that has been inferred at the immediate predecessors of that program point. Figure 2.3 shows how the analysis works and a step-by-step description is as follows:-

- At the start node, the analysis has no information about the value of x, y, and z. A symbol ? denotes unknown values.
- Following the start node is the assignment state $z = 10$, which updates the value of variable z to 10. Similarly, after the statement $y = 0$, the value of y is now updated to 0.
- An interesting update occurs in the true branch of the statement $if(x == 1)$, that checks whether the value of x is 1. After taking the true branch, the analysis identifies that the value of x is 1. However, the value of x in the false branch is still unknown.
- After evaluating both the true and false branches, the analysis concludes that the value of y is 14. At this point, the analysis has concluded that, at each immediate predecessor of the assertion, the value of y is indeed 14 and hence the assertion is valid.

The analysis might need to visit the same program point multiple times (due to the presence of loops), hence the term iterative approximation. □

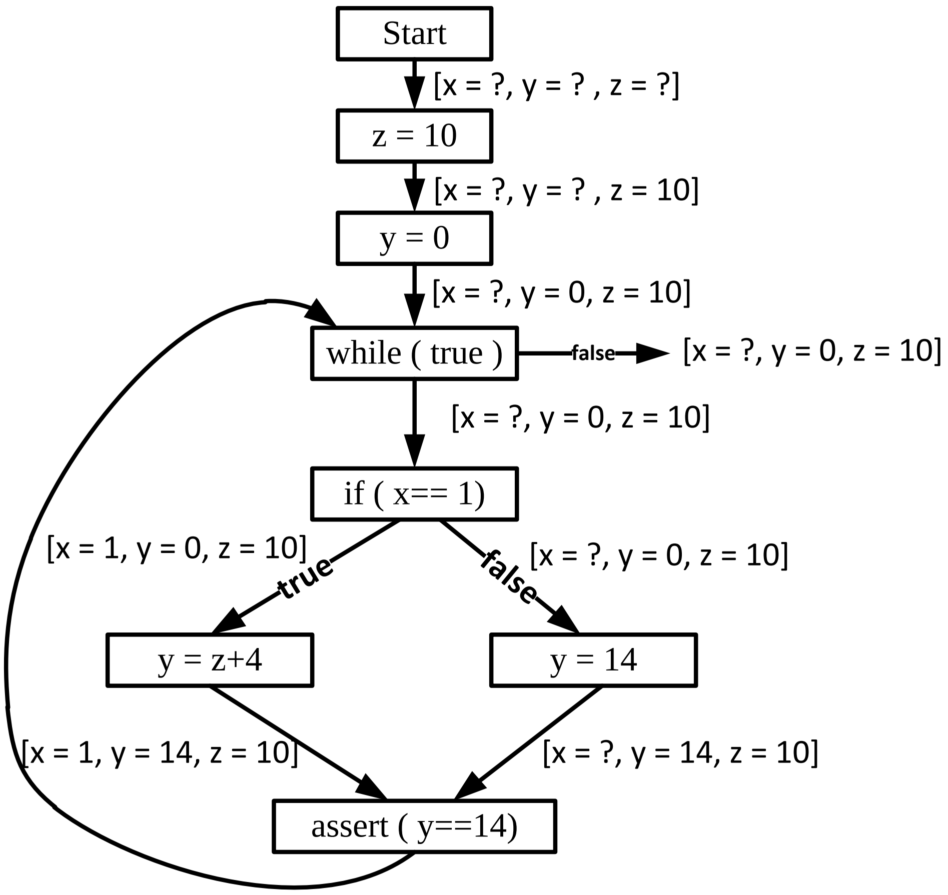


Figure 2.3: Static analysis using iterative approximation.

2.2.2 Data Flow Analysis

We present the necessary background related to our proposed static-dynamic combined sensitivity analysis method in Chapter 4. We first introduce the classical data flow analysis with an example, followed by lattice theory and then monotone framework for data flow analysis.

Data flow analysis is a type of static analysis for reasoning about the flow of data in a program [23]. As is the case for most static analysis approaches, data flow analysis usually operates on a suitable intermediate representation of the program. It typically operates on a control flow graph. A control flow graph is a graph that summarizes the flow of control in all possible runs of a given program. Here, we refer to the data flow analysis techniques which are sound but incomplete.

There are many methods based on data flow analysis such as, *Reaching Definitions Analysis* [24], *Available Expressions Analysis* [25], *Very Busy Expressions Analysis* [25], *Live Variable Analysis* [26] etc. To demonstrate how a data flow analysis works, we discuss the Reaching Definitions Analysis in detail.

Reaching Definitions Analysis

Reaching definitions analysis is based on data flow analysis [23] and it aims to determine for each program point, which assignments can reach the point and are not overwritten when execution reaches that point along some path. An assignment statement is a definition of a variable.

Example 2.2 Consider an assignment $[x = 1;]$, which assigns 1 to a variable x. This assignment statement $[x = 1;]$ is a definition of variable x. To identify each program point uniquely, each node in the control flow graph is assigned with a distinct label. Figure 2.2 shows each node with a label l, where l is a positive integer number ranging from 1 to N. Here, N is the number of nodes in the graph. For instance, the node with statement $z = 10$ is assigned a label 2. □

The reaching definition is then denoted as a pair $\langle v, l \rangle$ comprising the name of the defined variable, say v along with a label, say l of the node that defines it. More formally, for each node in the control flow graph, we assign a distinct label l. For each node in the graph with label l, the analysis computes two sets, the $IN[l]$ and the $OUT[l]$ set.

Definition 2.1 *$IN[l]$ is the set of facts at the entry of node l. It is the union of the set of facts at the exit of the nodes' immediate predecessors in the control flow graph, formally,*

$$IN[l] = \bigcup_{l' \in predecessors(l)} OUT[l']$$

■

Definition 2.2 *$OUT[l]$ is the set of facts at the exit of a particular node. The set of facts at the exit node l is equal to the set of facts at the entry of node l, minus any definitions that are overwritten by node l (we called this set as $KILL$ set) and union with any new definitions that are generated by node l (we called this set as GEN set). Formally,*

$$OUT[l] = (IN[l] - KILL[l]) \bigcup GEN[l]$$

■

Figure 2.4a shows an illustration of the union of OUT set of N predecessors of node l. Figure 2.4b shows the generation of the OUT set at a given node l. Algorithm 2.1 shows a general reaching definitions analysis algorithm. The algorithm starts by initiating IN and OUT set in the control flow graph to the empty set. The entry node OUT set denoted by $OUT[entry]$ is initialized to contain a hypothetical definition for each variable v in the program. ? denotes that each variable v is undefined or uninitialized at the start of the program. The algorithm

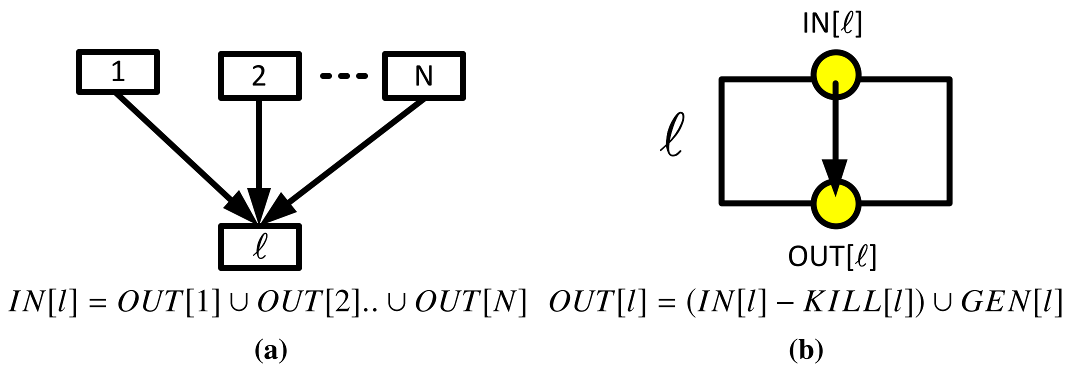

1
2
N
ℓ
IN[ℓ]
OUT[ℓ]
ℓ
$IN[l] = OUT[1] \cup OUT[2].. \cup OUT[N]$
$OUT[l] = (IN[l] - KILL[l]) \cup GEN[l]$
(a)
(b)

Figure 2.4: Illustration of the generation of (a) IN set and (b) OUT set [5].

then repeats for each node l to calculate the IN and OUT set of each node l and terminates when IN and OUT set for all l stops changing. Example 2.3 illustrates a working example of the algorithm.

Algorithm 2.1 Reaching definitions Analysis

```
 1: for each node $l$ do
 2:     $IN[l] = OUT[l] = \phi$
 3: end for
 4: $OUT[entry]$={$< v, ? >$ : $v$ is a program variable}
 5: repeat
 6:     for each node $l$ do
 7:         $IN[l] = \bigcup_{l' \in predecessor(l)} OUT[l']$
 8:         $OUT[l] = (IN[l] - KILL[l]) \cup GEN[l]$
 9:     end for
10: until $IN[l]$ and $OUT[l]$ stop changing for all $l$
```

Example 2.3 Consider a C program in Figure 2.1 with the corresponding labeled control flow graph in Figure 2.2. Our goal is to perform the analysis of the reaching definitions. Table 2.1 shows the final state of IN and OUT sets for all nodes l. The following is a step by step description of how the IN and OUT sets are obtained:-

- At the start node, the IN is initially empty. The OUT set is initialized to contain many variable-label pairs ⟨v,?⟩ where labels ? denotes unknown.
- After node 2, the variable z is defined. The information $\langle z, 2\rangle$ is then updated. The same applies for variable y at node 3.
- The interesting step is when the analysis encounter loops, for instance during the first iteration, for the definition of variable y, $IN[4]$ contains only the definition $\langle y, 3\rangle$ but not $\langle y, 6\rangle$ and $\langle y, 7\rangle$. This information is obtained at the subsequent iterations of the algorithm by taking the union of the information at nodes 3 and 8.

l	$IN[l]$	$OUT[l]$
1	...	$[\langle x, ?\rangle, \langle y, ?\rangle, \langle z, ?\rangle]$
2	$[\langle x, ?\rangle, \langle y, ?\rangle, \langle z, ?\rangle]$	$[\langle x, ?\rangle, \langle y, ?\rangle, \langle z, 2\rangle]$
3	$[\langle x, ?\rangle, \langle y, ?\rangle, \langle z, 2\rangle]$	$[\langle x, ?\rangle, \langle y, 3\rangle, \langle z, 2\rangle]$
4	$[\langle x, ?\rangle, \langle y, 3\rangle, \langle y, 6\rangle, \langle y, 7\rangle, \langle z, 2\rangle]$	$[\langle x, ?\rangle, \langle y, 3\rangle, \langle y, 6\rangle, \langle y, 7\rangle, \langle z, 2\rangle]$
5	$[\langle x, ?\rangle, \langle y, 3\rangle, \langle y, 6\rangle, \langle y, 7\rangle, \langle z, 2\rangle]$	$[\langle x, ?\rangle, \langle y, 3\rangle, \langle y, 6\rangle, \langle y, 7\rangle, \langle z, 2\rangle]$
6	$[\langle x, ?\rangle, \langle y, 3\rangle, \langle y, 6\rangle, \langle y, 7\rangle, \langle z, 2\rangle]$	$[\langle x, ?\rangle, \langle y, 6\rangle, \langle z, 2\rangle]$
7	$[\langle x, ?\rangle, \langle y, 3\rangle, \langle y, 6\rangle, \langle y, 7\rangle, \langle z, 2\rangle]$	$[\langle x, ?\rangle, \langle y, 7\rangle, \langle z, 2\rangle]$
8	$[\langle x, ?\rangle, \langle y, 6\rangle, \langle y, 7\rangle, \langle z, 2\rangle]$	$[\langle x, ?\rangle, \langle y, 6\rangle, \langle y, 7\rangle, \langle z, 2\rangle]$

Table 2.1: Final state of IN and OUT set for all nodes l performed by Reaching Definitions Analysis in the C program of Figure 2.1

- At nodes 6 and 7, the definition $\langle y, 3\rangle, \langle y, 6\rangle$ is killed at node 7 and the definition $\langle y, 3\rangle, \langle y, 7\rangle$ is killed at node 6.
- The last node 8 takes the union of nodes 6 and 7. □

Reaching definitions analysis is based on data flow analysis which computes an over-approximation of the pairs of reaching definitions at any program point. This means that the analysis reports all actual reaching definition pairs at any program point and may be more. Speaking in terms of soundness and completeness of the analysis, it is complete but not sound. The primary source of unsoundness of reaching definitions analysis is due to control conditions abstraction with non-deterministic choice. By non-deterministic choice, it means the analysis will assume that the condition may evaluate to either true or false, even if in actual runs, the condition always evaluates to only true [5]. Due to abstraction of branch conditions, the analysis will consider all paths that are possible in actual runs and thereby guarantee completeness, but at the same time consider paths that are never possible in actual runs, hence the analysis can be unsound.

Designing a sound and complete static analysis is hard and maybe impossible [27]. Approximation comes into rescue by allowing the analysis to reason the program approximately by providing an abstraction of the program's behavior. In the next section, we will discuss a technique for static analysis that is based on the mathematical *theory of lattices*.

Lattice theory

To motivate why we need lattice theory for our analysis, let us consider a simple analysis that determines the possible signs of the integer value of variables and expressions in the given program. In concrete executions, the values of the variables can be any arbitrary integer. The integer values can be abstracted and grouped into three categories: *positive* (+), *negative*

(-), and *zero* (0). The analysis will then operate on the abstract values instead of concrete values. However, the analysis must be prepared to handle uncertain information, for example, if the analysis does not know the sign of an expression, then a special abstract value ($\top$) representing *'don't know'* is added to represent such a case. There can be information that has no value in an execution, for example, values that are not numbers but instead pointers. In this case, a special abstract value ($\bot$) is added to represent such cases.

Example 2.4 Consider the program in Figure 2.5, where the analysis can conclude that, in all possible executions, at the end of the program, the variables a and b are positive integer values. On the other hand, the variable c can be either positive or negative depending on the value of n which is unknown. The analysis must report $\top$ for variable c, since variable c can be either positive or negative. For this analysis we have an abstract domain comprising of five abstract values $\{+, -, 0, \top, \bot\}$. The abstract domain can be organized with the least precise information at the top and the most precise information at the bottom as shown in Figure 2.6. The ordering illustrates the facts that $\bot$ represents the empty set of integer values and $\top$ represents the set of all integer values. $\square$

```
void main()
{
  int a=10;
  int b=20;
  int c;
  if(n==0) // n is user input
  {
    c=a+b;
  }
  else
  {
    c=a-b;
  }
}

```

Figure 2.5: Simple C program with three integer variables a, b and c.

We will now study some definitions related to lattice theory.

Definition 2.3 ***Partially ordered set [14]*** *: A partial ordering is a binary relation* $\sqsubseteq: L \times L \implies \{true, false\}$ *that is :*

- *Reflexive :* $\forall l : l \sqsubseteq l$
- *Transitive:* $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \land l_2 \sqsubseteq l_3 \implies l_1 \sqsubseteq l_3$
- *Anti-Symmetric :* $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \land l_2 \sqsubseteq l_1 \implies l_1 = l_2$

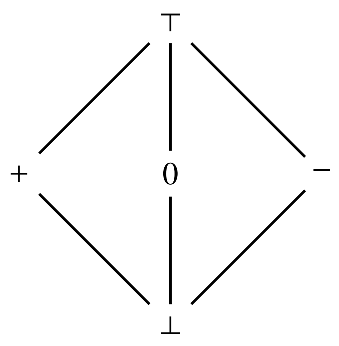


Figure 2.6: Hasse diagram representing abstract domain for sign analysis

A partially ordered set $(L, \sqsubseteq)$ *is a set L equipped with a partial ordering* $\sqsubseteq$. ■

Definition 2.4 ***Upper bound*** *[28] :For any* $S \subset L$, *an upper bound of a set S is any element* $x \in L$ *such that* $x \geq y, \forall y \in S$. ■

Definition 2.5 ***Lower bound*** *[28] :For any* $S \subset L$, *a lower bound of a set S is any element* $x \in L$ *such that* $x \leq y, \forall y \in S$. ■

Definition 2.6 ***Least upper bound [25]*** *: A least upper bound l of Y is an upper bound of Y that satisfies* $l \sqsubseteq l_0$ *whenever* l_0 *is another upper bound of Y denoted by* $\sqcup Y$. *Sometimes* $\sqcup$ *is called the Join operator.* ■

Definition 2.7 ***Greatest lower bound [25]*** *: A greatest lower bound l of Y is a lower bound of Y that satisfies* $l_0 \sqsubseteq l$ *whenever* l_0 *is another lower bound of Y denoted by* $\sqcap Y$. *Sometimes* $\sqcap$ *is called the Meet operator.* ■

Definition 2.8 ***Complete lattice [25]*** *: A complete lattice* $L = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *is a partially ordered set* $(L, \sqsubseteq)$ *where every subset A of L has both a least upper bound as well as a greatest lower bound.* ■

Every lattice has a unique *largest* element denoted by $\top$ and a unique *smallest* element denoted by $\bot$. The height of the lattice is defined to be the length of the longest path from $\bot$ to $\top$. As an example, the height of the sign analysis lattice is 2.

Definition 2.9 ***Monotone functions [14]*** *: Given a complete lattice L, a set of functions F on L is said to be a monotone function space associated with L if the following conditions are satisfied:-*

- *Each* $f \in F$ *satisfies the monotonicity condition,*

  $(\forall x, y \in L)(\forall f \in F)[x \sqsubseteq y \implies f(x) \sqsubseteq f(y)]$.

- *There exists an identity function I in F, such that*

$(\forall x \in L)[I(x) = x]$.

- *F is closed under composition, i.e. if $f, g \in F$ then the composition $f \circ g \in F$, where $(\forall x, y \in L)[f \circ g(x) = f(g(x))]$.* ■

Definition 2.10 ***Flow graph [29]*** *: A flow graph is a triple $G = (N, E, n_0)$, where:*

- *(N, E) is a finite directed graph.*
- *N is a finite set of nodes.*
- *$E \subseteq N \times N$ is the set of edges. The edge (x, y) enters node y and leaves node x, we say that x is the predecessor of y, and y is a successor of x. The edges represents the flow of control or data in the program.*
- *n_0 in N is the initial node.* ■

2.2.3 Data flow analysis with monotone frameworks

Classical dataflow analysis starts with a control flow graph and a lattice with finite height. For every node v in the control flow graph, a variable is assigned ranging over the elements of a lattice. A dataflow constraint is then defined for each node that relates the value of the variable of the node to other nodes. If all the constraints happen to be equations or in-equations with monotone right-hand sides, then a fixed point algorithm can be used to compute the analysis result as the unique least solution [14]. A dataflow analysis is required to compute the following information during the analysis:-

- $\mathcal{A}_{entry}(l)$ is the information that holds at the entry of a block.
- $\mathcal{A}_{exit}(l)$ is the information that holds at the exit of a block.
- $kill(l)$ holds information that is removed from the input.
- $gen(l)$ holds the information that is added to the input.

Data flow analysis can be broadly classified into the following categories [14]:-

Forward May Dataflow Analysis: At each program point, the forward may analysis computes information that holds at some path reaching the point starting from the initial node and following the edges of the flow graph. Figure 2.7 shows a diagram illustrating the forward data flow analysis. The analysis starts at the *entry* node and propagates information forward in the control flow graph. The analysis computes the following information:-

$$\mathcal{A}_{entry}(l) = \bigcup_{l_i \mapsto l} \mathcal{A}_{exit}(l_i) \tag{2.2}$$

$$\mathcal{A}_{exit}(l) = (\mathcal{A}_{entry}(l) - kill(l)) \bigcup gen(l) \tag{2.3}$$

$\mathcal{A}_{entry}(l)$ is computed as a union of all that may hold at the previous blocks. $\mathcal{A}_{exit}(l)$ is computed as the union of all facts at the previous blocks minus the facts that no longer hold in this block, union with the facts that are generated in this block.

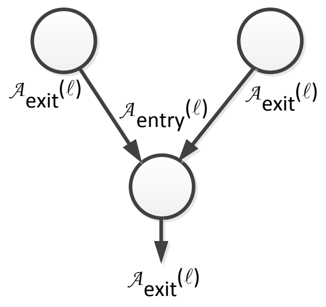

$\mathcal{A}_{exit}(\ell)$
$\mathcal{A}_{entry}(\ell)$
$\mathcal{A}_{exit}(\ell)$
$\mathcal{A}_{exit}(\ell)$

Figure 2.7: Forward analysis [30]

Backward May Data Flow Analysis: At each program point, the Backward May Data Flow Analysis computes information that holds at some path reaching the point starting from a terminal node and following the edges of the flow graph in the reverse direction. Figure 2.8 shows a diagram illustrating the backward data flow analysis. The analysis computes the following information:-

$$\mathcal{A}_{entry}(l) = (\mathcal{A}_{exit}(l) - kill(l)) \bigcup gen(l) \tag{2.4}$$

$$\mathcal{A}_{exit}(l) = \bigcup_{l_i \mapsto l} \mathcal{A}_{entry}(l_i) \tag{2.5}$$

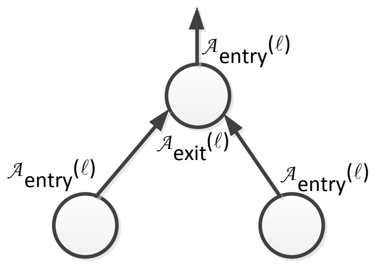

$\mathcal{A}_{entry}(\ell)$
$\mathcal{A}_{entry}(\ell)$
$\mathcal{A}_{exit}(\ell)$
$\mathcal{A}_{entry}(\ell)$

Figure 2.8: Backward analysis adapted from [30]

Forward Must Dataflow Analysis: At each program point, the forward must dataflow analysis looks for information that holds at all the paths reaching the program point starting from the initial program point and following the forward control flow edges. The analysis computes the following information:-

$$\mathcal{A}_{entry}(l) = \bigcap_{l_i \mapsto l} \mathcal{A}_{exit}(l_i) \tag{2.6}$$

$$\mathcal{A}_{exit}(l) = (\mathcal{A}_{entry}(l) - kill(l)) \bigcup gen(l) \tag{2.7}$$

Backward Must Dataflow Analysis: At each program point, Backward must dataflow analysis computes information that must hold at a program point by all of the paths starting at the terminal nodes and following the control flow edges in the reverse direction. The analysis computes the following information:-

$$\mathcal{A}_{entry}(l) = (\mathcal{A}_{exit}(l) - kill(l)) \bigcup gen(l) \tag{2.8}$$

$$\mathcal{A}_{exit}(l) = \bigcap_{l_i \mapsto l} \mathcal{A}_{entry}(l_i) \tag{2.9}$$

Definition 2.11 *A Monotone framework consists of [14], [25], [29], [30],*

- *A complete lattice L of the framework. The lattice L should satisfy the condition that each ascending chain $a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq \ldots$ is finite, i.e., there is an n such that $a_n = a_{n+1} = a_{n+2} = \ldots$*
- *A set $\mathcal{F}$ of monotone functions $f : L \longrightarrow L$ that contains the identity function $id : L \longrightarrow L; a \mapsto a$ and is closed under composition, i.e., if $f, g \in \mathcal{F}$ then the composition $f \circ g \in \mathcal{F}$.*
- *$G = (N, E, n_0)$ is a flow graph*
- *$M : N \longrightarrow \mathcal{F}$ is a function which maps each node in N to a function in $\mathcal{F}$.* ■

Lemma 2.1 *In a lattice L with finite height, every monotone function $f : L \longrightarrow L$ has a unique least fixed-point denoted $fix(f)$ defined as :*

$$fix(f) = \bigsqcup_{i \geq 0} f^i(\bot) \tag{2.10}$$

Proof 2.1 *Observe that $\bot \sqsubseteq f(\bot)$ since $\bot$ is the least element. Since f is monotone, it follows that $f(\bot) \sqsubseteq f^2(\bot)$ and in general , $f^i(\bot) \sqsubseteq f^{i+1}(\bot)$ for any i. Thus, we have an increasing chain:*

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots$$

Since L is assumed to have finite height, for some k, we must have that $f^k(\bot) = f^{k+1}(\bot)$, i.e., $f^k(\bot)$ is a fixed point for f. Let $fix(f) = f^k(\bot)$. Assume now that x is another fixed-point. Since $\bot \sqsubseteq x$ it follows that $f(\bot) \sqsubseteq f(x) = x$, since f is monotone, and by induction we *get that $fix(f) = f^k(\bot) \sqsubseteq x$. Hence, fix(f) is the unique least fixed point [14].* ■

Definition 2.12 *The analysis equations corresponding to a monotone framework are*

$$\mathcal{A}_{entry}(\ell) = \bigsqcup\{\mathcal{A}_{exit}(\ell') : (\ell', \ell) \in \mathcal{F}\} \sqcup \begin{cases} c, & \textit{if } \ell \in E. \\ \bot, & \textit{if } \ell \notin E. \end{cases} \tag{2.11}$$

$$\mathcal{A}_{exit}(\ell) = f_\ell(\mathcal{A}_{entry}(\ell)). \tag{2.12}$$

■

In general, we can generalize the following equations:-

$$\mathcal{A}_{entry}(\ell) = \begin{cases} c, & \text{if } \ell \in E. \\ \bigsqcup\{\mathcal{A}_{exit}(\ell') : (\ell', \ell) \in F\}, & \text{otherwise.} \end{cases} \tag{2.13}$$

$$\mathcal{A}_{exit}(\ell) = f_\ell(\mathcal{A}_{entry}(\ell)) \tag{2.14}$$

Where, depending on the specific analysis :

- The operator $\bigsqcup$ is either $\bigcap$ or $\bigcup$ set operators for joining information from the source nodes.
- A finite control flow graph of the program P, $flow(P)$. $flow(P)$ is either forward or backward control flow.
- Set of initial labels E, containing the labels of statements of program P of either the initial nodes or the set of final nodes.
- c specifies the initial value of the analysis at the initial or final nodes in forward or backward analysis respectively.
- f_l is the transfer function for the node l. A transfer function updates the values of the lattice L on which it is defined.

2.2.4 Dynamic Program Analysis

Our contributions in sensitivity analysis presented in this thesis in Chapter 3, Chapter 5 and Chapter 6 are essentially methods of dynamic analyses. We therefore present here an essence of dynamic program analysis. Dynamic program analysis refers to the method of analyzing applications at run-time [31]. The analysis discovers the properties of the program by examining one or more runs of the program. Typically, the program is instrumented and compiled to an executable and then the instrumented program is executed to perform the analysis. Dynamic analysis derives properties which hold for one or more executions of the program. Two essential characteristics that describe the usefulness of dynamic analysis are [32]:-

- **Precision of Information**: Since dynamic analysis examines concrete program executions, the analysis results are precise.
- **Dependence on program inputs**: The effectiveness and reliability of the analysis mostly depends on the sufficiency of the test inputs. With dynamic analysis, it is straight forward to relate changes in program inputs to changes in internal program behavior and program outputs [32].

The following are the common steps in most dynamic analysis techniques [33]. We now briefly discuss these steps in the following:-

- Program instrumentation.
- Profile/Trace Generation, and
- Analysis or monitoring.

Program instrumentation

Program instrumentation is the process of inserting additional code in the program for the purpose of generating program traces or for monitoring. During execution, these additional codes collect traces of the running program for analysis. Dynamic instrumentation can also be used for debugging purposes by adding instrumentation checkpoints to the program. Depending on the platform and program representation, instrumentation can be performed either at the source, binary or byte code. Example of dynamic binary instrumentation tools are Intel Pintool [34] and Valgrind [35]. LLVM compiler infrastructure [36] can be used for both source-to-source code transformation and for an intermediate representation of program code. ASM [37], JavaAssist [38] and BCEL [39] are tools for byte-code instrumentation.

Profile/Trace Generation

After instrumentation, the program is then executed with a given input set. The execution generates traces of the program that can be used for analysis.

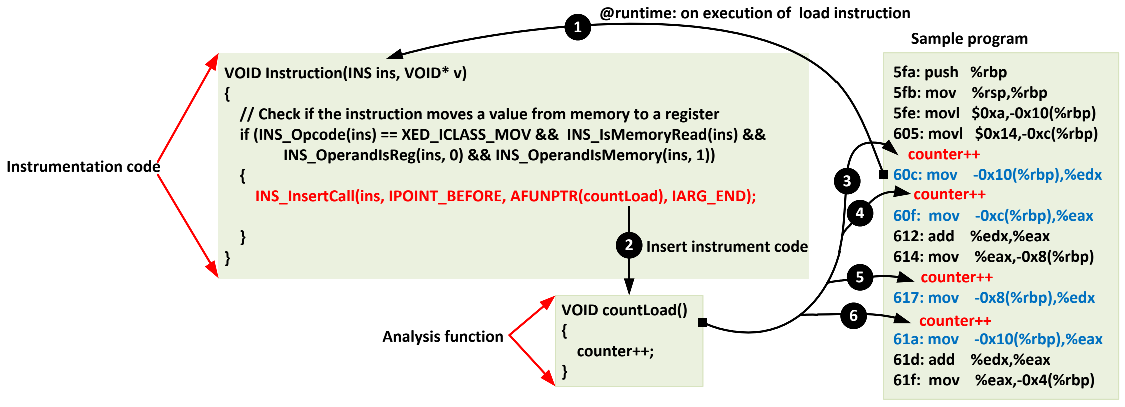
Analysis/Monitoring

The collected program traces are then used for analysis such as debugging and performance analysis. In case of monitoring, the required properties of the program states are captured for analysis.

Example 2.5 We illustrate a simple dynamic binary instrumentation using Intel Pintool [34]. Consider a simple analysis tool that counts the number of dynamic load instructions in the given program. The two basic steps for writing the Pintool are described as follows:-

- **Instrumentation**: The first step is to locate where to insert the instrumentation code in the given program binary. Pin API provides functions for inserting code at a specific location in the binary code. For instance, **INS_InsertCall** is a function for inserting code just before or after an instruction. The instrumentation code is called only once at runtime.

- **Analysis function**: The analysis function aids the instrumentation during runtime by executing the code that is inserted by the instrumentation. The analysis function is called multiple times depending on the running program.



```
VOID Instruction(INS ins, VOID* v)
{
   // Check if the instruction moves a value from memory to a register
   if (INS_Opcode(ins) == XED_ICLASS_MOV &&  INS_IsMemoryRead(ins) &&
           INS_OperandIsReg(ins, 0) && INS_OperandIsMemory(ins, 1))
   {
     INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(countLoad), IARG_END);

   }
}
```

```
VOID countLoad()
{
   counter++;
}
```

```
5fa: push   %rbp
5fb: mov    %rsp,%rbp
5fe: movl   $0xa,-0x10(%rbp)
605: movl   $0x14,-0xc(%rbp)
 counter++
60c: mov    -0x10(%rbp),%edx
 counter++
60f:  mov   -0xc(%rbp),%eax
612: add    %edx,%eax
614: mov    %eax,-0x8(%rbp)
 counter++
617: mov    -0x8(%rbp),%edx
 counter++
61a: mov    -0x10(%rbp),%eax
61d: add    %edx,%eax
61f:  mov   %eax,-0x4(%rbp)
```

Figure 2.9: Dynamic binary instrumentation using Pintool

Figure 2.9 shows the workflow of the Intel Pintool. ❶ On executing the load instruction **mov -0x10(%rbp),%edx**, the tool captures the instruction and checks if it is a load type. If the instruction is a load type, ❷ then the tool inserts the instrumentation code ❸ that will call the analysis routine (countLoad()) at runtime. Similarly, for other load instructions, the instrumentation code inserts the analysis code in the corresponding locations ❹ ❺ ❻. After the program execution, the tool collects the information required for further analysis. Pintool provides an API function PIN_AddFiniFunction(onExit, 0) to collect such information. The function onExit is a user defined function to capture information on program exit. □

In the following discussion, we present an overview of speculative execution that provides a platform for our approximability analysis to be adopted inside modern processors.

2.3 Speculative Execution in modern processors

We now present a brief overview of modern processors, specifically with respect to the speculative execution scheme, with which we connect the outcome of our approximability analysis. To increase the flow of instructions into the processor and hide memory latency, modern processors employ pipelines to improve the average number of instructions executed per clock cycle. An execution pipeline is divided into stages, which allows multiple instructions to be overlapped in the pipeline. Consider a simple pipeline architecture consisting of five stages i.e., *Fetch*, *Decode*, *Execute*, *Memory* and *Write back* stage [40]. In the Fetch stage, the instruction is fetched from the memory. The Decode stage decodes the instruction to produce the control signals. In the Execute stage, the processor performs the execution. In the Write stage, the processor performs reads or writes from or to the memory and finally, in the Write back stage, the processor writes the result to the register file if necessary. Figure 2.10 shows a simple 5-stage pipeline. In this example, each instruction is expected to require 5 cycles in order to complete execution.

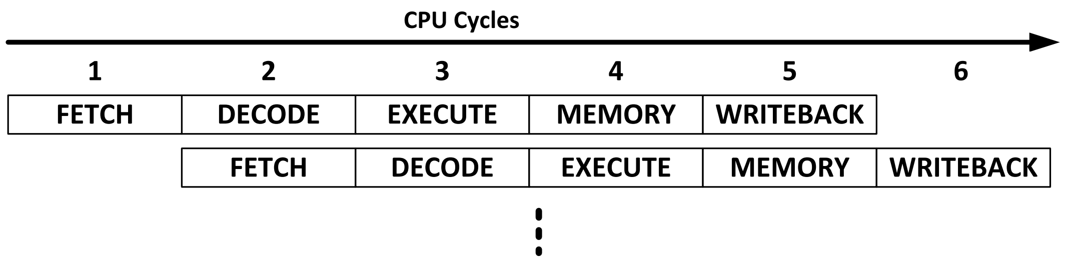


Figure 2.10: 5-stage pipeline processor

The performance of a pipeline is significantly affected by hazards, which can prevent a pipeline stage to complete the execution of an instruction in one clock cycle. The hazards cause the pipeline to stall. The three general types of pipeline hazards are described in the following.

- **Structural hazards**: arise due to hardware resource conflicts.
- **Data hazards**: occur due to dependency of instructions on the results of the previous instructions.
- **Control hazards**: arise due to change in the control of instruction stream.

Hazards significantly impact pipeline performance. To avoid potential stalls and hazards, modern processors employ optimization techniques, an example being that of speculative execution, which manifests either as data speculation or control speculation as discussed below. By executing instructions speculatively, the overall performance can be increased by avoiding pipeline stalls and reducing the waiting time of the processor i.e., without the need to wait for prior instructions to be resolved before executing the subsequent ones. It is used pervasively to improve the efficiency of all modern CPUs. Speculative execution can be implemented using a combination of both hardware and software techniques. We discuss briefly on control and data speculation in the following.

2.3.1 Control Speculation

In this case, the processor executes control-dependent instructions before resolving the branch outcome. Control speculation is of two broad types; speculating on the direction of a branch (taken / not-taken), and speculating on the target of the branch (potentially anywhere in the program's address space). Control changing instructions such as branches add difficulty in the execution of dependent instructions and consequently lead to large performance loss and energy wastage in pipelined processors. To overcome this problem, modern processors employ a prediction mechanism to predict the outcome of the branch instructions and based on the prediction of the predictor, the processor executes instructions speculatively. Control speculation heavily relies on the *branch prediction* techniques for predicting the outcomes of the branches. Branch predictors employ prediction algorithms in an effort to predict whether a branch will be taken or not taken. Branch predictors play an important role in performance and energy reduction in modern processors. If the prediction from the predictor is correct, it improves performance since the processor does not need to wait for the branch to get resolved and start execution on the predicted path. However, in case of a mis-prediction, the pipeline has to be flushed and all instructions that enter the pipe due to the mis-prediction have to be flushed and execution needs to be rolled back to the point where the mis-prediction happens. This wastes a lot of energy, since the execution of the instructions on the wrongly predicted path, turn out to be useless.

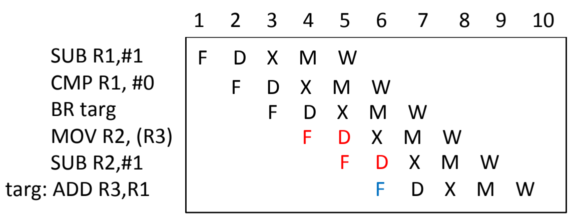
A number of branch prediction schemes have been explored in literature from simple prediction techniques [41], [42] which provide fast lookup and power efficiency but suffer high misprediction rate, to complex techniques such as neural based [43] which provide better accuracy but consume more power and increase the complexity. A branch predictor can be either static, dynamic or a combination of both. Static branch predictor techniques are programmer-based or profile-based and rely on the information at compile or at pre-execution time. For profile-based, the accuracy mostly depends on the representativeness of the profile input set, whereas for programmer-based, the accuracy largely depends on the heuristic of the programmer. Dynamic branch predictor techniques rely on dynamic information collected at run-time which adapts to changes in branch behavior and requires additional complex hardware.

Example 2.6 We now consider an example illustrating control speculation. Figure 2.11 shows a code snippet of an assembly code. Figure 2.12 shows the instructions that enter and exit the pipeline in all 5-stages. Figure 2.12(a) shows the execution of instructions in the correct predicted path and hence no flushing or rollback is required. Figure 2.12(b) shows the execution of the instruction in the incorrect path. On branch resolution, 2 instructions are required to be flushed from the pipeline. □

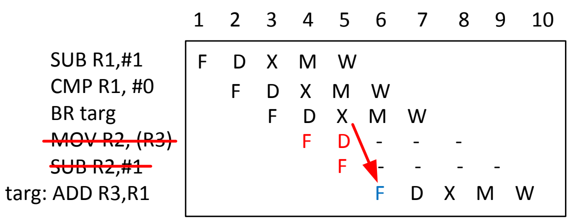
```
    SUB R1,#1
    CMP R1, #0
    BR targ
    MOV R1, (R3)
    SUB R2,#1
    targ: ADD R3,R1

```

Figure 2.11: Simple assembly code



(a) Correction prediction



(b) Recovery due to misprediction

Figure 2.12: Simple machine code executing in a 5-stage pipeline

Branch Prediction

Branch predictor is a component that predicts the target instruction of a branch in advance before it is resolved. Using branch prediction, the decision of the control flow is made in the fetch stage. The pipeline executes instructions in the predicted path and after the branch direction is resolved, in case of a detected mis-prediction, the instructions in the wrongly predicted path are flushed from the pipeline and the right instructions from the correct path are fetched. Since branch misprediction incurs high latency for about 14 to 15 cycles as reported in [44], the performance benefits in using such schemes depend on whether the prediction is correct and how soon it can check the prediction. Modern branch predictors have more than 95% accuracy and can reduce branch penalty on mispredictions significantly

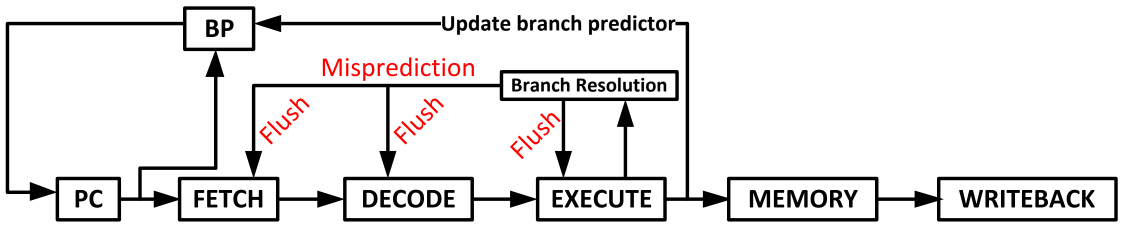


Figure 2.13: 5-Stage pipeline with branch predictor

and thus save energy and improve performance. Figure 2.13 shows a 5-stage pipeline with a Branch Predictor (BP). When a branch instruction is encountered during the fetch stage, the branch predictor predicts whether the branch is taken or not-taken and the corresponding instructions from the predicted path are fetched accordingly. When the branch gets resolved at the execute stage, the prediction is then checked for correctness. If the prediction is wrong, the pipeline needs to be flushed and required to fetch the instructions from the correct path. When the branch instruction exits the pipeline, the branch predictor updates its confidence of the prediction.

2.3.2 Data Speculation

Data dependent instructions require the results of the previous instructions, and hence often lead to pipeline stalls, waiting for values to be fetched from memory in case of cache misses. Data speculative execution allows such dependent instructions to execute with predicted values. The two broad types of data speculation are those that speculate on the address of the data, and those that speculate on the actual value of the data.

Data speculation typically exploits value locality i.e., predicts the results of instructions based on previously seen results. Often programs reuse values in the same instruction multiple times or reuse in subsequent instructions. The observation that a dynamic instruction may often produce the same result as in the previous instance is known as value locality [45]. Data speculation concerns prediction of either the location of data or the value of data. Value prediction depends on both spatial and temporal locality. The main idea in value prediction is to predict the results of instructions based on previously seen results. In the pipeline, the prediction is performed during the Fetch and Decode stages and prediction is verified before committing the result of an instruction. If a misprediction occurs, the processor has to re-execute the instruction.

Value Prediction

Data value prediction predicts data values and speculatively uses them in destination instructions. This increase the Instruction Level Parallelism (ILP) by allowing multiple dependent instructions to be executed in the same clock cycle. However, the predictor needs to predict accurately, since inaccurate predictions are costly to handle and recover from. Figure 2.14 shows a simple 5-stage pipeline with a value predictor. The value predictor predicts values during the fetch and decode stages and the predicted value is forwarded to dependent instructions. Before committing the instruction, it must be verified. If a mis-prediction happens, it must restart the instruction and execute with the correct values.

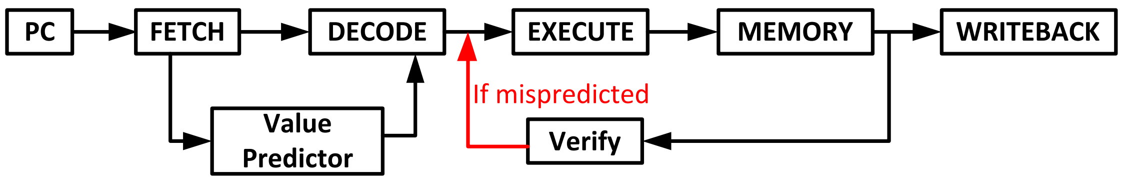


Figure 2.14: 5-Stage pipeline with value predictor.

As explained in Chapter 5, we utilize the speculative execution feature of modern processors with an objective of improving performance, by not requiring the processor to rollback in case of mis-prediction. In particular, we work closely with both the value and branch predictors to relinquish roll-backs when there is a mis-prediction on approximable data or branch instructions.

2.4 Related Work

In this section, we review some of the key techniques related to our work. We begin with a literature survey on approximate computing. We first introduce software approaches to approximation and then present techniques that use both software and hardware approaches.

2.4.1 Software Techniques for Approximate Computing

At the algorithm and programming language level of the computing stack, various approximation techniques in software exist in literature. These techniques can be broadly classified into two categories; *viz. manual-guided approximation* and *automated approximation*. The manual-guided approximation requires programmer interventions and efforts, whereas automated approximation requires minimal programmer interventions and efforts. In the following subsection, we discuss in brief, the various software techniques that are of relevance to this thesis, consisting of both manual and automated approximation.

Manual-guided Approximation

Manual-guided approximation techniques require the programmer to annotate parts of the program that are amenable for approximation. The following discussion presents some of the common manual-guided techniques and frameworks.

Task Skipping [46]: This technique handles errors or faults in the program at runtime by continuing execution of the remaining computations even if there are faults being encountered. Task skipping enables computations to survive errors and faults while providing a bound on the resulting output distortion. The technique first partitions the computations into tasks. The execution framework on encountering an error or a fault, simply discards the task and completes the computation by executing the remaining tasks. To guarantee the limit of

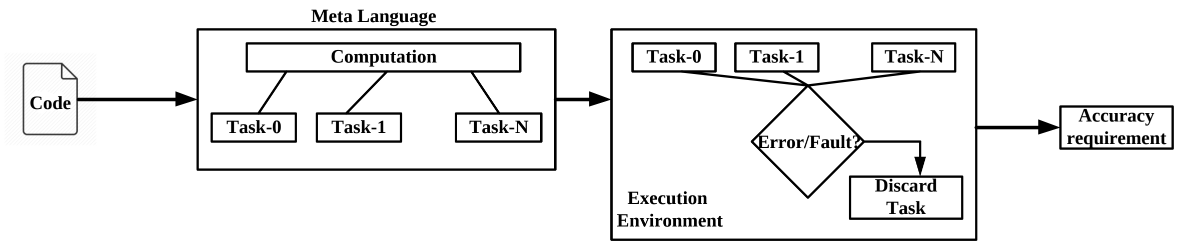


Figure 2.15: Task skipping high level overview

distortion of the output, the technique offers a *probabilistic distortion model* that characterizes the distortion of the output as a function of the failure rates in the computation and hence provides a *probabilistic bound* on the distortion. The user can then evaluate this bound, to ascertain whether the output satisfies the accuracy requirements.

Figure 2.15 shows a high-level overview of the task skipping technique. First, given any standard programming language such as C or Java, the programmer uses a Metalanguage (for example, the Jade metalanguage [47]) to partition the computation into tasks ($Task - 0$ to $Task - N$ as in the figure). In the second step, when a task encounters an error (hardware or software fault or both) at runtime, the execution environment simply discards the task, and picks the remaining tasks to complete the computation. Following are the basic steps of this approach.

- *Task decomposition*: The programmer uses the meta-language to identify and mark task blocks in a program that can be skipped.
- *Critical Testing*: The execution platform is configured to randomly fail executions of selected task blocks at target failure rates and observe the resulting output distortion. If the failures produce unacceptable distortion, it marks the task block as *critical*, else marks as *fault tolerant*.

- *Distortion model*: Randomly selects a fault tolerant task and runs repeated trials, executes the computation, and then records both the observed task failure rates and the resulting output distortion.
- *Timing model*: The execution time of the program for each trial is recorded and then it uses regression techniques to obtain a model that estimates the execution time as a function of the task failure rates.

The difference in our work is that we focus on studying the effect of approximate *data* storage in a computation instead of studying the effect of approximations in tasks. Therefore, our work may be seen to target approximation in more elementary parts of a computation, namely data, as against targeting approximation in a task, which may constitute both data and instructions together. The authors in [48] conclude that tasks which involve data move and store are generally critical, that is, not fault tolerant. This is because data errors propagate in the following computation and accumulate to produce unacceptable results. We find that within a program, even certain data can be categorised as fault tolerant and therefore, faults in even move and store on these approximable data can be tolerated. Therefore, our work in this thesis can further supplement the task-approximation proposed in [48], by going a step further and categorize certain tasks as fault tolerant, that involves stores or moves on approximable data only.

Manual Annotation of Programs using EnerJ [48]: This technique supports the Java programming language constructs for approximate computing. The *EnerJ* framework provides type qualifiers to isolate parts of the program that must be *precise* from those that can be *approximated.* They employ static analysis techniques to statically guarantee isolation of the precise program component from the approximate one. Using these type qualifiers, the compiler automatically maps approximate types to approximate storage or computing components. *EnerJ* is implemented as an extension to Java by adding additional annotations to incorporate the appropriate annotations.

- *Type annotations:* An approximate program has both approximate and precise types. Using *EnerJ*, a programmer can annotate approximate and precise variables with the **@Approx** and **@Precise** constructs respectively. The precise types are by default. The assignment from approximate-typed value into a precise-typed variable is illegal, however, an assignment from precise to approximate is allowed. Figure 2.16 shows some example assignments.

  For assigning of an approximate type to a precise one, a static function called *endorse* allows the programmer to use approximate data as if it were precise. For instance, if an application consists of a resilient image manipulation phase followed by a critical

```
    @Approx int x=...;
    int y;
    y=x; // Illegal assignment, from approximate-typed value to
precise variable.
    y=...;
    x=y; // Legal assignment, from precise-typed value to
approximate variable

```

Figure 2.16: *EnerJ* assignment statements.

checksum over the result, an *endorse* function is useful. Figure 2.17 shows one such example.

```
    @Approx int x=...;
    int y; //precise by default
    y=endorse(x); // legal

```

Figure 2.17: EnerJ *endorse* function usage example

- *Approximate operations*: Approximate computation is achieved by overloading operators and methods based on the type qualifiers, e.g. the + operator on integers; the + operation on two approximate operands producing an approximate integer. However, the + operation applied on two precise operands will produce a precise integer.

- *Control flow*: The framework disallows *implicit flows* that occur via control flow. The following example violates the desired isolation property. The EnerJ language prohibits

```
    @Approx int x=...;
    boolean flag; // precise
    if(x==5){flag = true;} else {flag = false;}

```

Figure 2.18: EnerJ example in conditions that affect control flow

approximate values in conditions that affect control flow. However, the static function *endorse()* enables a programmer to allow approximate values in control flow if needed. Figure 2.18 shows one such violation in conditions that affect the control flow.

- *Execution model:* The framework leverages both approximate *storage* and approximate *operations*. The hardware model offers approximate storage in the form of unreliable registers, data caches, and main memory. Approximate ALU operations are specified using appropriate approximate instructions. The approximate instructions give hints to the architecture that it may apply various energy-saving approximations when executing the given instruction.

- *Layout of approximate data*: The hardware model supports approximate memory data at a cache line granularity by having a bit per line in each page that specifies whether the corresponding line is approximate. A runtime system is used to segregate approximate and precise elements in different cache lines. A simple technique has been proposed for laying out of objects with both approximate and precise fields. The precise portion of the object is contiguously laid out first. Then, just after the end of the precise line, the approximate fields are laid out.

In this thesis, we propose methods that automatically identify program data that are approximable, unlike in EnerJ where a programmer has to explicitly annotate approximable data with type qualifiers. Thus, our work can enhance EnerJ like paradigm by reducing the responsibility of programmers.

Automated Approximation

Manually determining and classifying parts of a program that are amenable for approximation is a daunting and error-prone task. At the same time, it requires programmer efforts and domain expertise. Automated techniques emerged in order to ease approximate computing practitioners in applying approximate computing techniques at the language level. In this section, we discuss various automated techniques for classifying and determining parts of a program amenable for approximation.

Automatic Sensitivity Analysis for Approximate Computing (ASAC): This technique [49] aims to automatically discover approximable program data using statistical methods. The main idea is to systematically perturb variables and then observe the resultant output sensitivity. It consists of 3 main stages, namely *discovery*, *probe* and *testing*.

- *Discovery* : During the discovery stage, the framework extracts the variables of a program along with a range of values that each can expect during execution. Using range analysis, for each variable V_i, the range of V_i is given by range(V_i)=[R_{i1}, R_{i2}], where $R_{i1} \leq value(V_i) \leq R_{i2}$. If the range analysis cannot determine the range of values, range(V_i) is given by the data type of V_i. To calculate the value ranges of variables, the range analysis employs widening and narrowing operator based on data-flow analysis [50]. The Cartesian product of the variable range intervals is given by $H = [R_{11}, R_{12}] \times [R_{21}, R_{22}] \times ... \times [R_{n1}, R_{n2}]$, where $[R_{i1}, R_{i2}]$ is the range of variable (V_i). This Cartesian product forms an *n-dimensional hyperbox*. Each dimension of the hyperbox represents a variable and the corresponding edge is the range of the variable.

The number of dimensions in the hyperbox is determined by the number of variables in the program.

- *Probe*: The hyper-box constructed in the discovery stage represents the sample space for the statistical experiments. The hyper-box is divided into smaller hyperboxes of equal sizes. These small hyperboxes are obtained by discretization of edges and selecting only a subset from among them. A subset of these smaller hyperboxes is selected as samples using the Latin Hyperbox Sampling (LHS) algorithm [51]. From each sampled hyperbox, m points are chosen uniformly at random. Each point is a n-tuple coordinate containing the values of each variable at that point, where n is the number of variables in the program. During the execution of the program, these points are passed to the program and the values are perturbed forcefully to corresponding variables using a dynamic instrumentation tool. A program execution with this perturbation is called *a probe run*. Due to the perturbation, the program output can be expected to deviate from the correct output. Based on the difference between the QoS threshold of the application and the perturbed outputs, each such sample is marked as *"good"* (pass) or *"bad"* (fail). Consider P_i is a vector of all outputs of all the probe runs of sample S_i, f_{obj} is an objective function, and θ is a constant threshold. We test if $f_{obj}(P_i) \geq \theta$ and if so, it designates P_i to be a "good" sample, else marks it as a "bad" sample. The objective function is defined as $f_{obj} = (\sum_{j=0}^{j=k} w(P_i))/k$ where $w(P_i) = 1$ if $P_i \geq T_{qos}$, otherwise $w(P_i) = 0$. T_{qos} is the user-defined QoS threshold for the application.

- *Testing*: For each dimension of the hyperbox, a cumulative distribution curve for good and bad samples is constructed by plotting the number of good and bad samples against the range of values of that dimension. The distance between the two curves denotes the contribution of the variable towards the program output. The maximum distance between the two curves is calculated using the *Kalmogorov-Smirnov* hypothesis test [52]. This distance is called the d-statistics, and it translates to the sensitivity ranking. The larger is the distance, the higher is the sensitivity of the output to the variable of that dimension and vis-versa.

Figure 2.19 shows a high-level overview of the ASAC framework. The program under analysis consists of 3 variables i, a, and sum. ❶ The analysis constructs a hypercube where the dimension of the hypercube is the range of values that the variable can take at runtime. ❷ The variable's ranges are fine-tuned using range analysis. ❸ The analysis then picks a point from the hypercube, for instance, the analysis picks $sum = 3.0$, $a = 0.2$ and $i = 4$ for perturbation. The application executes and ❹ measures the QoS loss with respect to the acceptable QoS. ❺ Using the results from the perturbation runs, the cumulative distribution curve is constructed and ❻ hypothesis testing is performed to infer the sensitivity of the

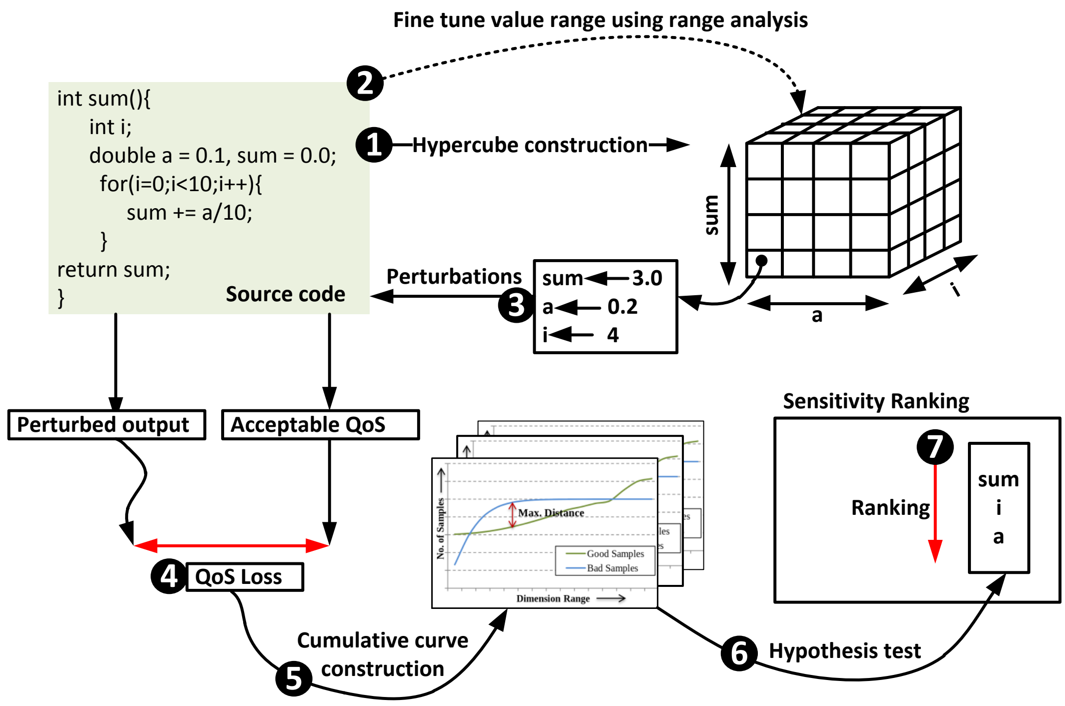


Figure 2.19: Overview of ASAC: Automatic Sensitivity Analysis for Approximate Computing Framework [49]

variables. 7 Finally, the sensitivity ranking of variables is performed.

In comparison with manual annotations by EnerJ [48], this technique of automated inferencing of approximable program data shows 86% accuracy in identifying approximable variables. In terms of scalability, the analysis is able to analyze large applications such as JPEG and H.264. The contributions of this thesis are most similar to ASAC which also proposes a method for automatic sensitivity analysis of program data using statistical methods. The sensitivity results reported, however, have no reliability guarantee. In addition, the number of program executions in the presence of perturbations required for performing the statistical test is chosen in an ad-hoc manner. The number of samples chosen influences the reliability and efficiency of the results. The proposed methods in this thesis not only provide probabilistic guarantees on the sensitivity classification but also require an optimal number of program executions for a desired confidence level of the analysis. Moreover, a hybrid static-dynamic sensitivity analysis is also proposed with an improvement in the efficiency.

Automatically Identifying Critical Input Regions and Code [53]: This technique proposes a system called *Snap*. *Snap* automatically identifies critical input regions and code in applications. It automatically groups related input bytes into fields. For each field and the corresponding regions of code that access data derived from that field, *Snap* classifies the field as either *critical* or *forgiving*. Snap instruments the application executions to generate trace information. The execution trace consists of the sequence of executed branch instructions, the influence trace records for each executed instruction, the input bytes that influence the operands of the instruction. *Snap* performs the following steps:

- *Baseline execution* : To record the baseline execution and influence traces, *Snap* executes an instrumented version of the application on a set of representative inputs. These traces are considered the normal behavior of the application.
- *Input Specification Generator* : Using the baseline influence traces, *Snap* groups adjacent input bytes into fields. Consider two adjacent input bytes i and j. Let E_i be the number of executed instructions whose operands are influenced by i but not influenced by j and E_j similarly be the ones influenced by j but not influenced by i. Let E_{ij} be the number of executed instructions whose operands are influenced by both i and j. The *Affinity* A_{ij} of i and j is defined as $A_{ij} = \frac{E_{ij}}{E_i+E_j+E_{ij}}$. If $A_{ij} \geq 0.75$, the input specification generator groups i and j into the same field. The effective threshold value 0.75 is determined empirically and found robust in practice.
- *Instrumented Executions on Fuzzed Inputs:* Given an input and a grouping of the bytes into fields, a suite of fuzzed inputs is produced by fuzzing each field input in turn. Each fuzzed input is the same as the original input, with the exception for the value in the fuzzed input field, which is set to an extreme value (all 0s or all 1s). The application runs on each fuzzed input to produce a suite of fuzzed execution traces, one for each fuzzed input.
- *Field Classification* : Using the baseline execution traces, the baseline influence traces, and the fuzzed execution traces, *Snap* uses hierarchical agglomerative clustering to classify each input field as either critical or forgiving. First, it uses the *behavioral distance* described below to check if the value of the field substantially influences the set of basic blocks that the application executes. Secondly, if the field is not classified as critical, it uses the *output influence* to determine if the field influences one or more operands of a substantial proportion of the executed instructions. If so, it classifies the field as critical or output critical. Otherwise, it classifies the field as forgiving.

  The *Behavioral distance* measures the similarity between the baseline execution and the execution on a fuzzed input of an application. The distance is a number in [0,1], 0 implies that the executions had identical behavior and 1 implies no behaviors in common. The behavioral distance is given by $D_{ij} = \frac{|B_i \triangle B_j|}{|B_i \cup B_j|}$ which is the normalized Hamming distance between the two sets of the executed basic blocks, where B_i is the set of basic blocks for a baseline execution. B_j is the set of basic blocks for execution on a fuzzed input. Using this information in combination with hierarchical agglomerative clustering [54], it classifies the input fields into critical and forgiving. The *Output Influence :* is computed by inspecting the influence trace of the baseline execution. It captures the influence of the input field on the quality of the output. It is calculated as the proportion of the executed instructions that have at least one operand whose value in the field influences. For a threshold 0.1 (10%), if the output influence exceeds the threshold, the input field is classified as critical.

- *Code classification*: *Snap* uses hierarchical agglomerative clustering [54] for classification of the executed basic blocks. Using the input fields for classifications and the baseline influence traces for all representative inputs from the previous steps, it classifies each basic block as critical or forgiving. A basic block is classified as critical if the operands of its instructions are derived primarily from critical input fields. On the other hand, if the operands of the instructions are primarily derived from forgiving inputs fields, a basic block is classified as forgiving.

As shown with experimental results on three benchmark applications i.e, GIF, PNG and JPEG, *Snap* makes classification with significant precision and recall.

The methods in this thesis are different from *Snap* because the focus here is to detect approximable data, that is not an input to the application of interest. Also, unlike *Snap* that marks code blocks as critical or approximable, the methods in this thesis do not analyze code regions for their approximability.

Application Resilience Characterization [55]: This technique proposes a framework called *ARC* : Application Resilience Characterization [55]. *ARC* partitions a given application into approximable and sensitive parts. Additionally, the framework characterizes potential approximable parts to evaluate the applicability of various approximate computing techniques. The framework consists of two major steps :

- *Potential resilient (approximate) computation identification*: As the first step, the instructions in a given program are partitioned into computation kernels. The method considers the innermost loops as atomic kernels. During program execution, random errors are injected in the program variables that are modified in a kernel and used in the kernel outputs. If the application crashes or hangs, or produces an output not within the user Quality of Service, the kernel is marked as sensitive, else marked as approximable.

- *Resilient computation characterization through approximation models*: The resilient kernels obtained from the previous step are then characterized for resiliency. The strategy for error injection is similar to the previous step with two key differences. First, the errors injected in the kernels are derived from approximation models that model the effects of various approximate computing techniques. The approximation models and techniques used for characterizations are Arithmetic operations, data representation, and approximations carried out at the algorithm level.

Experimental results on 12 recognition, mining and search applications show an average 83% of the application's run-time is spent in resilient kernels, out of which, 74% of the run-time is dominated by a single kernel called the dominant kernel. Therefore, the majority of resilience can be exploited by focusing on approximate computing design efforts on the dominant kernel.

Our focus in this thesis is different in the sense that we propose methods for identifying resilient data and resilient load and branch instructions only and not for identifying resilient code fragments or kernels.

Loop perforation [56]: The simplest form of automated approximation proposed in literature is Loop perforation. Loop perforation transforms loops to execute a subset of their iterations, with the intention of trading accuracy for performance.

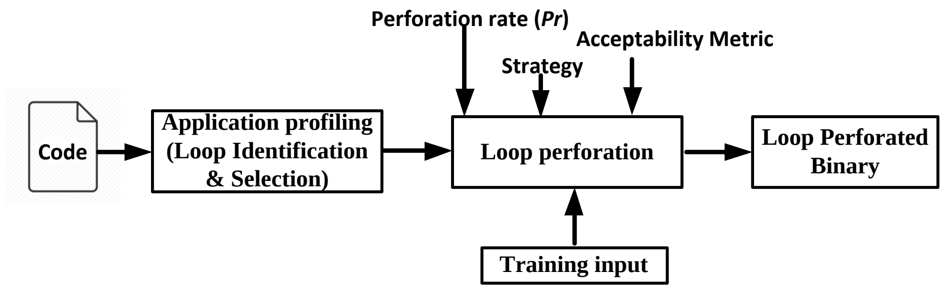


Figure 2.20: Overview of loop perforation transformation framework

Figure 2.20 shows a high level overview of a loop perforation transformation. The preprocessing phase is required to identify and select loops to perforate. The loop perforation step takes as input a loop, a percentage of iterations to skip during the execution of the loop, a perforation strategy, an acceptability metric, and training input. The percentage of non-executed iterations is called the *perforation rate (pr)*. The output is a loop perforated binary. The transformation generates a new variant of computations that produce approximate results. The performance / accuracy trade-off can be controlled by the perforation rate (pr). For example, a perforation rate $pr = 0.5$, means the iterations are skipped by half. The proposed implementation supports both static and dynamic loop perforation. The static loop perforation integrated with the compiler supports a range of perforations including modulo perforation (which skips every nth iteration), truncation perforation (skips initial or final block of iterations), interleaving perforation (transforms the loop to perform every n-th iteration) and random perforation (which skips randomly selected iterations at a mean given rate) Note that, static loop perforation does not require training input. Dynamic loop perforation perforates loops at runtime. It allows loop perforations to be turned on and off at runtime.

This work is orthogonal to ours given that our contribution is the sensitivity analysis of data and load/branch instructions. However, our analysis can help in the identification of loops that can be perforated in certain cases. For example, if a loop contains operations that modify only approximable data, then we can characterise the loop to be perforable. The above discussed work on loop perforation can deal with general loops as well.

In the following discussion, we mention about approximation techniques at the system implementation level that are used inside modern processors. The approximability analysis techniques that we have discussed earlier, either transform the program to an approximate variant, or mark approximable elements and leave it to the runtime to take appropriate actions. In addition, a number of schemes exist in literature that allow computations with approximate values. We mention some of these below.

2.4.2 Hardware and Software Techniques for Approximate Computing

In this section, we present in brief various hardware and software techniques that are of relevance to this thesis.

Load Value Approximation [57], [58]:

This work is based on the idea of load value prediction [45]. This approach minimizes execution rollback by allowing approximations. In traditional load value predictors [45], [59], [60], on a L1 cache miss, the predictor generates a value and allows the processor to execute speculatively. This allows the processor to proceed without waiting for the data to be fetched from the next level memory hierarchy. Concurrently, the load request is initiated for fetching the data from the next level of memory for validation. On arrival of the data, the prediction is validated against the actual value. If the prediction is wrong, the processor must rollback and restart executing the load instruction with the actual value. When the prediction is correct, the predictor increases its confidence for that value. If incorrect, the predictor decreases its confidence for that value. The main challenge for load value prediction is that small differences between the predicted and actual values may lead to execution rollback and hence degrade performance. In the case of floating point values, the prediction performs poorly since small variations in floating precision leads to costly rollbacks [60]. Load value approximation allows small deviations and hence often helps in reducing execution rollbacks.

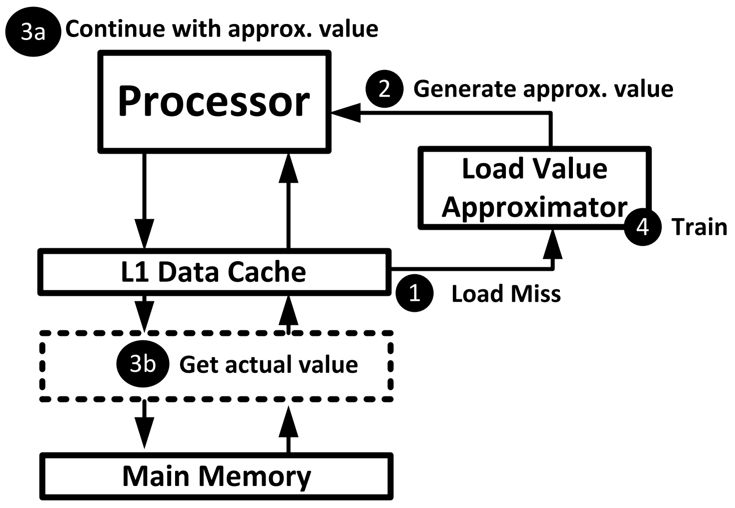


Figure 2.21: Load value approximation block diagram

Figure 2.21 shows an overview of load value approximation. When a load miss occurs in the L1 cache ❶, the load value approximator generates approximate values ❷. The processor proceeds with its execution with the approximated value 3a and concurrently sends the request to the next level of the memory hierarchy to fetch the actual data 3b. The actual data is then used to train the approximator for better accuracy 4. In conventional caches, a miss always fetches the data from the next level in the memory hierarchy. However, for load value approximation, fetching the data is optional. Fetching the data improves predictor accuracy. On the other hand, by choosing not to fetch the data, the technique can trade-off prediction accuracy for energy savings in the memory hierarchy. Also, with traditional predictors, the speculated value must be validated against the actual value. If the two are not identical, the speculative execution is rolled back and the load instruction is repeated. However, in load value approximation, the rollbacks are not required since it does not need to validate the correctness of its approximations. Load value approximation relies on the programmer to mark the approximable loads. The technique leverages on prior work by annotating data [48] and proposes an ISA extension to indicate which loads to approximate. On a range of PARSEC workloads, this technique observes an average speedup of up to 8.5% and energy savings of up to 12.6%.

A similar technique for GPUs is proposed in [58]. This technique proposes an approximation technique called Rollback-Free Value Prediction (RFVP). The main motivation is to predict the values of *safe to approximate* loads that miss in the cache, without checking for misprediction or recovery. In doing so, RFVP avoids high-cost pipeline flushes, long memory latency and at the same time is able to avoid re-executions. The technique introduces a knob called *drop rate* which is a predetermined fraction of the cache misses for which it is allowed to drop memory requests after predicting their values. The *drop rate* becomes a knob for performance and quality trade-off. By dropping the memory requests, the pressure on memory bandwidth is reduced and at the same time, the memory and cache contention is reduced. Based on the observation that in GPUs, simultaneous memory accesses require high off-chip bandwidth [61], [62], this technique applies RFVP in GPU processors to mitigate both latency and bandwidth problems. In order to inform the approximability information to the architecture, two new instructions are introduced, namely (1) an approximate load instruction and (2) an instruction for setting the drop rate. For an approximate load instruction, a bit in the opcode is set, hence permitting the micro-architecture to use RFVP. For drop rate to be visible to the micro-architecture, the ISA is extended with an instruction that sets the value of a special register to the desired drop rate.

In modern GPUs, each streaming multiprocessor (SM) contains several stream processors (SP). Each SM has its own L1 cache. The proposed technique augments each SM with an RFVP predictor. The RFVP predictor is only triggered when L1 data cache misses. In GPU

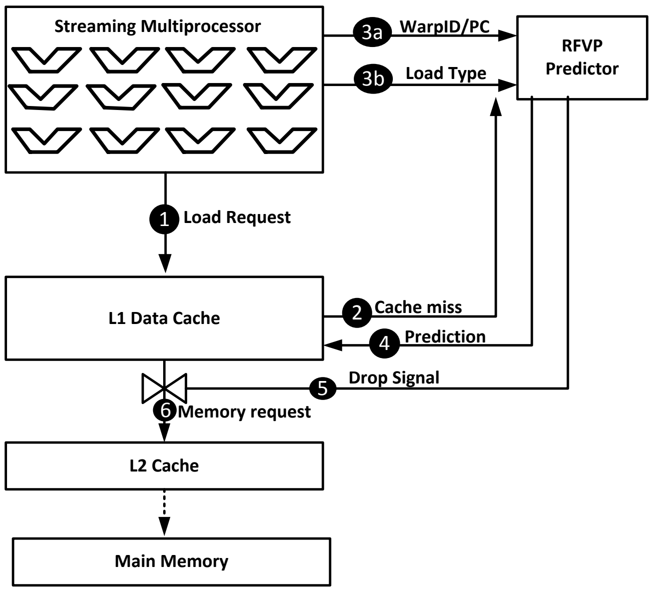


Figure 2.22: RFVP in GPU micro-architecture

SIMD [2], each SIMD load instruction accesses multiple data elements for multiple parallel threads. Figure 2.22 shows RFVP integrated in the GPU micro-architecture. In the GPU architecture, multiple threads share the same code. A group of threads sharing the same code is called a *warp*. In the figure, the SM accesses the RFVP using the WarpID and the Program Counter (PC). A load value is predicted only if its {WarpID, PC} matches the row tag in the predictor table. The predictor is of 192 entries, 4-way set associative, and consists of two delta predictors [63] and two last-value predictors [64]. On an approximable load miss in the L1 data cache, if the predictor drops the request, the RFVP predicts the entire cache line and supplies the requested words back to the SM, and also inserts the predicted line into the L1 data cache. The entire predicted lines present in the L1 data cache may be written to the memory, in this case, it might overwrite the precise content. Hence, it is required that any data accessed by a precise load must not share a cache line with the data accessed by the approximate loads. The technique relies on the compiler to allocate objects in the memory at cache line granularity and ensures that approximate data do not share a cache line with precise data.

Load value approximation relies on heuristic and programmers to annotate codes or instructions that are safe to approximate. Our proposed methods in this thesis focus on automated identification of safe to approximate loads/branches and requires little programmer intervention. Load value approximation and RFVP exploit load value approximation and perform an execution rollback if the QoS is beyond the acceptable limits. Our approach is different in the sense that safe to approximate loads identified by our analysis are guaranteed to produce an

[2]Single Instruction Multiple Data

output within acceptable limits with a user specified confidence even with load value misprediction. Also, we extend to exploit approximation in branch instructions by identifying safe to approximate branches and propose a rollback-free pipelined execution of such branches in the event of branch misprediction.

Approximate Loads in Chip Multiprocessors [65]:

This technique proposes to approximate coherence related load misses by returning stale values to the processors. The motivation of this work is based on the fact that invalidated lines in the MOESI [3] cache coherence protocol remains in the same state (not changed) for a long time despite being invalid. This allows the load miss to read those invalid lines and proceed with the execution speculatively [66]. Concurrently, the line is fetched from the memory hierarchy. On arrival of the valid lines, the validity is tested, if the lines did not match, the computation is rolled back and the load instruction is re-executed. However, invalidated lines often get evicted fairly early soon due to the cache replacement policy. To overcome this problem, they propose a Small Victim Cache (SVC) attached close to the L1 data cache to hold such invalidated lines. On a load miss, if the line is not present in the L1 cache, it is then looked up in the SVC. If it is present, it is used, otherwise, it performs the normal request to the next level memory hierarchy. With this approach, the coherence load misses could get serviced by a stale line from either the L1 Data cache or the SVC. However, such lines stay for too long in the SVC, leading to the computation getting more stale over time impacting application accuracy. To control the staleness of such lines in the SVC, they propose a time-bound called SVC + TB which sets a time threshold to limit the staleness of such lines residing in the SVC. With SVC + TB, the approach shows up to 10-15% speedup on PARSEC 2.0 workload.

In contrast to the work above, we propose techniques that connect statistical methods for approximability analysis to speculative execution in modern processors. Our approximability promise is based more firmly with the confidence of the trials that we run with the associated confidence measures. The central theme in all our chapters, as opposed to the ones in literature, is a statistical analysis technique for identifying approximable data or instructions. This gives us an unique advantage over other methods existing in literature. Our contributions for identifying approximable data and instructions automatically, as outlined in the following chapters, entail this additional guarantee. We therefore, believe that the approximability techniques proposed in this thesis have a distinct novelty over others. In the following chapters, we outline our contributions in detail.

[3]Modified Owned Exclusive Shared Invalid

Chapter 3

Automated Sensitivity Analysis of Program Data Using Dynamic Analysis[1]

Approximable programs generally consist of both *approximable* and *non-approximable* instructions and data elements. Identification of approximable code or data portions is an important step for approximate computing techniques. Approximation support at the algorithm or language level is desirable to allow a programmer explicitly mark the parts of program code or data that deem fit to be approximable. This chapter presents an automated dynamic analysis framework based on statistical hypothesis testing to automatically classify approximable program data versus the non-approximable ones with a probabilistic reliability guarantee. We present experimental results to compare our automated data classification with reported manual annotations on popular benchmarks used in approximate computing literature. We then discuss and show that our method achieves promising reliability results compared to manual annotations.

3.1 Introduction

Approximation aware programming languages have been proposed in programming languages research, where programmers can annotate data with type qualifiers (e.g. precise

[1]The contents of this chapter have been published at B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "Autosense: A framework for automated sensitivity analysis of program data", *IEEE Trans. Software Eng.*, vol. 43, no. 12, pp. 1110–1124, 2017. DOI: 10.1109/TSE.2017.2654251. [Online]. Available: https://doi.org/10.1109/TSE.2017.2654251.

(non-approximable) and approx (approximable)) to denote their reliability. However, programmers need to judiciously annotate so that the accuracy loss remains within acceptable limits. This can be non-trivial for large applications where error resilient (approximable) and non-resilient (non-approximable) program data may not be easily identifiable. Incorrect annotation of even one data element as error resilient / approximable may result in an unacceptable output. A standing challenge in approximate computing across the computing stack is to identify the error resilient components, that can in turn be realized and mapped onto the approximate hardware. At the application level, identifying program data which are error resilient against critical ones is a daunting challenge, since incorrect identification of critical data as error resilient can be catastrophic in terms of the output quality of the application. This has inspired a number of recent research articles in this direction [48], [53], [55], [68], [69]. However, existing work propose programming language frameworks for manual annotations of approximable data [48], identification of critical code segments [53], [55] or propose automated analysis of data resiliency without any quantitative reliability guarantee [69].

Our main motivation is to develop techniques that can automatically classify program data as either approximable or non-approximable with reliability guarantees. This chapter presents a dynamic analysis framework, a collection of systematic methods for program data classification with quantitative confidence guarantees. The contributions are as follows:

- We present a statistical method to classify program data as error resilient or critical based on dynamic analysis. Our method comes with a probabilistic guarantee derived from statistical tests.
- We present experimental results on benchmarks popularly used in approximate computing research to illustrate the proposed method.

Our framework can guide application designers to use programming language constructs like the one reported in [70] to effectively annotate resilient data components to gain energy efficiency without having any prior knowledge of the application domain. Moreover, the proposed technique can be tuned with a probabilistic confidence parameter and an acceptable QoS band, depending on designer requirements, to perform partitioning accordingly.

Our proposed technique has been evaluated on the Scimark benchmark [71] and three applications from the AxBench benchmark [72]. In all the evaluated applications, manual classification of error resilient data has been reported in earlier work [70]. We show that introducing approximations in all the manually classified error resilient data results in failed QoS requirement for 62.5% of the applications, for a large number of executions. On the other hand, 25% of the applications failed the same QoS requirement when approximation is introduced using our proposed method for identifying error resilient data, for the same number of executions.

3.2 Problem Overview

Before we introduce the problem formally, we reiterate on the approximate computing paradigm in the context of this work. Figure 3.1 shows a high-level overview of the approximate computing technique.

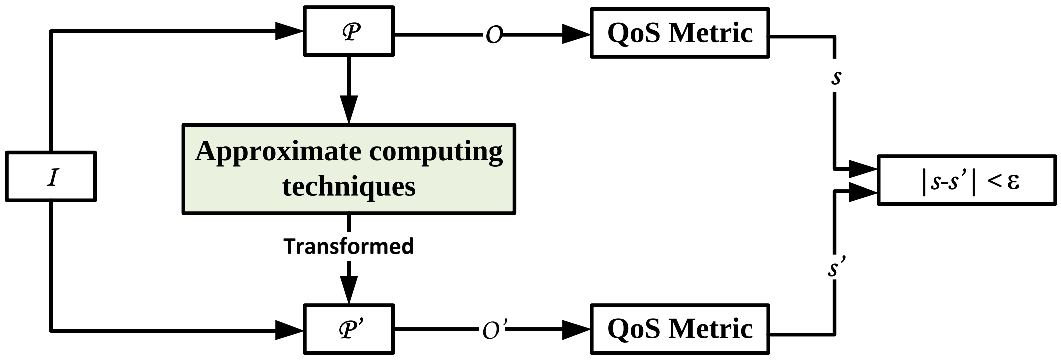


Figure 3.1: General approximate computing technique task

Consider a program $\mathcal{P}$, an input $\mathcal{I}$, the program $\mathcal{P}$ computes an output $\mathcal{O}$, i.e., $\mathcal{P}(\mathcal{I}) = \mathcal{O}$. Consider a quality of service metric $\mathcal{Q}$ which is a real-valued function that will assign a score to the output of $\mathcal{P}$, i.e., $\mathcal{Q}(\mathcal{O}) = s$. The approximate computing task is to apply approximate computing techniques to transform $\mathcal{P}$ to a new program $\mathcal{P}'$ such that $\forall \mathcal{I}.\mathcal{P}(\mathcal{I}) = \mathcal{O}$ and $\mathcal{P}'(\mathcal{I}) = \mathcal{O}'$ with $|\mathcal{Q}(\mathcal{O}) - \mathcal{Q}(\mathcal{O}')| < \epsilon$ or $|s - s'| < \epsilon$ for some user given ϵ, where $s' = \mathcal{Q}(\mathcal{O}')$ [73]. Here $|\mathcal{Q}(\mathcal{O}) - \mathcal{Q}(\mathcal{O}')|$ is the QoS band.

Definition 3.1 *An acceptable Quality of Service (QoS) band is a range of real-valued numbers* $[0, \epsilon)$ *from the QoS band, where* ϵ *is user-specified.* ■

Intuitively, techniques that transform the program $\mathcal{P}$ or its execution will most likely alter the output as compared to the output of the original program $\mathcal{P}$. The user is expected to provide a quantitative measure on how much change in the output from the original is *acceptable*. After the transformation of a program $\mathcal{P}$ to $\mathcal{P}'$, the transformed $\mathcal{P}'$ is expected to produce an output close to the output produced by $\mathcal{P}$, given the same inputs. The transformed program $\mathcal{P}'$ is expected to be more efficient in terms of time, space, energy, or some measure of performance [73]. It may be noted that not all programs are amenable to approximation. Applications such as mission-critical and real-time control are examples of applications that cannot be approximate. For certain applications, it may not be easy to define the QoS metric. For instance, for an application that processes images, we may use pixel-wise root mean square error as the metric. Also, it may not be possible to guarantee that the output of the transformed program will always be within the QoS band for all inputs. There may be *corner case* inputs that may violate the acceptable quality of service band [73]. The Quality of service metric is used to specify an application's constraints on approximation. The ideal case will be for the system to guarantee that a program's output will always be within the acceptable QoS band. In reality, it is impossible for systems to prove arbitrary quality bounds with perfect certainty. However, a realistic system can guarantee that the program's output will be within the acceptable QoS band with *high probability*. [74].

Sensitivity analysis has been applied to mathematical models of systems to understand a relation between the uncertainty in the system's output and the uncertainty in the input to the system. Questions like which are the system inputs that play a critical role in determining the variance of the system output can be answered with sensitivity analysis techniques [75]. Our focus here is to analyze the sensitivity of a program's output to its internal data rather than its input. In the following discussion, we use the terms *insensitive* and *approximable*, *sensitive* and *non-approximable* interchangeably.

Formally, let $\mathcal{I}$ and $\mathcal{R}$ denote the set of program input and output data respectively. Let $\mathcal{D}$ be the program data neither in $\mathcal{I}$ nor in $\mathcal{R}$. The objective of sensitivity analysis is to partition the set $\mathcal{D}$ into the set of sensitive data, $\mathcal{SD}$ and the set of insensitive data $\overline{\mathcal{SD}} = \mathcal{D} - \mathcal{SD}$.

The notion of sensitivity of program data with respect to a user defined Quality of Service (QoS) is formally defined as follows. Let E be the set of all possible executions of a program $\mathcal{P}$. Given an execution $e \in E$ and a program data $v \in \mathcal{D}$, let (v_e, ℓ) denote the value of v at program point ℓ in $\mathcal{P}$ during the execution e. We term this value as the *exact* value of v at location ℓ of $\mathcal{P}$ with respect to the execution e. Evidently, there could be multiple locations and therefore multiple (v_e, ℓ) values for the execution e. Let the set of program locations where v occurs in an execution e be denoted as ℓ_v^e. Let $(v_{approx}, \ell) \neq (v_e, \ell)$ denote any value of v at location ℓ. We term this as a candidate *approximate* value of v at location ℓ in $\mathcal{P}$ with respect to the execution e. The definition of *sensitivity* of v is now defined as:

Definition 3.2 *Given an acceptable QoS band for a program $\mathcal{P}$ and a sensitivity threshold probability θ, a program data $v \in \mathcal{D}$ is called sensitive if and only if $\forall e \in E$, the probability that the program output $\mathcal{R}$ remains in the acceptable QoS band when every instance (v_e, l) in e is replaced with some (v_{approx}, ℓ), is less than θ. Formally,*

$$\mathcal{SD} = \left\{ v \in \mathcal{D} \mid \forall e \in E, \forall \ell \in \ell_v^e, (v_e, \ell) \rightarrow (v_{approx}, \ell) \implies Pr(\mathcal{R} \in QoS) < \theta \right\} \tag{3.1}$$

where $(v_e, \ell) \rightarrow (v_{approx}, \ell)$ denotes the substitution of (v_{approx}, ℓ) in place of (v_e, ℓ). $\mathcal{R} \in QoS$ implies that the program output $\mathcal{R}$ is within the QoS band. The set of insensitive *data is*

```
1   bool binsearch(int lo, int hi)
2   {
3       unsigned int size = hi-lo + 1;
4       unsigned int mid = (lo+hi)/2;
5       if(lo>hi) return false;
6       if (size >= 1){
7           if(a[mid] == key)
8             return true;
9           else if(a[mid]>key)
10            return binsearch(lo, mid-1);
11          else
12            return binsearch(mid+1, hi);
13      }
14      return false;
15  }
```

Figure 3.2: Binary search procedure

defined as $\overline{\mathcal{SD}} = \mathcal{D} - \mathcal{SD}$. ■

Example 3.1 To illustrate the notion of sensitivity of program data, we present a simple example program of a binary search procedure to search for the presence of a key element in an input array, in Figure 3.2. The program has a data set $\mathcal{D} = \{lo, hi, size, mid\}$, input data set $\mathcal{I} = \{a\}$ and output data set $\mathcal{R} = \{ret\}$, where ret denotes the return value of the procedure. We consider a strict QoS here, which states that the output is acceptable only when it produces the accurate output, i.e., the procedure should return a $true$ when the search key is present in the input array and $false$ otherwise. It may be observed from the example that the data $size$ does not affect the output, the return value of the program. For any inexact value that $size$ may take other than 0, the binary search procedure will return an acceptable output. Therefore, we may conclude that $size$ is likely to be not *sensitive* to the output. Notice that the data $size$ is not *dead*[2] with respect to the program output as there exists a valuation in the data range of $size$, the value 0, such that when $size$ takes 0 as an inexact value, the program output may not produce an acceptable QoS. On the other hand, the data elements lo, hi and mid are likely to be sensitive to the program output since they are used in computing the indices of the input array a within which to search for the key. Observe that in line 4, mid depends on the data lo and hi and in line 8, mid is used as an array index. Therefore, an inexact value in any of lo, hi or mid may result in the array index mid in line 8 to be outside the allocated memory for array a. This may cause memory errors / unacceptable outputs. It is to be observed that for a multi-line code with a complex control and data flow, it is hard to classify sensitive and insensitive data by manual inspection. □

The notion of acceptable QoS depends on the application under consideration. As an example,

[2]A dead variable is one whose value has no effect on the program output and is eliminated by the compiler

the QoS of an image rendering application can be measured by the PSNR (Peak Signal to Noise Ratio) of the image. The output image of the application is deemed acceptable, given that the PSNR is less than a predefined acceptable threshold. An acceptable QoS band is user defined and it specifies the degree of precision desirable or in other words, how much approximation is acceptable to an user. In the experiments section, we define the QoS metrics considered for the benchmark applications.

Automatically identifying approximable data is a daunting task. Quantifying the relationship between program variables, inputs and outputs is difficult. Also, the test input set needs to be representative of the entire input-space. In terms of scalability, most applications consist of hundreds of program variables with different data types. Identifying program variables of different data types is a challenge and needs to be treated differently. For example, pointer types need a special analysis such as alias analysis to correctly test for sensitivity. In the following, we present our proposed methods for identification of approximable data.

3.3 Detailed Methodology

In this section, we present our dynamic sensitivity analysis method for automatic classification of sensitive and insensitive program data. The method provides probabilistic guarantees on the classification, i.e., it identifies application data which are insensitive to the QoS with a probability at least θ, where θ is user specified. Since our procedure is based on probabilistic methods, the classification may have errors. However, the probability of making an error can be bounded in the framework. To test for sensitivity of a data, our idea is to deliberately inject inexact values in the data and execute the program in the presence of inexactness. Such an execution of the program is considered as a random experiment. The outcome of the program is observed to check if it lies within the acceptable QoS band.

As an example, consider an application that renders images. We inject an inexact value in one of the application data and observe the noise in the final rendered image in the presence of inexactness to judge if it is to be rejected or accepted. In this way, every fault injected execution of an application can be interpreted as a Bernoulli trial with two possible outcomes, a *success* and a *failure*. The outcome is a *success* if the application output remains within the acceptable QoS band and is a *failure* otherwise. Example 3.2 provides a concrete example of our methodology.

Example 3.2 Consider an image processing application, a JPEG encoder for example. Figure 3.3 shows a high level overview of our methodology. Consider a hypothetical code snippet of an image processing program $\mathcal{P}$. Here, x is the target variable for sensitivity analysis. An instance of a random experiment is an experiment with an input $\mathcal{I}$, an image in this

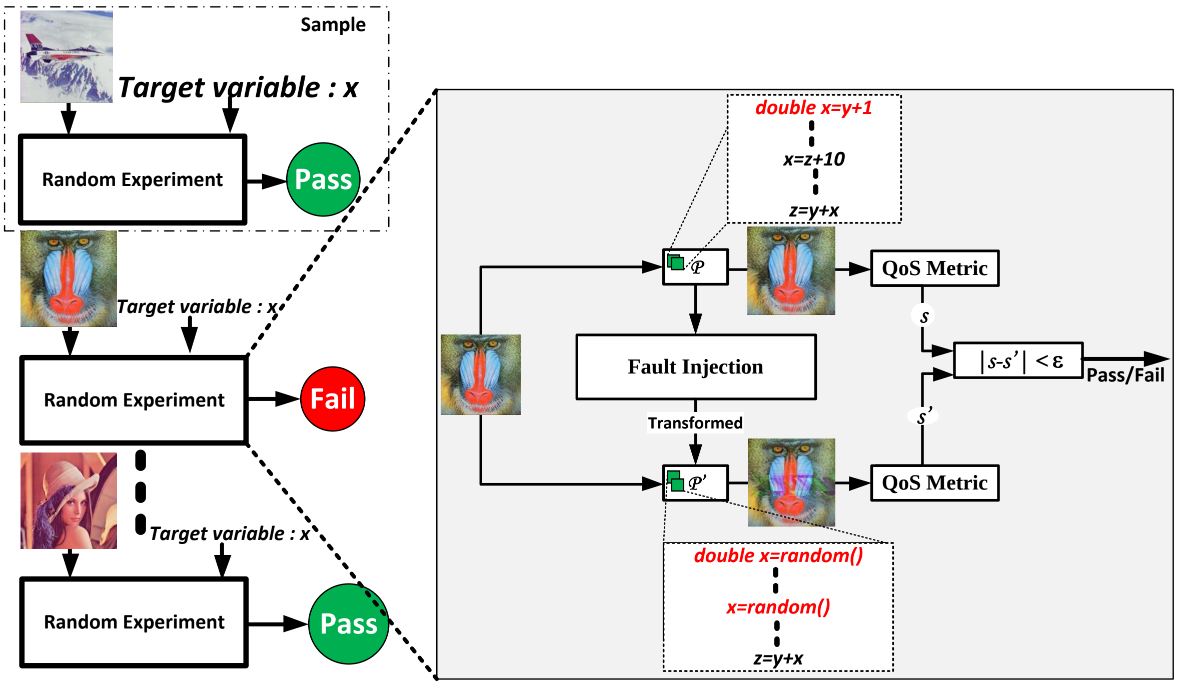


Figure 3.3: Experiment methodology of a random experiment using fault injection experiment on an image processing application

example. The experiment injects fault in the variable x by replacing the right hand side of all assignment statements of x with randomly generated values. For instance, the assignment statement $x = y + 1$ is replaced by $x = random()$ and the statement $x = z + 10$ is replaced by $x = random()$ in the transformed program. The output of $\mathcal{P}$ and $\mathcal{P}'$ is then compared using the QoS metric to decide whether the experiment is a pass or fail, based on the user defined ϵ. This random experiment is considered as a sample in our statistical analysis. We perform the analysis by running such experiments till we are confident of the sensitivity of a given target variable. $\square$

Broadly, our framework consists of the following two steps:

- Conducting Bernoulli trials by executing fault injected applications and
- Performing statistical analysis of the outcomes of the Bernoulli trials to identify sensitivity of application data.

We now discuss the fault injection mechanism and the statistical methods used in our work in the following sections.

3.3.1 Fault Injection Model

An important step in our method is fault injection, whereby we inject faults in an application execution. The purpose is to emulate an approximate computation and observe its effect on the output quality. There are standard tools for fault injection in a program. [76] presents a LLVM [77] based fault injection tool. We however, implemented our own fault injector for better control, as discussed in Section 3.4. In the fault model, we inject faults in data write operations during an execution of the program. A program data to be tested for sensitivity is selected and faults are injected at every assignment to this data during an execution. This allows us to capture the effect of faults in the chosen data under test only and therefore, the sensitivity analysis of a data is not influenced by the sensitivity of other application data. No fault is injected when a data is read. Since any program data is expected to be initialized before use, the proposed fault model presents an inexact value at every use of the data under test during execution. If the data under test has no assignments during an execution, then the fault model injects no faults, resulting in an exact computation.

3.3.2 Solution Methodology

Our work follows in the lines of [78], [79] which proposes probabilistic model checking using acceptance sampling. Acceptance sampling is used in the context of quality control of production systems where a sample of products are examined to either accept or reject the production system [3]. We use hypothesis testing [80] for acceptance sampling in our framework based on finitely many samples. In our context, a sample is an execution of the application with an input and an injected error into an application data.

Definition 3.3 *A sample is a random experiment that consists of an execution of a program $\mathcal{P}$ and execution of a fault injected program $\mathcal{P}'$. An outcome of the sample can be either a success or a failure depending on whether the fault injected program $\mathcal{P}'$ passes or fails the user-defined acceptable threshold ϵ.* ■

Definition 3.4 *A Bernoulli trial is a random experiment that can have one of two outcomes: a success or a failure [81].* ■

A fault injected program execution is interpreted as a Bernoulli trial [81] with a *success* or *failure* outcome depending on the QoS of the output. The requirement is that an experiment should complete in finite time to produce an output. Notice that we might have an infinite run of an application after an inexact value is injected into a program data. This could be possible if the injected inexact values induce an unbounded loop in the program. In our framework, we observe an execution for a reasonable time bound and deem the trial as *failed* if it does not terminate within the time bound. It can be observed that the statistical experiment of observing outcomes of n samples (trials) models a Binomial distribution, with the probability $f(x)$ of observing exactly x number of *success* given by:

$$f(x) = \binom{n}{x} p^x (1-p)^{n-x} \tag{3.2}$$

where p and $(1-p)$ are the probabilities of observing a *success* and *failure* outcome of a trial respectively. When x denotes the number of successes, $(n-x)$ denotes the number of *failures*. The mean of the distribution is $\mu = np$ and the variance is $\sigma^2 = np(1-p)$. For data sensitivity analysis using acceptance sampling, we need to choose the number of trials n and the expected number of *success* outcomes x out of the n trials to deem the data under test as tolerable to approximation. To have a probabilistic confidence on the acceptance sampling, we may decide on a minimum threshold on the probability of observing at least x *success* outcomes denoted by θ, given by:

$$\begin{aligned} f(\geq x) &= 1 - f(< x) \\ &= 1 - \sum_{k=0}^{x-1} \binom{n}{k} p^k (1-p)^{n-k} \end{aligned} \tag{3.3}$$

Note that the probability threshold θ given by Eq. 3.3 depends on n, x and p. If we consider the value of p as 0.5, we need to choose an n and x such that the probability of Eq. 3.3 is larger than the probability threshold θ. For any program data, we can then perform n fault injection trials and if the number of successful outcomes is larger than or equal to the computed x, we can classify the data as *insensitive*. However, this method does not provide any probability guarantee on the sensitivity classification since the considered minimum probability of observing at least x *success* outcomes depends on the number of samples n and the probability of a *successful* outcome p. Moreover, there is no bound on the probability of making an error, i.e., accepting a *sensitive* data erroneously as *insensitive* or vice-versa.

To address the discussed issues of acceptance sampling using the probability density function of Binomial distribution, we propose to perform acceptance sampling using hypothesis testing.

3.3.3 Acceptance Sampling using Hypothesis Testing

Hypothesis testing is a systematic way for testing a claim or hypothesis about a parameter in a population, using data measured in a sample. To test a population parameter, we test two competing hypotheses i.e., the *null* hypothesis and the *alternative / contrary* hypothesis, only one of which can be true. We illustrate these briefly below.

- *Null hypothesis* : denoted by H. The null hypothesis is the statement about the population parameter that is assumed to be true.
- *Alternative hypothesis* : H' is a statement that directly contradicts a null hypothesis.

To test a hypothesis, we need to translate a claim to a mathematical statement, for example, if the claim value is k and the population parameter is μ, then some possible pairs of null and alternative hypotheses are

$$\begin{cases} H : \mu \leq k \\ H' : \mu > k \end{cases} \quad \begin{cases} H : \mu \geq k \\ H' : \mu < k \end{cases} \quad \begin{cases} H : \mu = k \\ H' : \mu \neq k \end{cases} \tag{3.4}$$

The next step is to set the criteria for a decision which can decide whether to retain or reject the value stated in the null hypotheses. A sample is selected from the population to measure the population parameter. Finally, we compute the test statistic which produces a value that can be compared to the criterion that was set before the sample was selected.

In our context of program data sensitivity, we assign an hypothesis and a contrary hypothesis for every program data. For every $v \in \mathcal{D}$, we propose a hypothesis that $\forall e \in E, \forall \ell \in \ell_v^e, (v_e, \ell) \rightarrow (v_{approx}, \ell) \implies \mathcal{R} \in QoS$, where E, ℓ_v^e, (v_e, ℓ) and (v_{approx}, ℓ) are as defined in Definition 3.2. Let us denote such an hypothesis by K. In our analysis, we test the following null and contrary hypotheses:

$$\begin{aligned} H &: Pr(K) < \theta \\ H' &: Pr(K) \geq \theta \end{aligned} \tag{3.5}$$

where $Pr(K)$ is the probability that the hypothesis K is true.

Observation 3.1 *It may be noted that the truth of the hypothesis H implies that the program data v is* sensitive *by Definition 3.2. Similarly, the truth of the contrary hypothesis H′ implies that the data v is* insensitive. □

Definition 3.5 *A successful Bernoulli trial is an evidence of the null hypothesis whereas a failed trial is an evidence of the contrary hypothesis.* ■

The test of hypothesis can be performed using procedures for hypothesis testing [78], [82]. In this way, sensitivity analysis of application data is formulated as an instance of the classical *hypothesis testing* problem.

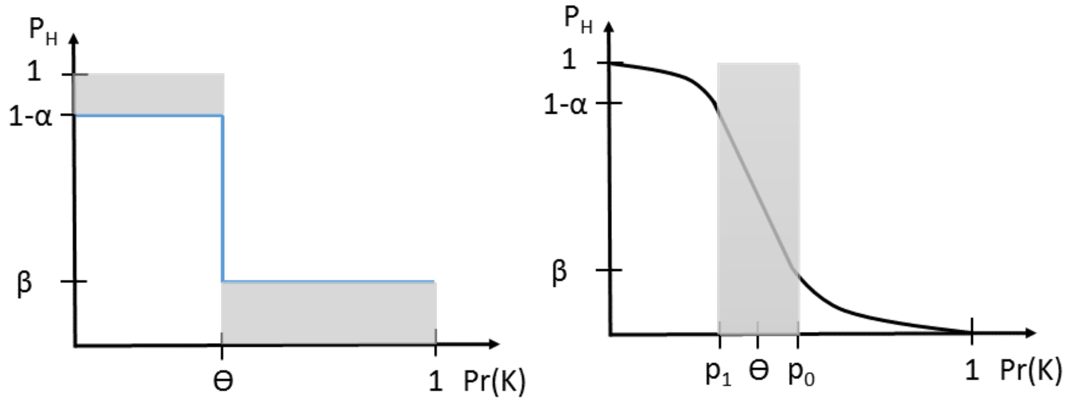
Any statistical procedure for hypothesis testing has a probability of accepting a false hypothesis. However, it is possible to have the probability of making an error reasonably low. The probability of accepting the contrary hypothesis H' when H holds is denoted as α and is called a Type I error or false negative [83]. Similarly, the probability of accepting H when H' holds is denoted by β and is called a Type II error or false positive [83]. We expect the testing procedure to have both α and β to be low (generally less than 0.5). The parameters α and β define the strength of the acceptance sampling test. The probability of accepting the hypothesis H (P_H) with acceptance sampling test of strength $\langle \alpha, \beta \rangle$ is shown in Figure 3.4a. When $Pr(K) < \theta$, the null hypothesis is accepted with a probability of at least $1 - \alpha$, shown as the grey region to the left of θ in the figure. When $Pr(K) \geq \theta$, the null hypothesis is accepted with a probability of at most β, shown as the grey region to the right of θ.

Observation 3.2 It may be observed that in an acceptance sampling test, the probability of accepting H when $Pr(K) = \theta$ should be at most β and for $Pr(K) = \theta - \epsilon$, where ϵ is infinitesimally small, the probability of accepting H should be at least $1 - \alpha$. Acceptance testing in such a case would either demand nearly exhaustive sampling of the sample space, which is infeasible for large sized sample spaces or will have $\alpha = 1 - \beta$, meaning that keeping the probability of making Type I error low makes the probability of Type II error high and vice-versa. Therefore, a test-procedure with high strength (small α and β values) with few test samples is hard to achieve. □

To overcome this problem, the use of *indifference region* has been proposed in literature [78]. Two probabilities p_0, p_1 close to θ are used such that $p_0 > p_1$ and new hypotheses $H_0 : Pr(K) \leq p_1$ and $H_1 : Pr(K) \geq p_0$ are tested instead. The null hypothesis H is accepted if H_0 is accepted and the contrary hypothesis H' is accepted if H_1 is accepted. If the probability $Pr(K)$ lies in the interval $[p_1, p_0]$, the test is indifferent to both the null and the contrary hypotheses and there is no bound on the probability of accepting a false hypothesis. This is why the probability region defined by the interval $[p_1, p_0]$ is called the indifference region. Figure 3.4b shows the typical characteristics of a realistic acceptance sampling test using indifference region [78], shown as the gray shaded area. Indifference region allows a smooth transition from accept to reject decision and as it goes narrow, we approach the ideal behavior. We now discuss the hypothesis testing procedure used in our framework.

3.3.4 Sequential Probability Ratio Test

We use sequential probability ratio test (SPRT) out of the many algorithms for hypothesis testing [4], [78]. The principle behind SPRT is to decide whether additional experiments need to be performed to accept or reject a hypothesis on the basis of the previously observed outcomes. It requires provably an optimal number of trials to test an hypothesis when



(a) Probability of Accepting the Null Hypothesis H with a Hypothetical Acceptance Sampling Test of Strength $\langle \alpha, \beta \rangle$

(b) Probability of Accepting the Hypothesis $H_0 : Pr(K) \leq p_1$ with a Typical Acceptance Sampling Test of Strength $\langle \alpha, \beta \rangle$ using Indifference Region.

Figure 3.4: Indifference Region in Acceptance Testing

$Pr(K) = p_0$ or p_1 [4]. After conducting k Bernoulli trials with outcomes $x_1, \ldots, x_k$, the procedure computes the ratio of the two probabilities as shown below:

$$\frac{p_{1k}}{p_{0k}} = \prod_{i=1}^{k} \frac{Pr[X_i = x_i | p = p_1]}{Pr[X_i = x_i | p = p_0]} = \frac{p_1^{b_k}(1-p_1)^{k-b_k}}{p_0^{b_k}(1-p_0)^{k-b_k}} \tag{3.6}$$

where X_i is a random variable associated with the i^{th} Bernoulli trial and x_i is the outcome of the trial. p_{1k} and p_{0k} denote the probabilities of observing the sequence $x_1, \ldots, x_k$ given that $Pr[X_i = 1] = p_1$ and $Pr[X_i = 1] = p_0$ respectively. $b_k = \sum_{i=1}^{k} x_i$ is the number of successful trials. The algorithm terminates by accepting the hypothesis H_0 if:

$$\frac{p_{1k}}{p_{0k}} \leq B \tag{3.7}$$

and it accepts the hypothesis H_1 if:

$$\frac{p_{1k}}{p_{0k}} \geq A \tag{3.8}$$

In practice, we choose $A = \frac{1-\beta}{\alpha}$ and $B = \frac{\beta}{1-\alpha}$ for hypothesis testing with strength $\langle \alpha, \beta \rangle$ to closely match the strength [4]. The algorithm is guaranteed to terminate, either accepting H_0 or H_1. In our framework, the test strength $\langle \alpha, \beta \rangle$ and the indifference region $[p_1, p_0]$ are user defined parameters.

Running Time

We now discuss about the running time of the method used. The running time of SPRT depends on two parameters, (1) the number of trials to decide the acceptance of the hypothesis and (2) the time taken to complete a Bernoulli trial. It is shown in [4] that the number of trials depends on the distance of the actual probability $Pr(K)$ to the indifference region. The number of trials tends to increase as $Pr(K)$ gets closer to the indifference region and gets maximum when it is equal to the center of the indifference region. The number of samples decreases as $Pr(K)$ moves away from the indifference region. The time taken to complete a Bernoulli trial depends on the application. Applications that requires a long time to complete are expected to result in reduced efficiency compared to applications that require a few seconds to complete.

3.3.5 Overall Approach

A schematic of the dynamic analysis method with the fault model is shown in Figure 3.5. In Algorithm 3.1, program data are tested for sensitivity with a hypothesis tester and partitioned as either *sensitive* or *insensitive*. The hypothesis testing procedure is shown in Algorithm 3.2. The null hypothesis is assigned on the data to be tested in line 3. The exact output of the program is computed on an input in line 4. Bernoulli trials are performed in a loop until the null hypothesis is accepted or rejected in line 10, by executing the program with the same input as in line 4 in the presence of faults in the data x, to get an approximate output. The approximate output is compared with the exact one to either accept or reject the output as per the acceptable QoS requirement in line 11. The SPRT test updates the ratio of p_{1k} to p_{0k} and compares it with A and B to decide on the acceptability of the hypothesis in lines 16 to 21.

Algorithm 3.1 Dynamic Sensitivity Analysis

```
1: function PartitionData($\mathcal{P}$, $\mathcal{D}$)
2:     $\overline{\mathcal{SD}} \leftarrow \emptyset, \mathcal{SD} \leftarrow \emptyset$
3:     Initialize $\theta$, indifference region $[p_1, p_0]$, test strength $\alpha, \beta$
4:     for all $x \in \mathcal{D}$ do
5:         $res \leftarrow$ TestSensitivity($\mathcal{P}, x, \theta, p_0, p_1, \alpha, \beta$)
6:         if $res$ then
7:             $\mathcal{SD} \leftarrow \mathcal{SD} \cup \{x\}$
8:         else
9:             $\overline{\mathcal{SD}} \leftarrow \overline{\mathcal{SD}} \cup \{x\}$
10:        end if
11:    end for
12: end function
```

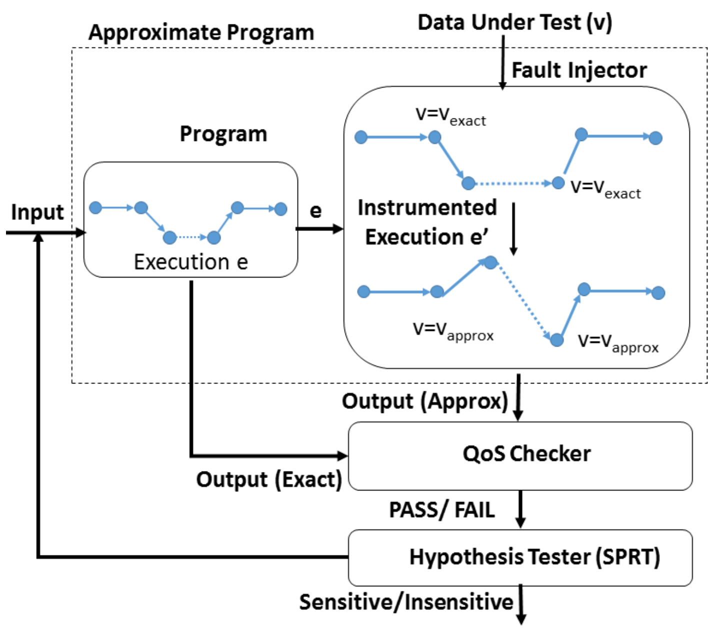


Figure 3.5: Framework of Dynamic Sensitivity Analysis with Hypothesis Testing

Algorithm 3.2 Testing Sensitivity with Hypothesis Testing

```
1: function TestSensitivity($\mathcal{P}, x, \theta, p_0, p_1, \alpha, \beta$)
2:     Assign hypothesis $K$ for data $x$
3:     Assign null hypothesis $H_0 : Pr(K) \leq p_1$
4:     $r_e \leftarrow$ execution of $\mathcal{P}$ on an input $ip$
5:     $A = \frac{1-\beta}{\alpha}, B = \frac{\beta}{1-\alpha}$
6:     $k = 0$                                        ▷ number of Bernoulli Trials
7:     $b_k = 0$                                      ▷ number of successful Bernouilli Trials
8:     $p_{1k} = p_1^{b_k}(1-p_1)^{k-b_k}, p_{0k} = p_0^{b_k}(1-p_0)^{k-b_k}$
9:     while $H_0$ not accepted/rejected do
10:        $r_a \leftarrow$ execution of $\mathcal{P}$ on input $ip$ with faults in $x$
11:        if QoS($r_a, r_e$) acceptable then
12:            $k = k + 1, b_k = b_k + 1$                ▷ Successful trial
13:        else
14:            $k = k + 1$
15:        end if
16:        $p_{1k} = p_1^{b_k}(1-p_1)^{k-b_k}, p_{0k} = p_0^{b_k}(1-p_0)^{k-b_k}$
17:        if $\frac{p_{1k}}{p_{0k}} \leq B$ then
18:            Accept $H_0$ and return true
19:        end if
20:        if $\frac{p_{1k}}{p_{0k}} \geq A$ then
21:            Reject $H_0$ and return false
22:        end if
23:    end while
24: end function
```

Example 3.3 The results of running the dynamic sensitivity analysis on the motivational

	Confidence Measure θ							
Data	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
lo	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$
hi	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$
size	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$	$\overline{\mathcal{SD}}$
mid	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$	$\mathcal{SD}$

Table 3.1: Dynamic Sensitivity Analysis on Binary Search

example of binary search in Figure 3.2 is shown in Table 3.1. The confidence θ of the analysis is the probability θ of Equation 3.5. A program data is classified as sensitive ($\mathcal{SD}$) in the table corresponding to a θ if the analysis accepts the hypothesis H of Eqn 3.5. Observe that the data *lo* and *hi* are classified *insensitive* with a confidence of θ from 0.3 to 0.5 but not for 0.6 and beyond. The data *mid* is classified as sensitive for all confidence values θ from 0.3 to 1.0, since it is an array index data. The data *size* is marked as insensitive with probability 0.9 and we do expect it to be highly insensitive. Observe that the data *size* is marked as *insensitive* with a confidence $\theta = 1$ by the analysis, however, *size* is not a dead program data as discussed previously. This is an instance when the algorithm accepts the contrary hypothesis, $H' : Pr(K) = 1$ when the null hypothesis, $H : Pr(K) < 1$ is true and therefore exhibits a Type I error. Therefore, we see that the analysis may infer sensitivity of program data wrongly due to the presence of Type I and Type II errors, however, with bounded probability. □

The dynamic analysis using SPRT is implemented keeping the probability of making Type I and Type II errors, α and β respectively, fixed to 0.01. The width of the indifference region $[p_1, p_0]$ is fixed to 2δ where $\delta = 0.01$.

3.4 Implementation

In our implementation, all the fault injection experiments on an application are conducted with a fixed input to the application. Our fault model is implemented with program binary instrumentation. We consider Java applications and perform bytecode instrumentation given the bytecode for a classfile and the application data where faults are to be injected using the Byte Code Engineering Library (BCEL) from Apache Commons [39]. At runtime, a Java Agent [84] intercepts the execution and replaces the Java bytecode for the class under inspection and injects a faulty value for the variable under test. We intend to emulate faults in memory considering random memory bit flip errors. Under such memory faults, a faulty value could be any random value in the range of the data. Therefore, to reflect memory bit flip errors in the experiments, the faulty value is generated from a uniform distribution in the interval of the range of the data under test. We consider the following kinds of program data in our implementation and instrument each as follows:

Fields: A Java class can have final/non-final, static/non-static data members. During instrumentation of a particular field variable, we set its initial value using the BCEL API to a random value obtained from a uniform distribution and remove all instructions which change its value.

Method Parameters: We set the values for the method parameters at the beginning of the method and prevent all overwrites to its values in the successive instructions.

Method Local Variables: Each local variable in a method has a different scope of existence. Two or more variables with the same name can exist in a method. Hence they must be instrumented based on their scope. We track each write instruction for the variables within their scope and replace them with a faulty value.

Method Return Value: Each method can have different points of exit i.e. they can have multiple return statements. We instrument all those return statements, for methods having non-void return type, to return the faulty value.

3.5 Evaluation

We evaluate our framework on applications [71], [72] which are known to be tolerant to approximations. Our framework provides two types of tunable parameters. The first, is the analysis confidence probability θ used in the hypothesis of Eqn. 3.5. The second, is the QoS degradation tolerance threshold, γ, which provides a measure of how much approximation could be tolerated in the application's output. The value of γ depends on the QoS metric used in the framework which can be any error metric like the mean square error, normalized error, absolute error, root mean square error etc. as discussed below. The threshold γ specifies the maximum tolerable error. Our analysis reports all loop counters and array indices as *sensitive* by default, irrespective of any specified value of the parameters. Array indices are considered as *sensitive* since an inaccurate value in the index may cause out of bound memory access. Although loop counters can be insensitive in principle, they are conservatively assumed as *sensitive* by our framework. For example, an approximate value in a counter of an empty loop will be insensitive. However, it is unlikely to have programs with empty loops. In programs with non-empty loops, approximate loop counters can potentially introduce large errors depending on the loop instructions. We discuss below the QoS metrics used in our experiments to validate our approximate data selection strategy.

3.5.1 Applications for Evaluation

Applications in the Scimark 2.0 benchmark [71] are evaluated to test for performance and accuracy of analysis on floating point intensive numerical computations. The applications in the benchmark include kernels for the Fast Fourier Transform (FFT), LU decomposition of a matrix (LU), Sparse Matrix Multiply (SMM), Successive Over Relaxation for solving system of equations (SOR) and the Monte Carlo method (MC). We elaborate on each briefly below.

- **Fast Fourier Transform (FFT)** : This kernel performs complex arithmetic, shuffling, non-constant memory references and trigonometric functions. The kernel performs a one-dimensional fourier transform of 4K complex numbers.
- **Jacobi Successive Over-relaxation (SOR)** : This kernel is for solving a system of linear equations. It uses a variant of the Gauss-Seidel method.
- **Monte Carlo Integration** : This kernel is to approximate the value of Pi (π). It computes the integral of the quarter circle by choosing random points with the unit square and computes the ratio of those within the circle.
- **Sparse Matrix Multiply** : This kernel performs matrix multiplication on an unstructured spare matrix stored in compressed-row format with a prescribed sparsity structure.
- **Dense LU Matrix Factorization** : This kernel computes the LU factorization of a dense matrix using partial pivoting.

In addition to the Scimark benchmark, experimental results on a simple Raytracer [3], a 3D image renderer, Jmeint - a triangle intersection detector in 3D and ZXing, a barcode decoder [4] application for mobile devices running on Android OS are also shown to illustrate the scalability of our method. The following discussion presents a brief detail of these.

- **Raytracer** : It is a computer graphics application that renders image by tracing the path of light as pixels in an image plane and simulates the real-word behavior of light bouncing off surfaces and colors accumulating from their paths.
- **Jmeint** : It is a triangle intersection detection application that is used in 3D gaming. The input is a pair of co-ordinates for two triangles in the 3-D space and the output is a Boolean value which indicates whether the two triangles intersect [72].
- **ZXing**: ZXing is a bar-code reader application.

[3] https://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=5590&lngWId=2
[4]https://github.com/zxing/zxing

The benchmarks have been chosen as in [70] to enable a comparison of the manual annotations reported in the paper.

3.5.2 QoS Metric

The QoS metric used in the evaluation of our framework is selected based on the application. A brief description of some of the metrics are:

Normalized Mean Error (NME): We use this metric for evaluating the QoS distortion in applications that produce single or multiple numerical results. A normalized error is given by the absolute difference between the actual and the approximate result but the difference is bounded by 1, i.e., if the absolute difference is greater than 1 then the normalized error is taken as 1. The mean of the normalized errors is denoted as NME. Formally, NME is defined as below.

$$NME(x, x') = \frac{\sum_{i=1}^{N} min(|x_i - x'_i|, 1.0)}{N} \tag{3.9}$$

x_i and $x_{i'}$ are the reference and the approximate values respectively. N is the number of outputs.

Peak Signal to Noise Ratio (*PSNR*) [85]: *PSNR* is a commonly used metric for measuring the noise introduced in an image due to a lossy compression. Given a reference image I and a noisy image K, *PSNR* is defined as:

$$PSNR = 20 \log_{10} \left(\frac{MAX_I}{\sqrt{MSE}} \right) \tag{3.10}$$

MAX_I is the maximum possible pixel value of the image and Mean Squared Error (*MSE*) is defined as:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i, j) - K(i, j)]^2 \tag{3.11}$$

$I(i, j)$ and $K(i, j)$ are the pixel values of image I and a noisy image K respectively. The size of both I and K is $m \times n$.

We use the *PSNR* metric to measure the QoS distortion introduced in image processing applications with our proposed approximation. It is measured in the logarithmic decibel scale (DB). Higher the value of *PSNR*, lesser is the noise in the image with respect to the reference image. A *PSNR* of 20-25 DB is generally acceptable.

Percent Error: We use this metric for benchmarks which produce a single numerical output. Percent error between an exact value X and an approximate value X' is given by $\frac{|X-X'|}{X} \times 100$.

Matching / Exact: Some applications are evaluated without any error tolerance in the QoS. We use this term to refer to the QoS comparison in applications where the reference output and the approximate output are compared for an exact match. If the outputs do not match exactly, then the introduced approximation in the application is not acceptable. Essentially, this way of measuring QoS acceptability does not allow any approximation in the application's results and is a strict notion of QoS acceptance. We use this metric for evaluating our methodology on applications which produce a Boolean output.

The QoS degradation tolerance threshold, denoted by γ, can be any specified value between 0 and 1, considering normalized mean error as the metric. Setting $\gamma = 0$ specifies that no error is to be tolerated in the application output. In this case, our analysis is going to identify program data which are highly insensitive (e.g. dead variables). On the contrary, setting $\gamma = 1$ specifies that any error is acceptable and therefore all data can be identified as *insensitive*.

3.5.3 Evaluation of Dynamic Sensitivity Analysis

Evaluation results of our dynamic analysis method by varying the parameters θ and γ are shown in Table 3.2 and Table 3.3. Table 3.2 shows an increase in the number of *insensitive* data identifications by relaxing the QoS requirement and keeping the confidence measure fixed at $\theta = 0.5$. Similarly, Table 3.3 shows the increase in identifications by relaxing the confidence measure and keeping the QoS degradation threshold fixed at $\gamma = 0.5$. Since there is a probability of accepting a false hypothesis in our analysis (though very small), the number of identified *insensitive* data may vary over runs of our analysis. Therefore, the reported results are taken as the median over three runs. We have verified that there is negligible change in the reported percentage of *insensitive* data derived over multiple runs of our framework. We observe that for the benchmarks LU, MC and SMM, the percentage of *insensitive* program data does not change by varying the parameters. This implies that the insensitive data of these applications are highly *insensitive* and the other data are highly *sensitive*.

Table 3.4 shows the number of trials that the dynamic analysis performed for different confidence values over some program data in the benchmark applications. The experiment uses the QoS tolerance parameter γ fixed at 0.5. A plot of the data is shown in Figure 3.6. For the hypothesis K defined in Eqn. 3.5 on a data under test, let $Pr(K) = p$. It is known that the number of trials in SPRT depends on the distance of the parameter θ from p [78]. The number

QoS	Percent Data Derived Insensitive				
γ	FFT	SOR	MC	SMM	LU
0.010	0	0	33	15	11
0.025	0	0	33	15	11
0.050	0	0	33	15	11
0.075	0	0	33	15	11
0.1	0	0	33	15	11
0.2	0	13	33	15	11
0.3	0	27	33	15	11
0.4	7	27	33	15	11
0.5	7	27	33	15	11

Table 3.2: Percentage insensitive data reported by our analysis on varying γ and fixed $\theta = 0.5$

Conf.	Percent Data Derived Insensitive						
θ	FFT	SOR	MC	SMM	LU	Raytracer	Jmeint
0.3	10	33	33	15	11	48	24
0.4	10	33	33	15	11	48	24
0.5	7	27	33	15	11	44	24
0.6	7	20	33	15	11	44	24
0.7	7	20	33	15	11	44	24
0.8	7	20	33	15	11	44	24
0.9	7	20	33	15	11	44	24

Table 3.3: Percentage insensitive data reported by our analysis on varying θ and fixed QoS $\gamma = 0.5$ (Scimark2), PSNR=10.5 (Raytracer) and Exact (Jmeint)

of trials tends to increase as θ gets closer to p and it decreases as θ gets farther away from p. Therefore, the plot in Figure 3.6 can give us an idea of p for the data under test. For example, the number of trials for the considered data in FFT and SOR is maximum when $\theta = 0.5$. We can therefore deduce that $Pr(K) \approx 0.5$. Similarly, we can deduce that $Pr(K) \approx 0.8$ for the considered data in the application Jmeint. For the applications MC, SMM, LU and Ray-tracer, the number of trials decreases on increasing θ from 0.3 to 0.9, which implies that $Pr(K) \leq 0.3$ for the considered data in these applications. In the application Zxing, the number of trials increases on increasing θ from 0.3 to 0.9, which implies $Pr(K) \geq 0.9$ for the considered data.

Note that in SOR, the dynamic analysis classified the data as *insensitive* for $\theta = 0.5$ and as *sensitive* for $\theta = 0.6$. This means that the hypothesis $H : Pr(K) < 0.5$ has failed (the contrary hypothesis $H' : Pr(K) \geq 0.5$ has passed) and the hypothesis $H : Pr(K) < 0.6$ has passed. This observation indicates that $0.5 \leq Pr(K) < 0.6$. Similarly in FFT, we get that $0.6 \leq Pr(K) < 0.7$. However, due to Type I and Type II errors in SPRT, $Pr(K)$ may not always be in the same interval. In general, the confidence that $\theta_1 \leq Pr(K) < \theta_2$ is given by the probability $(1 - \alpha)(1 - \beta)$, when the contrary hypothesis $H' : Pr(K) \geq \theta_1$ and the null

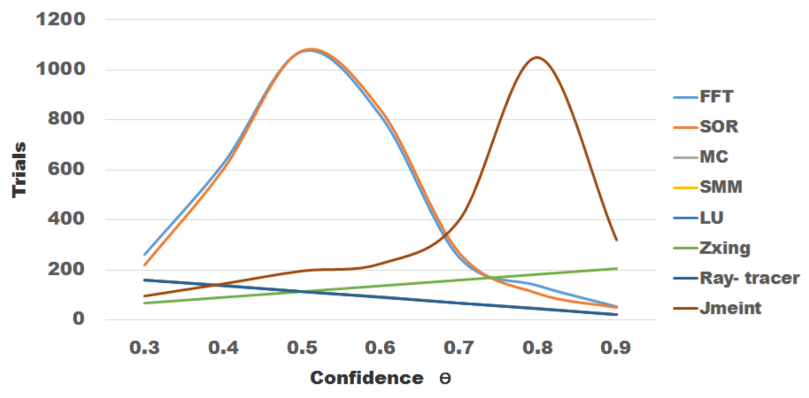


Figure 3.6: Number of Trials vs. Confidence θ in SPRT

hypothesis $H : Pr(K) < \theta_2$ passes the test. Recall that α and β are the probability of making a Type I and Type II error in the test respectively. Since the confidence values of $Pr(K) \geq \theta_1$ and $Pr(K) < \theta_2$ are $(1-\alpha)$ and $(1-\beta)$ respectively, the confidence of $\theta_1 \leq Pr(K) < \theta_2$ is $(1-\alpha)(1-\beta)$.

Application	Data	Number of Trials in SPRT						
		$\theta = 0.3$	$\theta = 0.4$	$\theta = 0.5$	$\theta = 0.6$	$\theta = 0.7$	$\theta = 0.8$	$\theta = 0.9$
FFT	n	261 (I)	626 (I)	1075 (I)	817 (I)	249 (S)	136 (S)	52 (S)
SOR	N	219 (I)	601 (I)	1075 (I)	839 (S)	267 (S)	105 (S)	49 (S)
MC	x	161 (S)	138 (S)	115 (S)	92 (S)	69 (S)	46 (S)	23 (S)
SMM	row	161 (S)	138 (S)	115 (S)	92 (S)	69 (S)	46 (S)	23 (S)
LU	Aii	161 (S)	138 (S)	115 (S)	92 (S)	69 (S)	46 (S)	23 (S)
Zxing	row	69 (I)	92 (I)	115 (I)	138 (I)	161 (I)	184 (I)	207 (I)
Ray-tracer	xe	161 (S)	138 (S)	115 (S)	92 (S)	69 (S)	46 (S)	23 (S)
Jmeint	xx	97 (I)	146 (I)	197 (I)	226 (I)	401 (I)	1050 (I)	321 (S)

Table 3.4: Number of trials in dynamic analysis of a single data by varying confidence

3.5.4 Experimental comparison with other methods

Our proposed dynamic analysis is most similar to ASAC [69], which also proposes a method for automated sensitivity analysis of program data using a statistical method. We implemented the ASAC algorithm with the fault injection model presented in Section 3.3.1. In order to obtain an interval to select a random instrumented value for fault injection on a data, [69] proposes performing a static range analysis. In our implementation, we do not perform a range analysis since our goal is to emulate memory bit flip errors which can result in any erroneous data in the datatype range and therefore, selecting fault values from an obtained range by static analysis is not justified. Table 3.5 reports the percent data identified as *insensitive* by ASAC as compared to our analysis for the Scimark 2.0 benchmark. The QoS metric used for the applications is *normalized mean error*, with QoS tolerance $\gamma = 0.5$. Our analysis derivations are with a confidence of $\theta = 0.5$. The algorithm in ASAC has two tunable parameters to qualitatively improve the confidence of the derivation. One of the parameters is denoted as the *discretisation constant* k and the other is denoted as the number of samples m. The derivation confidence increases with larger values of k and m. We choose $k = 30$ and $m = 100$ for our experiments which can be considered giving derivations with medium to high confidence. We observe that the performance of our analysis is better than ASAC mostly because it requires fewer executions. We also observe that if there is a highly sensitive data in the set of data tested by ASAC, all the data are derived as sensitive. This is because, the experiments in ASAC are performed by introducing perturbations simultaneously in all the data and high sensitivity of a single data in the set results in all outcomes failing the QoS test. This is a limitation of ASAC. For example, 0% data is derived as *insensitive* in the application MC because we include a data *under_curve* in the set of data to be tested by ASAC which is highly sensitive. A similar phenomenon is also seen in the application FFT.

			ASAC		Our Analysis	
Application	LOC	%Data Tested	%I	Time (Sec)	%I	Time (Sec)
MC	22	100	0	44	33	1
SMM	29	38	22	76	40	18
SOR	36	60	33	92	27	27
FFT	119	62	0	163	7	22
LU	165	35	9	155	11	41

Table 3.5: Performance comparison of ASAC and our analysis

3.5.5 Reliability Evaluation

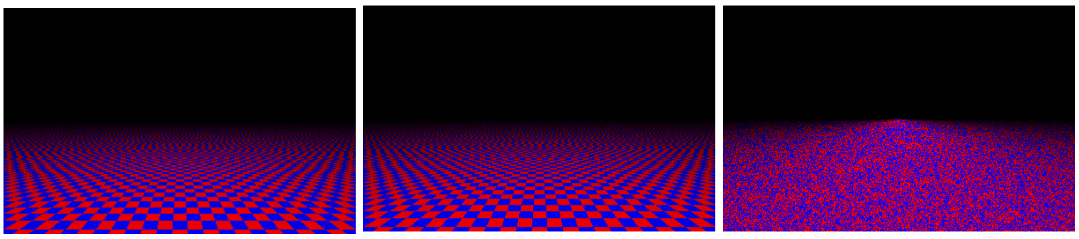
To evaluate the reliability of our analysis, we compare our sensitivity classification with manual annotations in the applications reported in [70] in which approximable data are annotated with the *@approx* keyword. The authors of [70] mention that an approximate value in their manual annotations guarantees that the application does not crash and keeps a balance between reliability and energy saving. We observe the output of 1000 executions of the applications in the benchmark by injecting errors in all the manually annotated data and report the percentage of outputs that failed the QoS degradation threshold. Similarly, we perform 1000 executions of the benchmark applications by injecting errors in *insensitive* data derived by us with confidence $\theta = 0.3$ and $\theta = 0.5$ respectively and report the percentage of outputs failing the same QoS degradation threshold. The QoS threshold is fixed to $\gamma = 0.5$ for the QoS metric of normalized error, PSNR ≥ 10.5 for the QoS metric of PSNR.

For two applications, there is no tolerance to QoS degradation. A comparison of the percentage of output failures is reported in Table 3.6. Column 5 (#Manual Annot.) shows the number of data in the application that is annotated as *approximable* in [70], while Column 6 (%Fail Manual) shows the percentage of output failures with inexactness in the manually annotated approximable data. Column 7 (%Data our analysis Tested) shows the percentage of data that is tested by our analysis with hypothesis testing. The eighth and eleventh columns (%I) show the percentage of data in the application that is classified as *insensitive* by our analysis with confidence $\theta = 0.3$ and $\theta = 0.5$ respectively. We see that 100% of the executions failed the QoS requirement for the applications MC, Ray-tracer, LU and Zxing when there is any inexact value in the manually annotated data. On the other hand, 100% of the executions passed the same QoS requirement when there is an inexact value in the data derived as *insensitive* by our analysis with confidence $\theta = 0.5$ in MC, Ray-tracer and LU. However, 10% of the executions failed the QoS in Zxing. Considering that our analysis performed the sensitivity classification with confidence probability of $\theta = 0.5$, the number of executions that may fail the QoS should be less than 50% of the executions with inexact value in the *insensitive* data. In both the applications Zxing and Jmeint, the percentage of executions that failed with inexactness in data derived as insensitive by us, is less than 50% (10% and 29% respectively). We also see that SOR, FFT, Jmeint and Zxing failed in 49%, 77%, 29% and 10% of the executions with inexact value in *insensitive* data derived with confidence $\theta = 0.3$. This shows the reduced reliability of the derivations with reduced confidence parameter value of θ. Note that a confidence $\theta = 0.3$ allows at most 70% failure of executions with inexactness in the derived *insensitive* data. The observed failure percentage is less than 70% except in FFT (77%). We believe that this is because of the classification of some *sensitive* data as *insensitive* due to Type I or Type II errors. The percentage of *insensitive* data in a program gives a measure of its error resiliency. For example, we can deduce that the application *Ray-tracer* is the most error resilient of the applications tested with our analysis since it has 44% of *insensitive* data with confidence $\theta = 0.5$.

The reliability of our analysis is further illustrated on the application Ray-tracer, a 3D image renderer. Figure 3.7b shows the rendered image when inaccuracies are injected in all the data identified as *insensitive* by our analysis with a confidence of $\theta = 0.5$. Figure 3.7c shows the rendered image when inaccuracy is injected in a manually annotated approximable data which our analysis classified as *sensitive*. The perceived noise in Figure 3.7c is much more in comparison to Figure 3.7b. This shows that inaccuracy in even one sensitive data may cause unacceptable program output.

							$\theta = 0.3$			$\theta = 0.5$		
Application	QoS Metric	QoS (γ)	LOC	#Manual Annot.	%Fail Manual	%Data Our analysis Tested	%I	Time (Sec)	%Fail	%I	Time (Sec)	%Fail
MC	Normalized Mean Error	0.5	22	2	100	100	33	1	0	33	1	0
SMM	Normalized Mean Error	0.5	29	4	0	54	15	18	0	15	18	0
SOR	Normalized Mean Error	0.5	36	3	0	60	33	27	49	27	27	0
RayTracer	PSNR	≥ 10.5	111	11	100	55	48	221	0	44	613	0
FFT	Normalized Mean Error	0.5	119	24	0	62	10	22	77	7	22	0
LU	Normalized Mean Error	0.5	165	22	100	36	11	41	0	11	41	0
Jmeint	Pass if correct, Fail otherwise	Exact	2694	109	28	50	24	947	29	24	1150	29
ZXing	Pass if correct, Fail otherwise	Exact	15785	115	100	80	35	1188	10	35	1435	10

Table 3.6: Percentage of output failing QoS with confidence $\theta = 0.3$ and $\theta = 0.5$



(a) Original Image (b) Rendered image with our analysis guided approximation (c) Rendered image with manually annotated data

Figure 3.7: Illustrating QoS reliability in Raytracer rendered image with our analysis guided approximation

3.5.6 Summary

Statistical methods are good tools for sensitivity analysis in mathematical and computational models. For example, the two-sample Kolmogorov-Smirnov test (K-S test) is a statistical method for testing the hypothesis that two given cumulative distribution functions (CDF) are identical. The maximum distance between the CDFs (D-statistic) is computed and the test accepts the hypothesis if this distance is less than a critical threshold. Such a statistical procedure for hypothesis testing can be used for sensitivity analysis of a model. A standard way of sensitivity analysis using K-S test is to run Monte-Carlo simulations of the model on different values of the input factors as statistical experiments and partition the simulation output as either an *acceptable* or *unacceptable* event, based on some criteria. The criteria of acceptance of a model behavior or simulation output could be any metric showing the deviation of the observed behavior from the expected behavior. The CDF of the acceptable and unacceptable events are passed to the K-S test. A rejected hypothesis indicates a considerable distance between the two CDFs, indicating sensitivity of the model to the input factors. Similarly, there are other sensitivity analysis techniques based on statistical methods such as variance and correlation based analysis, Bayesian uncertainty estimation etc. There are sensitivity analysis techniques not using statistical methods too. A simple such analysis is by computing the partial derivative of a model output with respect to an input factor. These remain to be explored in future.

The dynamic analysis step requires a number of program executions to complete hypothesis testing. There can be applications where the executions are sensitive to the input. In such applications, the executions used for testing should be selected with an input sampling strategy in order to satisfy some coverage criteria such as all branch or all statement coverage. This might require the use of automated test case generators to intelligently sample the inputs for testing. However, for applications for which automated test case generators cannot generate inputs, a corpus of inputs for the dynamic analysis engine is to be created manually and this can be challenging. Another limitation of dynamic analysis is that there can be code and data covered by complex conditionals which are triggered in very few specific executions. Such data are likely to be marked as insensitive by the analysis but they can be critical. For example, if there is data used in triggering an error handler inside a complex conditional branch, then such a critical data gets classified as insensitive. Perhaps global sensitivity analysis (GSA) techniques might be useful in such applications instead of regional sensitivity analysis (RSA) and this remains a subject of future research.

For each program data, the dynamic analysis requires substantial number of program executions to complete the hypothesis. In term of scalability, for applications that require long time to complete, dynamic analysis will fail to scale. This motivates us to look for alternative methods to improve the sensitivity analysis step. In the following chapter, we present a static-dynamic combined sensitivity analysis scheme to address some of these issues by performing dynamic analysis on a subset of initial program data and then using this information to statically infer the sensitivity of the remaining program data.

Chapter 4

A combined static-dynamic method for sensitivity analysis of program data[1]

4.1 Introduction

For programs having a large number of data elements, the dynamic analysis method discussed in the previous chapter can be slow since an hypothesis for each data needs to be tested. Thus, compute and data intensive programs may take a long time to terminate, making each trial during the hypothesis testing expensive, thereby slowing down the overall analysis. To address these performance issues, this chapter presents a hybrid static-dynamic combined sensitivity analysis approach. We first present an overview of our proposed technique and then demonstrate the efficiency of our approach on some approximate computing benchmarks.

The hybrid analysis approach starts with an initial set of data sensitivity analysis outcomes obtained from the dynamic analysis step discussed in Chapter 3. Starting with this set, it uses dataflow information together with data sensitivity rules that basically govern the flow of information between program data, to further enrich the outcomes. The analysis then statically classifies the program data as either *sensitive* (non-approximable) or *insensitive* (approximable).

[1]The contents of this chapter have been published at B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "Autosense: A framework for automated sensitivity analysis of program data", *IEEE Trans. Software Eng.*, vol. 43, no. 12, pp. 1110–1124, 2017. DOI: `10.1109/TSE.2017.2654251`. [Online]. Available: `https://doi.org/10.1109/TSE.2017.2654251`.

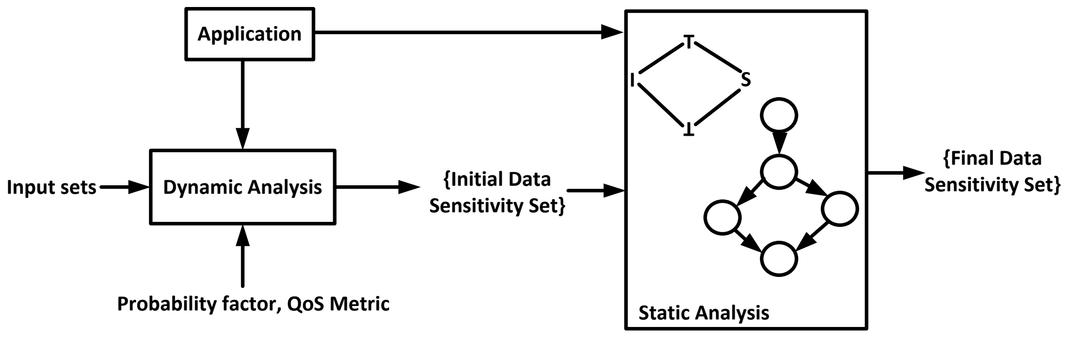

Application
Input sets
Dynamic Analysis
{Initial Data Sensitivity Set}
Probability factor, QoS Metric
T
I
S
⊥
Static Analysis
{Final Data Sensitivity Set}

Figure 4.1: High level overview of our hybrid static-dynamic analysis

Figure 4.1 shows a high level overview of our hybrid analysis approach. It consists of two steps as described below.

- **Dynamic analysis:** A subset of program data is chosen as the initial set for sensitivity analysis. The analysis then classifies each program data in the initial set as either *sensitive* or *insensitive* using the hypothesis testing procedure discussed in Chapter 3.
- **Static analysis:** The input to this is the target application and the sensitivity set obtained from dynamic analysis. The analysis then classifies the remaining program data based on the information from the initial sensitivity set, with a static dataflow analysis.

Our contribution in this chapter is a hybrid analysis method for approximability analysis, together with experimental results on public domain benchmarks.

In the following sections, we provide elaborate discussions on our approach.

4.2 Detailed Methodology

In this section, we present our approach in detail. We begin with an illustration of the static analysis step, followed by an illustration of the overall framework.

4.2.1 Static Analysis for Program Data Sensitivity

We now discuss our contribution of inferring data sensitivity using static analysis, building on the foundations of data flow analysis and lattice theory, presented in Chapter 2. As discussed earlier, the idea behind data flow analysis [25] is to abstract the program structure as a control flow graph and define data flow equations specifying the value of the analysis of interest at the entry and exit points of the program statements. The entry point of a program statement denotes the program point just before the statement and the exit point denotes the program point immediately following that statement. The equations are solved for a *may* or *must* analysis solution at the program points [25]. In Chapter 2, we have discussed that a *may* analysis solution at a program point is satisfied by at least one execution path. The *must* analysis solution at any program point is satisfied by every execution path. There is an underlying framework in solving the data flow equations, called the *Monotone framework* [25]. Any data flow analysis which is formulated as a monotone framework can be solved with a generic maximum fixed point algorithm (MFP) [25]. A monotone framework consists of two components, a complete lattice L over the property space satisfying the ascending chain condition and a set $\mathcal{F} : L \rightarrow L$ of monotone functions closed under composition and containing the identity function. Monotonicity and the ascending chain property of the lattice guarantee termination of the MFP algorithm on lattices of finite height. The property space is the domain of values of interest to any program analysis, e.g., intervals for interval analysis, set of variables for live variable analysis etc. The set $\mathcal{F}$ contains the transfer functions of the analysis. A transfer function $f_\ell \in \mathcal{F}$ specifies how the program statement at a program point ℓ can transform the value of the analysis.

Data Sensitivity Lattice

We present an approach for static analysis of program data sensitivity as an instance of the monotone framework. An instance of a monotone framework consists of the following:

- A complete lattice, L, of the framework
- The space of functions, $\mathcal{F}$, of the framework.
- A finite control flow graph of the program P, $flow(P)$.
- A set of initial labels, E, containing the labels of statements of program P which are its entry points.
- An initial value of the analysis, $init \in L$, for labels $\in E$.
- A mapping, $f : Labels \rightarrow \mathcal{F}$, which maps each program statement label to a transfer function in $\mathcal{F}$.

where a label of a statement in a program is a unique natural number assigned to the statement. $Labels$ is the set of all such labels of a program. The elements of the complete lattice L of our analysis are mappings $\sigma : \mathcal{D} \rightarrow \{\bot, S, I, \top\}$. $\sigma(x) = \bot$ denotes that no information is known about the data x whereas $\sigma(x) = \top$ denotes that x may be *sensitive* or *insensitive*. $\sigma(x) = S$ and $\sigma(x) = I$ denote x to be *sensitive* and *insensitive* respectively. Borrowing from the work on language based information flow methods to ensure security properties in [86], we define a *data sensitivity lattice* over the range of σ, i.e., $\{\bot, S, I, \top\}$ as shown in Figure 4.2.

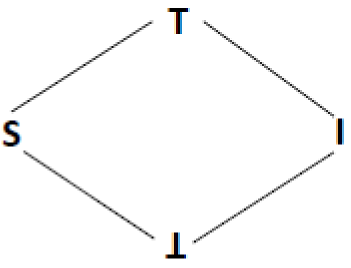


Figure 4.2: Data Sensitivity Lattice

Definition 4.1 *The partial order* $\sqsubseteq$ *on* σ *is defined as follows:*

$$\begin{aligned} &\forall\sigma : \bot \sqsubseteq \sigma \\ &\forall\sigma_1, \sigma_2 : \sigma_1 \sqsubseteq \sigma_2 \textit{ iff } \forall x, \sigma_1(x) \sqsubseteq_D \sigma_2(x). \end{aligned} \tag{4.1}$$

where $\bot \in \sigma$ *maps every* $x \in \mathcal{D}$ *to* $\bot$, $\sqsubseteq_D$ *denotes the partial order relation of the* data sensitivity lattice. ■

Definition 4.2 *The* join *operation over* σ *is defined as:*

$$(\sigma_1 \sqcup \sigma_2)(x) = \sigma_1(x) \sqcup \sigma_2(x). \tag{4.2}$$

■

Transfer Functions

The transfer functions in our analysis are based on the assumption that approximate or *insensitive* data do not flow into *sensitive* data. A similar approach is adopted in the EnerJ programming model [48] where programmers have the provision to annotate datatypes with @approx or @precise keywords but the type-checker does not allow assignment of approximate data into precise ones. Considering such a flow restriction on approximate programs, we define the transfer functions of our assignment statements. The transfer function for all other types of statements are considered to be an identity function. Considering a general assignment statement block $[x := a]$, a being any expression, we define the transfer functions of our analysis as:

$$
\begin{aligned}
[x = a] : f(\sigma) &= \begin{cases} \sigma(x \rightarrow I) & \textit{if } \forall v \in FV(a), \sigma(v) = I \\ \sigma(x \rightarrow S) & \textit{if } \forall v \in FV(a), \sigma(v) = S \\ \sigma(x \rightarrow \top) & \textit{if } \exists u, v \in FV(a) \\ & \;\textit{s.t. } \sigma(u) = S, \sigma(v) = I \\ \sigma & \textit{if } FV(a) = \emptyset \end{cases} \\
[\cdots] : f(\sigma) &= \sigma
\end{aligned}
\tag{4.3}
$$

where $[\cdots]$ denotes any program statement which is not an assignment statement and $FV(a)$ is the set of all free variables of the expression a. Essentially, we classify the data in the lhs of the assignment statement as *insensitive* if the data in the rhs of the assignment statement are classified as *insensitive*. In this case, since *insensitive* data is flowing into the lhs data x, we classify it to be *insensitive* as well. Similarly, we classify the data in the lhs as *sensitive* if the data elements in the rhs of the assignment statement are already classified to be *sensitive*. This expresses the condition that *sensitive* data should flow into *sensitive* data only. When there are both *sensitive* and *insensitive* data in the rhs, we are *inconclusive* about the sensitivity information of the lhs data x, and we assign it to the $\top$ element of the *data sensitivity lattice*. In the last case, when there is no free variable in the rhs (i.e, a is an expression with constants), we keep the sensitivity mapping of the argument.

We now present the definition of our static sensitivity analysis:

Definition 4.3 *Sensitivity Analysis ($\mathcal{SA}$) is an instance of a monotone framework consisting of:*

(i) *The complete lattice $L = (\sigma, \sqsubseteq, \sqcup)$ such that $\sigma : \mathcal{D} \rightarrow \{\perp, S, I, \top\}$ and $\sqsubseteq$, $\sqcup$ are as defined in Eq. 4.1 and Eq. 4.2 respectively.*

(ii) *The set of monotone functions $\mathcal{F} = \{f : \sigma \rightarrow \sigma\}$*

(iii) *A finite control flow graph of the program, $flow(P)$.*

(iv) *A set of initial labels of the program, E.*

(v) *An initial value of the analysis, $\sigma_{init} \in L$, for each label in E.*

(vi) *A mapping, $f : Labels \rightarrow \mathcal{F}$, which maps the labels ℓ of the assignment statements of P to the functions in $\mathcal{F}$ as defined in Eq. 4.3 and maps all other statement labels to the identity function.*

σ_{init} is a given initial data sensitivity mapping. If no sensitivity information is known initially, then σ_{init} is $\perp$. Note that with $\sigma_{init} = \perp$, our proposed analysis is expected to produce a solution of $\sigma = \sigma_{init} = \perp$, since it propagates the initial known sensitivity information by the data flow relations. Therefore, the effectiveness of the proposed method depends largely on σ_{init}. ■

In this context, Section 4.2.2 discusses our approach of using the previously proposed dynamic analysis methods to initialize σ_{init}. While solving our static analysis, we do not update the sensitivity mapping of program data that is already mapped to *sensitive*(S) or *insensitive*(I) by σ_{init}.

It is easy to see that L satisfies the ascending chain condition since the number of elements in L is finite. $\mathcal{F}$ contains the identity function and is closed under composition. An instance of the analysis can be solved using the maximum fixed point algorithm (MFP) [25] using Eq. 4.4.

$$
\begin{aligned}
\mathcal{SA}_{entry}(\ell) &= \begin{cases} \bigsqcup \mathcal{SA}_{exit}(\ell') \mid (\ell', \ell) \in flow(P) & \textit{if } \ell \notin E \\ \perp & \textit{if } \ell \in E \end{cases} \\
\mathcal{SA}_{exit}(\ell) &= f_\ell(\mathcal{SA}_{entry}(\ell)), \ \textit{where } f_\ell = f(\ell).
\end{aligned} \tag{4.4}
$$

$\mathcal{SA}_{entry}(\ell)$ and $\mathcal{SA}_{exit}(\ell)$ denote the solution of the analysis at the entry and exit points of the statement labeled ℓ. After obtaining a solution of the analysis, we get the set of sensitive data $\mathcal{D}$ using Eq. 4.5.

$$
\mathcal{SD} = \bigsqcup \{\mathcal{SA}_{exit}(\ell) \mid \ell \in final\} \tag{4.5}
$$

where $final$ is the set of labels of the exit points of a program. Eq. 4.5 states that the data sensitivity is the join of the sensitivity derived at all the exit points of the program.

4.2.2 Combining Static and Dynamic Analysis

As discussed, the effectiveness of the static analysis depends on the initial known sensitivity information in σ_{init}. We propose to obtain σ_{init} by running our dynamic analysis presented in Chapter 3 Section 3.3 on some of the data from $\mathcal{D}$. In our dynamic analysis, we perform dynamic analysis on the following types of data, (1) global data, (2) method parameters and (3) method local data whose expression has constant(s) or function call(s). We then apply static analysis method-wise. We perform the sensitivity of method parameters and global data dynamically as method local data are likely to be data dependent on its method parameters and global data. Local variables which are initialized to constants are tested dynamically because the static analysis would not be able to identify the sensitivity of such variables. Method variables for which the assignment expression has a function call are also tested dynamically because their sensitivity depends on the sensitivity of the return values of the functions which might not be known at the time of solving the analysis statically.

The MFP algorithm is an iterative procedure that processes the edges (ℓ, ℓ') of the flow graph $flow(P)$, until no further edges are left for processing. The edges to be processed are collected in a data structure called $worklist$. Initially, all the edges of $flow(P)$ are inserted in the $worklist$. Another data-structure, $Analysis$, stores the current analysis solution at the entry of a statement labeled at i at $Analysis(i)$. Initially, $Analysis(i)$ contains the initial value of the analysis. In an iteration, an edge (ℓ, ℓ') is removed from the $worklist$ and the transfer function $f_\ell = f(\ell)$ is applied on $Analysis(\ell)$. If $f_\ell(Analysis(\ell))$ is not $\sqsubseteq$ $Analysis(\ell')$ then $Analysis(\ell')$ is updated to $f_\ell(Analysis(\ell)) \bigsqcup Analysis(\ell')$ and all the edges (ℓ', ℓ'') in $flow(P)$ are inserted into the $worklist$. The intuition behind inserting the edges (ℓ', ℓ'') into the $worklist$ is that since $Analysis(\ell')$ is updated, the $Analysis(\ell'')$ needs to be recomputed. The iterations continue until the $worklist$ is empty indicating that the fixed point solution has been reached for the analysis at every program statement.

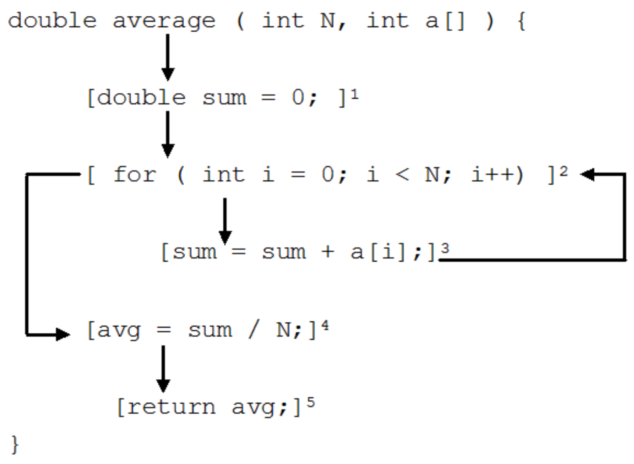


Figure 4.3: Control Flow Graph of an Average Routine

Example 4.1 We illustrate the proposed static-dynamic hybrid sensitivity analysis over an averaging routine for N numbers. The control flow graph of the program is shown in Figure 4.3. The superscripts of the program statements denote the labels associated with the statements for ease of illustration. The sensitivity of the program data N, a and sum are derived using dynamic analysis as per the initialization rules discussed in Section 4.2.2. Let the derived sensitivity of N, a and sum be *insensitive* (I) by the dynamic analysis. The initial value of the analysis, σ_{init}, therefore assigns N, a and sum to I and the remaining data avg to $\perp$. The worklist initially contains all the edges $W = \{(1,2),(2,3),(2,4),(3,2),(4,5)\}$. The MFP iterations and the analysis value at every entry point is shown in Table 4.1. The table

		Analysis value at entry points				
Iters	Worklist	1	2	3	4	5
1	{(1,2),..., (4,5)}	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$
2	{(2,3),..., (4,5)}	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$
3	{(2,4),..., (4,5)}	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$
4	{(3,2),(4,5)}	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$
5	{(4,5)}	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$
6	Empty	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow \bot$	$N \rightarrow I$, $a \rightarrow S$, $sum \rightarrow I$, $avg \rightarrow I$

Table 4.1: MFP Iterations for $\mathcal{SA}$ of an Averaging Routine

entries show the value of the analysis at the entry points of the program statements, i.e, the mapping σ from the program data to the elements of the *data sensitivity lattice*. In the first iteration, the edge $(1, 2)$ is removed from the $worklist$ and the transfer function corresponding to the assignment statement labeled by 1 ($sum = 0$) is applied. Since the rhs is a constant, there is no free variable and the sensitivity mapping remains unchanged at $Analysis(2)$. In the second iteration, the edge $(2, 3)$ is removed from the $worklist$ and the transfer function corresponding to the for statement, the identity function, is applied on $Analysis(2)$ resulting in no change. Similarly, edge $(2, 4)$ is removed from the $worklist$ in the third iteration and results in no change of values in $Analysis$. In the fourth iteration, the edge (3, 2) is removed and also results in no change of values in $Analysis$ since the lhs of the assignment statement labeled by 3 is *sum* which is already mapped as *insensitive* (I) by σ_{init} and we do not update mappings by σ_{init}. In the fifth iteration, the last remaining edge in the $worklist$, $(4, 5)$ is removed and the transfer function corresponding to the assignment statement labeled by 4 is applied on $Analysis(4)$. The free variables of the rhs are N and *sum* which are both mapped as *insensitive*(I) in $Analysis(4)$. Therefore, the transfer function $f(4)$ maps the lhs data avg

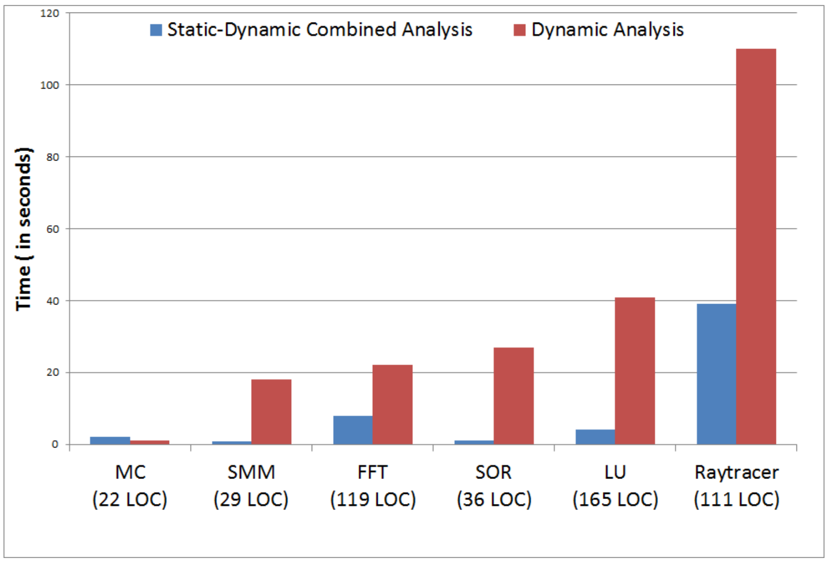


Figure 4.4: Performance of Static-Dynamic Combined Analysis in Comparison to Dynamic Analysis

as $insensitive(I)$. The updated mapping σ is not $\sqsubseteq$ to the old $Analysis(5)$ since $\sigma(avg) = I$ is not $\sqsubseteq$ to the sensitivity of avg in $Analysis(5)$ which is $\perp$. Thus, $Analysis(5)$ is updated to $Analysis(5) \sqcup \sigma = \sigma$. This modified mapping is shown in $Analysis(5)$ in iteration 6 of the table. At this stage, the $worklist$ is empty and the algorithm terminates. Overall, we see that the analysis value remains the same as σ_{init} in all the iterations except the last. □

4.3 Evaluation

In this section, we show the performance gain with our proposed static-dynamic combined

Application	TP	FP	FN	Precision (%)	Recall (%)
FFT	0	0	3	0	0
SOR	3	0	0	100	100
MC	1	0	1	100	50
SMM	2	0	0	100	100
LU	0	0	9	0	0
Raytracer	0	1	2	0	0

Table 4.2: Precision and Recall of the Static-Dynamic Combined Analysis in Comparison to Dynamic Analysis

sensitivity analysis as discussed in Section 4.2.2. Figure 4.4 shows the performance improvement on an average over multiple runs. Our analysis is implemented to analyze Java programs and it performs the analysis class-wise and method-wise.The analysis of Raytracer is most expensive with 110 seconds since it involves an expensive image rendering algorithm. The dynamic analysis in combination with the static sensitivity analysis of the Raytracer program takes 39 secs, though it fails to identify many *insensitive* data. Table 4.2 shows that the gain in performance with the combined analysis is at the cost of precision and recall in some cases. The precision and recall are compared against the sensitivity derivations using the dynamic analysis only method with hypothesis testing, as described in the previous chapter. In the table, TP, FP and FN denote the number of *true positives*, *false positives* and *false negatives* respectively. A *true positive* means that the data classified as *insensitive* by our sensitivity analysis ($\mathcal{SA}$) is also classified as *insensitive* by the dynamic analysis. A *false positive* means that the data classified as *insensitive* by $\mathcal{SA}$ is classified *sensitive* by dynamic analysis and a *false negative* means that the data classified as *sensitive* by the combined analysis is classified as *insensitive* by dynamic analysis. The precision is computed as $\frac{TP}{(TP+FP)} \times 100$ and the recall is computed as $\frac{TP}{(TP+FN)} \times 100$. Observe that the static-dynamic combined analysis displays 100% precision and 100%, 50% and 100% recall for SOR, MC and SMM respectively and completes much faster than the dynamic analysis alone. Note that the poor precision is not because of high *false positives* but because of low *true positives*. In essence, the combined analysis is efficient but misses to identify many *insensitive* data of a program. The precision of the combined analysis is expected to increase when the combination uses more of dynamic analysis and the performance is expected to increase when the combination uses more of static analysis.

Reliability Analysis: Table 4.2 shows the precision and recall of the hybrid analysis technique. It may be noted that there is only 1 *false positive*, for the application Raytracer. Therefore, the sensitivity classification of the combined analysis is mostly in accordance with that of the the monolithic dynamic analysis. Since, we have already tested the reliability of our dynamic analysis (presented in Section 3.5.5), a separate reliability testing for the static-dynamic combined analysis is not carried-out.

4.4 Summary

Identifying insensitive error resilient data of an application is non-trivial, especially when the application is large and has substantial data and control dependencies. Manual annotation of data may not be reliable and may result in unacceptable output even when one data is mis-annotated as insensitive / resilient in approximation aware programming languages like EnerJ. This chapter and the previous one present our contributions in this direction. In this chapter, to address the limitations of a dynamic analysis only approach, a combination of static and dynamic sensitivity analysis is proposed that is efficient, especially for applications running compute intensive algorithms, but results in many false negatives compared to the dynamic analysis. However, 100% precision with the combined analysis could be achieved for some of the benchmarks and this is quite encouraging for our further analysis.

Once the data and instructions that are approximable are identified, the next step that we propose is to make use of this information in an execution environment to derive maximum benefits. While there exists proposals in literature to use the sensitivity analysis output to modify appropriate program segments or parts of the generated code, for faster execution or energy savings, without compromising program output beyond the accepted threshold, we take a different route altogether in this thesis. The main highlight of our approach is to couple this approximability information with the execution runtime, thereby exploiting this advantage to sometimes go ahead with approximate / partially incorrect outcomes without the need for execution rollbacks. In the following chapters, we present our proposals of using the outcomes of our approximability analysis inside processor runtimes during execution for more performance and energy benefits. In the next chapter, we discuss techniques to aid speculative execution in modern processors by embracing approximate computing. In particular, we propose to selectively relax the performance penalty of load-value mis-predictions and branch mis-predictions by avoiding execution roll-backs in a pipelined execution for data and instructions that are found approximable by our analysis methods proposed in this chapter. In the following chapter, we extend the same philosophy to a concurrent execution environment for a multi-processor shared cache setup. The way we couple approximate computing with speculation and runtime execution through the processor runtime is quite novel and distinctly different from approaches existing in literature, and has been seldom looked at as well in the program analysis or approximate computing research community, to the best of our knowledge.

Chapter 5

Improving Runtime Efficiency of Programs using Approximate Computing[1]

In this chapter, we propose approximate computing techniques that are intrinsic to the architecture on which the application executes. We adapt the sensitivity analysis methods discussed in the previous chapters to perform approximability analysis of instructions, particularly load and branch instructions. Using the sensitivity knowledge of these instructions, we propose a mechanism for improving the performance and energy usage of a program running on modern processors. Modern processors use speculative execution extensively for performance enhancements. However, on a mis-speculation, a performance penalty is associated as an execution roll-back. In the approximate computing paradigm, instructions can be marked as either *approximable* or *non-approximable*. In this work, we perform the sensitivity analysis of load and branch instructions and propose a speculative execution with a no roll-back policy for approximable instructions. The sensitivity analysis that is presented in the previous chapters is able to infer the sensitivity of a program to faults in an individual program data or instruction. This chapter further proposes an analysis technique based on Bayesian analysis that addresses the sensitivity of a program to faults in a set of data and instructions together.

[1]The contents of this chapter have been published at B. Nongpoh, R. Ray, M. Das, and A. Banerjee, "Enhancing speculative execution with selective approximate computing", *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 2, 26:1–26:29, Feb. 2019, ISSN: 1084-4309. DOI: `10.1145/3307651`. [Online]. Available: `http://doi.acm.org/10.1145/3307651`.

5.1 Introduction

Speculative execution is an optimization technique used in modern processors by which predicted instructions are executed in advance with an objective of overlapping the latencies of slow operations. *Branch prediction* and *load value speculation* are examples of speculative execution used in modern pipelined processors to avoid execution stalls discussed in Chapter 2, Section 2.3. However, speculative execution incurs a performance penalty as an execution roll-back, when there is a misprediction. In this chapter, we propose to aid speculative execution with approximate computing by relaxing the execution roll-back penalty associated with a misprediction. Building on the foundation of the previous chapters, we present a sensitivity analysis method for data and branches in a program in order to identify the ones which can be executed without any roll-back in the pipeline and yet can ensure a certain user-specified quality of service of the application with a probabilistic reliability. Our analysis is based on statistical methods similar to the one proposed in Chapter 3 and 4. We also present a study of the cumulative effect of approximation in program branches using Bayesian analysis and program data using the multiple hypothesis testing method. We then perform an architectural simulation of our proposed approximate execution method and report the benefits in terms of CPU cycles and energy utilization on selected applications from the AxBench, Accept, and Parsec 3.0 benchmarks suite.

Processors today employ deep instruction pipelines [88] in order to implement instruction level parallelism. However, branch and load instructions pose challenges in a pipelined execution. Branch instructions introduce a stall since the next instruction to be executed in the pipeline is known only after the evaluation of the branch condition. Load instructions introduce a stall due to access time latency of the primary memory (also known as the memory wall [89]). *Speculative execution* is an optimization technique used in modern processors to mitigate a pipeline stall by executing the likely next instructions beforehand [90]–[96]. *Branch prediction* [94]–[96] and *load value speculation* [92], [93] are examples of speculative execution. *Branch predictors* speculate the next instruction to be pipelined in the fetch stage of a pipeline to avoid delay, while a *load value predictor* speculates a load value and allows the processor to continue execution with the speculated value when there is a miss in the data cache. However, speculative execution incurs a performance penalty due to pipeline flushing and re-loading, when there is a misprediction of the branch or the load value for a data. In this work, we propose to embrace approximate computing to relax the penalty associated with speculative execution. The main highlight of our proposal is to automatically identify the misprediction tolerant branches and data in a program. We term a data / branch as misprediction tolerant given that in the event of a data value / branch misprediction during speculative execution, the usual process of execution roll-back can be relaxed on these selected load / branch instructions and yet, it can be asserted with a certain probability that the application will produce an acceptable Quality of Service (QoS). We propose to execute these instructions on the speculated path to completion (in case of a branch) and with the speculated data value (in case of a load), with a no rollback policy, even if they encounter a misprediction.

The application specific QoS metrics are also natural and commonly used in the approximate computing literature. In our proposed technique of enhancing speculative execution, the same common QoS metrics as in the previous chapters are considered to evaluate our selective rollback speculative execution strategy. The intrinsic tolerance towards approximate computing that these applications exhibit, as acknowledged in already published literature, is what we leverage on to make speculative execution more effective for these. No additional user-defined metric is needed to adopt our proposal, the already defined and standardized metrics in the respective application domains, that have been considered as good candidates for approximate computing, can simply be used to compare the quality of results and the performance benefits achieved.

Our contributions in this chapter are as follows:

- We formalize the notion of mis-prediction tolerant data / branches in a program, and setup a bridge with our concepts presented in the earlier chapters.
- A dynamic analysis driven automated method for identification of mis-prediction tolerant data and branches in a program is proposed. Our analysis is based on a statistical method [4], [79] that can be applied on a program, to derive with a probabilistic certainty that the user-defined acceptable QoS will be met even when misprediction penalty is relaxed in one of the tolerant data / branches in the program.
- We propose approximate ISA extensions of a processor for load and branch instructions to execute the exact and the approximate version of the instruction. The sensitivity inference can be passed on to a compiler to transform the program by replacing the relevant load and branch instructions by their approximable counterparts, such as $\langle load.approx \rangle$ and $\langle br.approx \rangle$.
- We show an application of the proposed sensitivity analysis in enhancing speculative execution of pipelined processors by selectively allowing rollback-free execution in the event of a mis-prediction. A pipelined execution continues on the speculated execution path till completion for fault tolerant data loads and branches identified above, however, works as usual for the intolerant ones.
- A detailed hardware implementation of the proposed selective rollback-free pipelined execution is presented.

5.2 Methodology

The main kernel of our proposal is a sensitivity analysis step for determining which load and branch instructions are approximable. The sensitivity analysis phase identifies the misprediction tolerant instructions in a program. It is performed once per application per QoS definition off-line. We denote this as *pre-execution analysis*, which involves the following three main steps:

- First, we find a candidate set of data loads and branches in a program to perform our sensitivity analysis. We select the data in the code regions with heavy data-cache miss rate. Similarly, we select the branches in the code regions with high branch misprediction rate. Such a selection is done with the expectation of having considerable energy and performance benefits with our misprediction penalty free execution. Cache-miss and branch misprediction heavy code regions are identified using profiling on representative inputs.
- Second, we systematically analyze the sensitivity of the program's QoS with respect to faults in the selected load and branch instructions in the program. The program's QoS is measured in terms of the user-specified QoS metric for the application. If the program's QoS stays within the user-specified QoS degradation tolerance, in the presence of faults in a data or a branch direction, we consider that a load-value or a branch misprediction penalty in the pipelined processor can be safely relaxed for such data or branch instructions.
- Third, the program is transformed by replacing the misprediction tolerant load and branch instructions with their corresponding approximable counterparts. This transformed program is an approximate version of the original one which can be executed in a pipelined processor which relaxes the penalty of load-value and branch misprediction of approximable loads and branches selectively.

We perform an architectural simulation of our scheme to present the corresponding benefit in terms of reduced energy and CPU cycles in comparison to an exact execution, on the architectural simulator *Sniper* [97].

An evaluation of our sensitivity analysis on 9 approximate computing benchmark applications shows a maximum of 36% of the tested data and 59% of the tested branches to be approximable. Architectural simulation of our proposed rollback-free execution on the approximable loads and branches show a maximum of 72% reduction in CPU cycles and energy consumption on the *Bodytrack* application from *PARSEC 3.0* [98]. On an average, we report

23% reduction in CPU cycles and energy consumption on the architectural simulation of the nine benchmark applications.

5.3 Motivating Example

In this section, we present an overview of our work using the *x264* and *Sobel* applications from the ACCEPT benchmark [99], a popular benchmark suite used in approximate computing research. We will first discuss a motivating example for a load, followed by an example for a branch instruction.

5.3.1 Load Instruction

A code snippet of *x264*, an application for encoding video streams into *H.264/MPEG-4 AVC* format is shown in Fig. 5.1. A cache profiling of the code shows 4.78% average L1 data cache miss for a test input set. The annotations in the snippet are around statements where the profiler detects a substantial cache miss. There are 14 data load instructions within the annotated snippet of code shown in the figure. These load instructions are considered as candidates for sensitivity analysis. We systematically analyze the sensitivity of the application's QoS with respect to faults in the load data of the candidate instructions using techniques similar to the ones discussed in the previous chapter. For *x264* and *Sobel* in particular, we consider Structural Similarity Index (SSIM) as the metric in order to measure the QoS degradation in the presence of approximations in data and branches. SSIM is a metric used to measure the noise in an image with respect to a reference image [100]. Since *x264* and *Sobel* produce images, SSIM is a natural choice for QoS evaluation. The sensitivity analysis also needs an acceptable threshold on the QoS degradation of an application in the presence of approximations. As discussed earlier, we term an application data as *approximable* if we know that an execution of the application with a faulty / inexact value in the data is not going to distort the application's QoS beyond the acceptable threshold of QoS degradation. We formalize the concept of approximability in the subsequent discussion.

Line No	Inst. Addr.	Assembly Code	D1 Miss (in %)
9	**0x43cf97**	**movzbl (%r14,%rbx,2),%eax**	**3.21**
9	**0x43cfa2**	**movzbl 0x2(%r14,%rbx,2),%eax**	**1.15**
9	0x43cf77	movzbl (%r15,%rbx,2),%edi	0.68

Table 5.1: A subset of load instructions from the annotated segment in *x264* and the corresponding D1 cache miss percentage on a test input.

```
1  static void mc_chroma(..){
2   ...
3  for( int y = 0; y < i_height; y++ )
4  {
5  for( int x = 0; x < i_width; x++ )
6   {
7    //@cache miss heavy
8
9   dstu[x] = (cA*src[2*x]+cB*src[2*x+2]+cC*srcp[2*x]+cD*srcp[2*x
      +2]+32)>>6;
10   dstv[x] = ( cA*src[2*x+1]+cB*src[2*x+3]+cC*srcp[2*x+1]+cD*srcp[2*
      x+3]+32)>>6;
11   // @end
12  }
13 }
14 ...
15 }
```

Figure 5.1: *Code snippet of x264*

Our analysis based on statistical methods identifies 9 out of the 14 load instructions as *approximable*, i.e., fault tolerant. Quantitatively, it means that faulty data values in one of these 9 load instructions will not distort the QoS of *x264* beyond a threshold of 0.8 Structural Similarity Index, with a probability of at least 0.5. The acceptable threshold of QoS degradation (SSIM of $\geq$ 0.8 in this example) and the confidence of inference (probability $\geq$ 0.5) are user tunable in our method.

Table 5.1 shows the first 3 instructions from the annotated segment ranked by the D1 cache miss rate. In the table, the 1^{st} column shows the line number in the source code, the 2^{nd} column shows the instruction address in hexadecimal, the 3^{rd} column shows the corresponding assembly instruction and the 4^{th} column shows the percentage of D1 cache miss in the instruction. Instructions with addresses ***0x43cf97*** and ***0x43cfa2*** are classified as approximable whereas the instruction *0x43cf77* is classified as non-approximable by our analysis. ***0x43cf97*** and ***0x43cfa2*** correspond to statements ***srcp[2*x]*** and ***srcp[2*x+2]*** in line 8 respectively. Note that these approximable load instructions correspond to loading elements of the data *srcp* and not the loading of data *x*. Using sensitivity analysis, we deduce that any approximate load value of ***srcp*** can be used in the statements ***srcp[2*x]*** and ***srcp[2*x+2]*** in the event of a cache miss on their load data and yet have an assurance that the application QoS will not be distorted beyond the user-defined threshold, with a user-defined confidence. We use this knowledge to allow an execution in the pipeline to continue with a mispredicted load value, in the event of a cache miss on the load data. As a result, the delay associated with fetching the data from primary memory to cache memory can be avoided. However, we perform the usual process of pipeline flushing and reloading, when a load value misprediction is detected on data *src* in *src[2*x]* in line 8, since it corresponds to the non-approximable load instruction *0x43cf77*.

5.3.2 Branch Instruction

We now provide an overview of our work on program branches using a code fragment from the *Sobel* application [101] from the ACCEPT benchmark suite [99], shown in Fig. 5.2. This fragment of the code contains branches with considerable branch mispredictions, in other words, branches which are hard to predict, on a test input set. Table 5.2 shows the 3 branch instructions from the code snippet ranked by percentage of branch misprediction reported from a profiler on a test input.

```
 1 // sobel.c
 2 void sobel_filtering( ){
 3  ...
 4  pixel_value = 0.0;
 5  for (j = -1; j <= 1; j++){
 6   for (i = -1; i <= 1; i++) {
 7     pixel_value += weight[j + 1][i + 1] * (image1[y + j][x + i]);
 8   }
 9  }
10  ...
11 for (j = -1; j <= 1; j++) {
12   for (i = -1; i <= 1; i++) {
13    pixel_value_app += weight[j + 1][i + 1] *  image1[ya + j][xa + i
      ];
14    }
15   }
16  pixel_value_app = MAX_BRIGHTNESS * (pixel_value_app - min) / (max
     - min);
17  image2[ya][xa] = (unsigned char)pixel_value_app;
18  ...
19 }
```

Figure 5.2: *Code snippet of Sobel*

Line No	Inst. Addr.	Assembly Code	Mispredictions (in %)
12	0x40158d	jne 401557	37.4
5	**0x40159c**	**jne 401540**	**12.5**
11	0x401714	jne 4016b8	12.4

Table 5.2: Branch instructions and the corresponding percentage branch mispredictions in *Sobel* on a test input

The 1st column shows the corresponding line number in the source code, the 2nd column shows the instruction address in hexadecimal, the 3rd column shows the corresponding assembly instruction and the 4th column shows the percentage of branch mispredictions contributed by the instruction. For example, the instructions with addresses *0x401714* and *0x40158d* in the table correspond to the branches on condition *(*$j \leq 1$*)* and *(*$i \leq 1$*)* in lines 11 and 12 respectively in the source code. We systematically analyze the sensitivity of the application's QoS with respect to faults in the direction of these branch instructions. The analysis is performed by defining an SSIM of (≥ 0.8) as the acceptable degradation in QoS. A fault in a branch direction is the resulting execution of the application when the branch direction is alternated. Roughly speaking, we term a branch as *approximable* if we know that an execution of the application with a faulty direction for the branch is not going to distort the application's QoS beyond the acceptable threshold of QoS degradation. In the Sobel example, we observe that the branch having the address ***0x40159c*** corresponding to ***(***$j \leq 1$***)*** in line 5 in the source is classified as approximable by our analysis. Essentially, the program behavior is not affected beyond the allowable QoS distortion threshold regardless of the branch being taken or not taken in the *for* loop, and therefore renders it to be misprediction tolerant. Note that this branching code is not discarded by the compiler even with the highest optimization level -O3 in *GCC*, since the compiler cannot statically deduce that one out of the two sides of the branch is dead. This shows that there can be fault tolerant branches in a program which compiler optimization techniques cannot detect and discard. Having the sensitivity knowledge about the branches in an application, a branch misprediction on the tolerant branches can be ignored to continue along an incorrect program path, saving energy and improving performance as a result of no execution rollback.

5.4 Overall Approach

In brief, our sensitivity analysis is based on a perturb and observe strategy which is similar to the one described in Chapter 3. Faults are injected in the load value/branch direction of the load/branch instruction under test and the resulting effect on the application's QoS is observed. Each such fault injection experiment is interpreted as a Binomial trial [81] as earlier with either a *success* or a *failure* outcome, given that the resulting QoS distortion of the application remains acceptable or not respectively. After performing the necessary number of Binomial trials, the sensitivity of a load / branch instruction is determined. For example, our sensitivity analysis performs 115 fault injection trials on the load instruction with address *0x43cf97* in *x264*, and finds the result of all to be *success*, in order to infer that the instruction is *approximable* with a confidence probability of at least 0.5. The number of trials necessary to make an inference with a certain probability is governed by the statistical inference algorithm as in the earlier chapters. An architectural simulation of misprediction penalty free execution by continuing with any speculative data value on these approximable loads in *x264* shows nearly 2% reduction in CPU cycles and energy consumption. Similarly, our method of sensitivity analysis performs 151 fault injection experiments on the branch instruction with address *0x40158d* in *Sobel*, and finds 18 with an outcome of *success* and 133 with an outcome of *failure* to infer that the instruction is *non-approximable*. An architectural simulation of roll-back free execution on the event of a branch misprediction on the approximable branches in *Sobel* shows nearly 23% reduction in CPU cycles and energy consumption.

5.5 Sensitivity Analysis by Hypothesis Testing

The sensitivity analysis method is similar to the one proposed in Chapter 3. However, the fault models are adapted to loads and branches. We consider two types of fault experiments, one on program data and the other on program branches respectively. We define the two types of fault injection experiments on a given program $\mathcal{P}$ as follows:

Data-Fault Injection Experiment: Let $\mathcal{D}$ be the set of program data. Let E be the set of all possible executions of a program $\mathcal{P}$. Given an execution $e \in E$ and a program data $v \in \mathcal{D}$, let $\langle v_e, \ell \rangle$ denote the value of v at program point ℓ in $\mathcal{P}$ during the execution e. We term this value as the *exact* value of v at location ℓ of $\mathcal{P}$ with respect to the execution e. Evidently, there could be multiple locations and therefore multiple $\langle v_e, \ell \rangle$ values for the execution e. Let the set of program locations where v occurs in an execution e be denoted as ℓ_v^e. Let $\langle v_{approx}, \ell \rangle \neq \langle v_e, \ell \rangle$ denote any value of v at location $\ell \in \ell_v^e$. We term this as a candidate *approximate* value of v at location ℓ in $\mathcal{P}$ with respect to the execution e. Here, v_{approx} models a mispredicted load / store value at a load / store instruction at location ℓ. A data-fault injection experiment of $\mathcal{P}$ is the resulting execution when $\langle v_e, \ell \rangle$ is replaced with a candidate approximate value $\langle v_{approx}, \ell \rangle$.

Example 5.1 We consider a JPEG Encoder application 3.2 to describe our methodology. Figure 5.3 shows an overview of our fault injection experiment. In the code snippet shown in the figure, the arrow pointed line ($z = y + x$) is the statement of our interest for sensitivity analysis, labeled as the target load. The statement is broken down into ISA instructions, i.e., load x into register $r1$, load y into register $r2$, perform an arithmetic add operation and finally store the result in the variable z. Let us consider a single load instruction for analysis. In this example, we consider the instruction that loads x into register $r1$ (load $r1$, x). Our fault model injects an erroneous data into register $r1$. For instance, if the precise value of $r1$ after executing the load instruction should have been 60 (say) for some input, the fault model injects a random value say 1452 into the register $r1$. The results of the exact execution

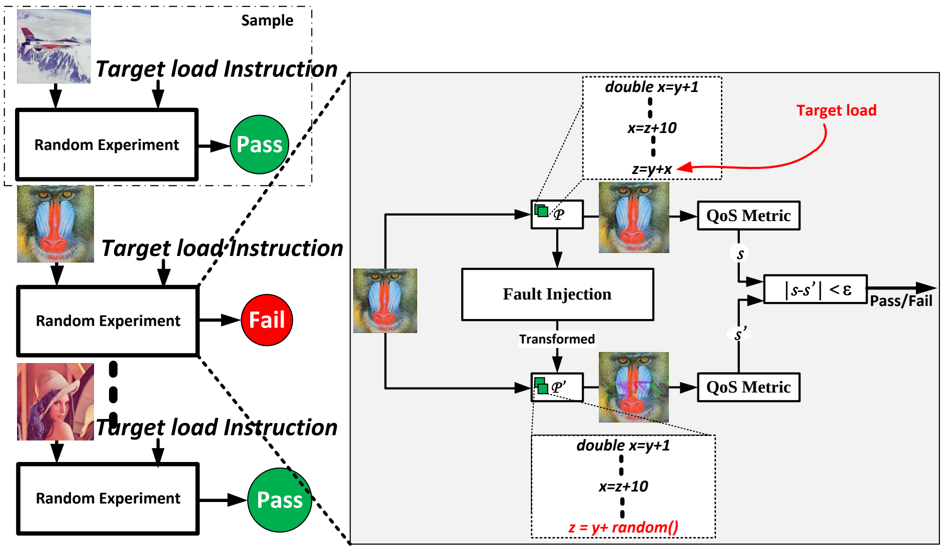


Figure 5.3: Our methodology of fault injection on an image processing application

and the fault injected execution are then compared using the QoS metric of the application. When the difference in QoS is within an acceptable tolerance ϵ, the experiment is marked as *Pass*. Otherwise, it is marked as *Fail*. □

Branch-Fault Injection Experiment : Let $\mathcal{B}$ be the set of program branches and E be the set of all possible executions of a program $\mathcal{P}$. Let $e \in E$ be an execution and $b \in \mathcal{B}$ be a branch at line ℓ of the program. When b is along the execution path of e, let b_e denote the branch direction (0 or 1) during the execution. A branch-fault injection experiment is the resulting execution e' when b_e is changed to $(1 - b_e)$. Essentially, such an experiment captures the effect of an execution along a wrong path due to a branch misprediction.

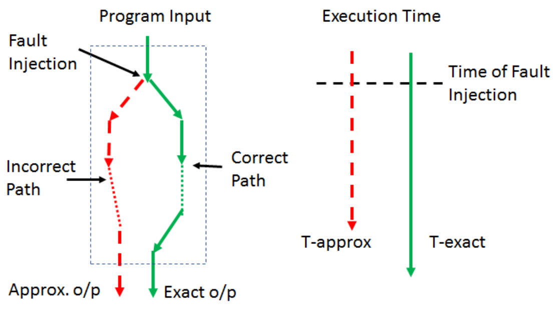


Figure 5.4: A fault injected path shorter than the correct path

Figure 5.4 shows pictorially a fault injection experiment on a program. After a fault injection, the program path may deviate from the correct path (shown as a bold line) and produce an incorrect path to completion (shown as a dashed line). If the new execution path produces an output with an acceptable error, the fault injection experiment provides an evidence for the hypothesis. The figure also depicts that the new incorrect path after a fault injection may result in a shorter path to completion, i.e., a path that terminates in a shorter time with respect to the exact path. This may happen due to fewer or less expensive instructions or both, on the incorrect path. □

Example 5.2 Figure 5.5 shows an overview of the branch-fault experiment methodology. Consider a basic block with a branch instruction labeled as target branch in the figure. The Green color blocks (A-B-D-H) show the exact path of a program for some input. Our branch-fault injection model flips the direction of the branch in block A. After the fault injection, the program path changes to A, C, E and H shown in Red. The result of exact execution and the execution with branch-fault at the target branch is then compared to determine the QoS degradation. When the degradation is within the tolerance ϵ, the experiment is marked as *Pass*. Otherwise, it is marked as *Fail*. □

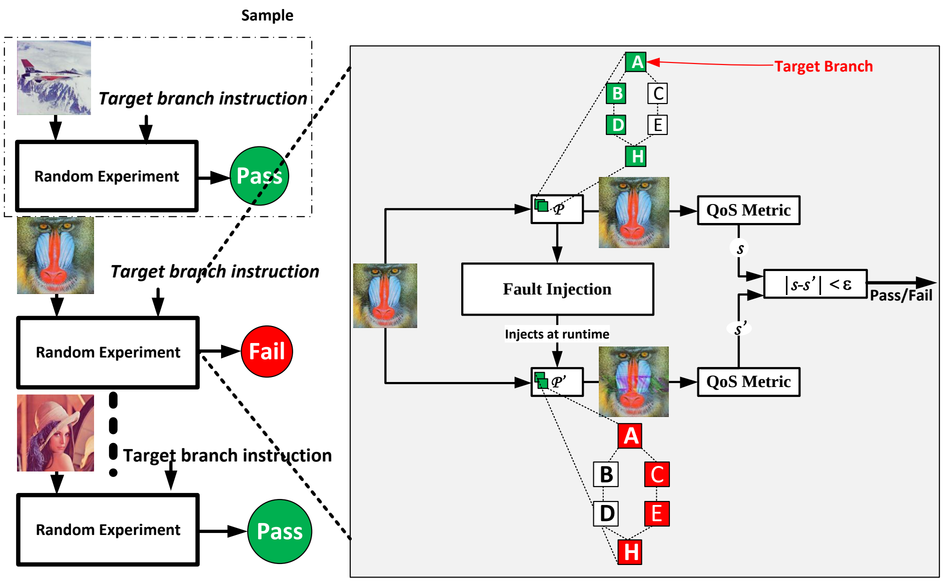


Figure 5.5: Experiment methodology of a random experiment using branch-fault injection experiment on an image processing application

We now recall the definition of a *sample space* of a statistical experiment [102]:

Definition 5.1 *The sample space of a statistical experiment is a pair* $(\Omega, \mathcal{S})$, *where*

1. Ω *is the set of all possible outcomes of the experiment.*

2. $\mathcal{S}$ *is the set of all subsets of* Ω.

Any set $A \in \mathcal{S}$ *is known as an* event. *If the outcome of an experiment happens to be an element of A, then we say that an event* A *has occurred.* ■

With respect to the sensitivity analysis of data / branch instructions, we consider the following sample space:

$$\Omega = \{X_i = x_i \mid x_i \in \{0, 1\}, 1 \le i \le k,\ k \ge 0\} \tag{5.1}$$

X_i is a random variable associated with the i^{th} sample and x_i is the outcome. A data / branch fault experiment which is a *pass* is represented with $x_i = 1$. A fault injection experiment which is a *fail* is represented with $x_i = 0$. Essentially, Ω represents the set of all possible outcomes of a sequence of k data/branch fault injection experiments in data / branch instructions of a program, for different program inputs. The notion of an *approximable program data* with respect to a user-defined Quality of Service (QoS) is formally defined as follows [67].

Definition 5.2 Approximable Data: *Given a confidence of inference θ and an application's allowable QoS distortion limit α, a program data $v \in \mathcal{D}$ is called approximable if and only if $\forall e \in E$ (the set of all executions), the probability that the program output remains within the acceptable QoS limit α on a data fault at v is at least θ. Formally, the definition is:*

$$\mathcal{AD} = \left\{ v \in \mathcal{D} \mid \forall e \in E, \forall \ell \in \ell_v^e, \langle v_e, \ell \rangle \rightarrow \langle v_{approx}, \ell \rangle \implies Pr(\mathcal{R} \in QoS_\alpha) \geq \theta \right\} \tag{5.2}$$

where $\mathcal{AD}$ denotes the set of approximable program data and $\langle v_e, \ell \rangle \longrightarrow \langle v_{approx}, \ell \rangle$ denotes the substitution of $\langle v_{approx}, \ell \rangle$ in place of $\langle v_e, \ell \rangle$. $\mathcal{R}$ denotes the program output. $Pr(\mathcal{R} \in QoS_\alpha)$ denotes the probability that the data-fault injection keeps the program output within the acceptable QoS limit α and it is the same as $Pr(X_i = 1)$, for any i. ■

On a similar note, we deem a branch as approximable when an incorrect direction of execution from that branch does not affect the output of the application beyond an acceptable error limit. Previous work has shown the existence of such branches in various applications, in both the scenarios when an approximation is permitted [56] and when no approximation is permitted [103]. We define an *approximable* branch as follows:

Definition 5.3 Approximable Branch: *Given a confidence of inference θ and an application allowable QoS distortion limit α, a branch $b \in \mathcal{B}$ is* approximable *if and only if $\forall e \in E$, the probability that the program output remains within the acceptable QoS limit α upon a branch fault on b, is at least θ. Formally, the definition is:*

$$\mathcal{AB} = \left\{ b \in \mathcal{B} \mid \forall e \in E, b_e \rightarrow (1 - b_e) \implies Pr(\mathcal{R} \in QoS_\alpha) \geq \theta \right\} \tag{5.3}$$

where $\mathcal{AB}$ denotes the set of approximable program branches and $b_e \longrightarrow (1 - b_e)$ denotes changing the branch direction of b corresponding to the execution e, to the alternate direction.

Now that we have defined an approximable load and branch in a program, we resort to our technique of automatically identifying them. In a manner similar to our idea presented in Chapter 3, we propose statistical hypotheses on the approximability of data and branches and test them with a testing procedure, particularly using the method of Sequential Probability

Ratio Testing (SPRT). The details of hypothesis testing can be found in Section 3.3.3 and the details of SPRT can be found in Section 3.3.4.

5.6 Cumulative Sensitivity Analysis using Bayesian Networks

The sensitivity knowledge of program data and branches given by the analysis discussed in the previous section provides us the effect of having an approximation at a particular data or a branch of a program at a time, during an execution. The cumulative effect of approximations in multiple data and branches cannot be obtained from the analysis. In this section, we present a study of the cumulative effect of approximation of program data and branches. We now propose to study the following:

- The cumulative effect of approximations in multiple data and
- The cumulative effect of approximations in multiple branches.

We first define the notion of *jointly approximable data* in a program $\mathcal{P}$:

Definition 5.4 ***Jointly Approximable Data:*** *Given a confidence of inference θ and an application allowable QoS distortion limit α, a set of program data $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$ is* jointly approximable *if and only if $\forall e \in E$ (E is the set of all executions), the probability that the program output remains within the acceptable QoS limit α when there are data-fault injections in all $d \in \mathcal{D}$ simultaneously, is at least θ.* ■

The concept of *Jointly approximable branches* in a program is similarly defined as follows:

Definition 5.5 ***Jointly Approximable Branches:*** *Given a confidence of inference θ and an application allowable QoS distortion limit α, a set of program branches $\mathcal{B} = \{b_1, b_2, \ldots, b_n\}$ is* jointly approximable *if and only if $\forall e \in E$, the probability that the program output remains within the acceptable QoS limit α when there are branch-fault injections simultaneously in all $b \in \mathcal{B}$, is at least θ.* ■

A simple method to infer the jointly approximable data and branches is by extending the hypothesis testing based analysis discussed in the previous section on multiple data and branches. The sample space will be similar to Eqn. 5.1, with the difference that X_i is now a random variable to represent the outcome of the i^{th} fault injection experiment in all data $d \in \mathcal{D}$ / all branches $b \in \mathcal{B}$ simultaneously. We then test: $H_0 : Pr(X_i = 1) \geq \theta$ and $H_0' : Pr(X_i = 1) < \theta$. Acceptance of the null hypothesis (H_0) implies that the set of data $\mathcal{D}$ or the set of branches $\mathcal{B}$ is *jointly approximable*. However, note that there can be an exponential number of possible queries on joint approximability of data / branches. Hypothesis testing to answer each such query is prohibitively expensive. To address this problem, we propose a method for inference of jointly approximable branches using Bayesian networks [104].

5.6.1 Bayesian Networks

Queries on jointly approximable data and branches can be answered with the *full joint probability distribution*. The full joint distribution for n Boolean variables essentially is a table containing 2^n rows and the corresponding probability that the events in the row jointly occur. Bayesian networks can concisely represent the full joint probability distribution. It is a data structure to represent dependencies among the random variables of the domain of interest, where random variables represent the events in the domain.

A formal definition of a Bayesian network is given below:

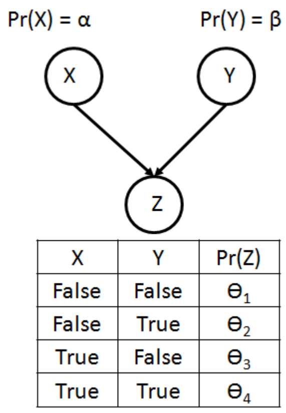
Definition 5.6 *A Bayesian network (BN) is a structure $\mathcal{D} = (\mathcal{G}(\mathcal{V}, \mathcal{E}), \mathcal{X}, parents, C_i)$, where*

- *$\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a directed acyclic graph with the set of vertices $\mathcal{V}$ as the random variables $\mathcal{X}$.*
- *$\mathcal{X}$ is the set of random variables, representing the events of interest.*
- *$parents : \mathcal{X} \rightarrow 2^{\mathcal{X}}$ maps a random variable X_i to its parents. A node X_j is said to be a parent of a node X_i if there is a directed edge from X_j to X_i in $\mathcal{G}$.*
- *C_i is the* conditional probability table *(CPT) associated with every variable $X_i \in \mathcal{X}$ specifying the probabilities $Pr(X_i \mid parents(X_i))$.*

where $parents(X_i)$ denotes parents of X_i. ■

In the context of this work, the domain of interest is a program. Given a program $\mathcal{P}$, the sample space is the set of all data and branch fault injection experiments E of $\mathcal{P}$.

Given a data fault injection experiment $e \in E$, we say that e is an evidence of the event represented by the random variable X if and only if the outcome of the experiment is a *success*. Similarly, as an outcome of branch sensitivity analysis, we say that a branch fault injection experiment is an evidence of the event represented by the random variable B if and only if the outcome of the experiment is a *success*. Therefore, given an experiment e, a random variable X takes the value *success* (1) or *failure* (0). In this setting, every random variable is Boolean. Given a Bayesian network (*BN*), the probability that events $X_1 = x_1$, $X_2 = x_2, \ldots, X_n = x_n$ jointly occur is given by [104]:



X	Y	Pr(Z)
False	False	θ_1
False	True	θ_2
True	False	θ_3
True	True	θ_4

Figure 5.6: A Bayesian network with three nodes

$$Pr(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} Pr(x_i \mid parents(X_i)) \tag{5.4}$$

where x_i denotes the value of the variable X_i and can be either $true$ or $false$. *Parents(X_i)* denotes the values of parents(X_i) that appear in $x_1, x_2, \ldots, x_n$. The product terms on the right hand side can be obtained from the CPT entries of the *BN*. ■

Example 5.3 We first discuss our proposal on constructing the CPTs of a *BN* for sensitivity analysis. For simplicity of illustration, we consider a *BN* with three random variables X, Y and Z such that Z has X and Y as its parents. This *BN* is shown in Fig. 5.6 along with the CPT at each of the nodes. The CPT at the nodes X and Y contains only the entries for $Pr(X)$ and $Pr(Y)$. The CPT of the node Z contains the value of $Pr(Z)$ for all possible values of its parents X, Y, as shown in Figure 5.6. □

Our method for constructing the CPT is a *Monte Carlo* method, based on *random sampling* of the program under analysis [105]. Consider S random samples, let S_X and S_Y be the set of samples out of S, which are an evidence of the events represented by random variables X and Y respectively. We denote the cardinality of a set S by $\mid S \mid$. Then, $Pr(X)$ and $Pr(Y)$ corresponding to the nodes X and Y in Fig. 5.6, are given by $\alpha = \frac{\left|S_X\right|}{\left|S\right|}$ and $\beta = \frac{\left|S_Y\right|}{\left|S\right|}$ respectively. In the figure, the last column of the CPT corresponding to the node Z shows $(Pr(Z)|X,Y)$. For example, $(Pr(Z)|X = true, Y = true)$ is denoted by θ_4 in the CPT of node Z. In order to compute θ_4, we count the number of samples which are an evidence of the events when X is $True$ and when Y is $True$, out of the S samples. Let us denote this by $S_{X\cap Y}$. We consider these $S_{X\cap Y}$ samples and see how many of these are an evidence of the event represented by the variable Z, let us denote this by $S_{Z\cap X\cap Y}$. We then compute $\theta_4 = \frac{\left|S_{Z\cap X\cap Y}\right|}{\left|S_{X\cap Y}\right|}$, using the following relation: $Pr(A|B) = \frac{Pr(A\cap B)}{Pr(B)}$. Similarly, we compute $\theta_1 = \frac{\left|S_{Z\cap\overline{X}\cap\overline{Y}}\right|}{\left|S_{\overline{X}\cap\overline{Y}}\right|}$,

$\theta_2 = \frac{\left|S_{Z\cap\overline{X}\cap Y}\right|}{\left|S_{\overline{X}\cap Y}\right|}$ and $\theta_3 = \frac{\left|S_{Z\cap X\cap\overline{Y}}\right|}{\left|S_{X\cap\overline{Y}}\right|}$.

The precision of the computed CPTs depends on the number of samples considered, as with any Monte-Carlo method. Based on this approach, we present Algorithm 5.1 to generate the CPTs at the nodes of a general BN, obtained from a program. In the algorithm, S denotes the set of random samples considered in the generation, and G represents the directed acyclic graph of the Bayesian network.

Algorithm 5.1 CPT generation by random sampling

```
1: procedure GENERATE-CPT($G$, $S$)
2:     for each node $X$ of $G$ do
3:         $S_X \leftarrow$ subset of S satisfying X
4:         $\overline{S_X} \leftarrow S - S_X$
5:     end for
6:     for each node $X$ of $G$ do
7:         $X_1, X_2, \ldots, X_k$ = parents(X)
8:         for each enumeration $e$ of $X_1, X_2, \ldots, X_k$ do
9:             for $i = 1$ to $k$ do
10:                if $X_i = true$ in $e$ then
11:                    $S_i = S_{X_i}$
12:                else
13:                    $S_i = \overline{S_{X_i}}$
14:                end if
15:            end for
16:            $Pr(X|e) = \frac{\left|(\cap_{i=i}^{k} S_i)\cap S_X\right|}{\left|\cap_{i=i}^{k} S_i\right|}$
17:        end for
18:    end for
19: end procedure
```

5.6.2 The Bayesian Network Structure

A Bayesian network models the conditional dependency between the events via the directed edges. We consider to use the acyclic *data dependency graph* of a program to reason about joint sensitivity of program data and the *dominator tree* of a program to reason about the joint sensitivity of program branches. This choice is based on the observation that the data-dependency graph and the dominator tree of a program intrinsically capture the conditional dependency between the program data and branches respectively [106]. We now elaborate our methodology using a dominator tree. In a dominator tree, the program branches are taken as nodes of the tree and there is an edge from a branch node b_1 to a branch node b_2 if and only if b_1 is an *immediate dominator* of b_2. We define the *dominator* and *immediate dominator* relations between program branches as follows:

Definition 5.7 *Given program branches b_1 and b_2, we say that b_1* dominates *b_2 if and only if all program paths to b_2 in the CFG of the program pass via b_1. Branch b_1 is said to be an* immediate dominator *of branch b_2 if and only if b_1 dominates b_2 and every branch b that dominates b_2 also dominates b_1.* ■

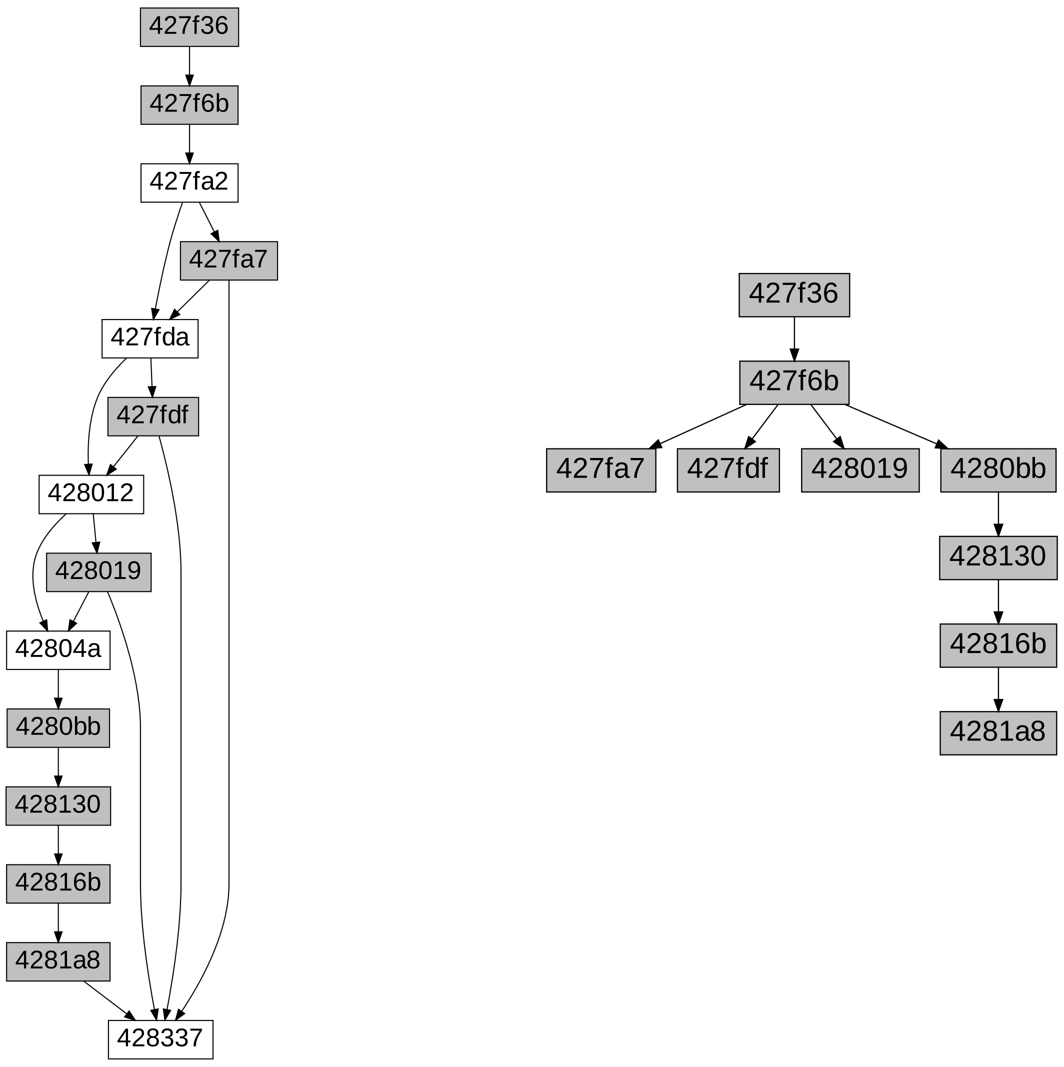


Figure 5.7: The CFG and the Dominator tree consisting of the approximable branches of a code snippet from JPEG encoder

Example 5.4 As an example, consider a CFG and the corresponding dominator tree of a code snippet from the *build_huffman* method in the *JPEG Encoder* application shown in Fig. 5.7. The nodes shown in gray are the ones identified as approximable by our analysis, whereas the nodes shown in white are the ones which are found to be intolerant to approximation, given a permissible QoS distortion and a confidence measure. □

We are interested in probing the joint probability distribution of only those program branches which are known to be approximable individually. This is because the answer to a query for deriving the probability of a set of branches being jointly approximable which contains a sensitive branch is going to be trivially 0. We obtain the sensitivity knowledge of the individual nodes using the hypothesis testing procedure discussed earlier.

Figure 5.8 shows the essence of the discussion on constructing a Bayesian network from a program. First, the CFG is extracted from the program to be analyzed. The results of sensitivity analysis by hypothesis testing of the candidate branches in the program are taken to trim the nodes of the CFG corresponding to the sensitive branches. Then, a dominator tree is constructed from the trimmed CFG and taken as the Bayesian network structure, for branch sensitivity analysis. Finally, the CPT at each node is obtained using the proposed sampling based algorithm in Algorithm 5.1.

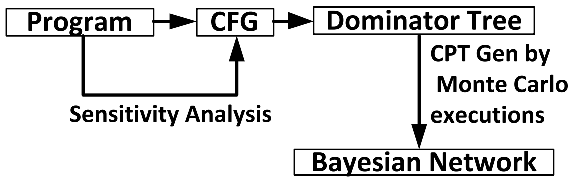


Figure 5.8: A block diagram depicting the construction of a Bayesian network from a program.

The performance of the Bayesian analysis depends on (1) the number of random executions and (2) the size of the Bayesian network and (3) the time to complete a fault injection experiment. (1) can be tuned for the desired precision. Let k be the number of nodes in the directed acyclic graph of the Bayesian network, S be the number of fault injection experiments (random samples) and t_e be the average time of completing a fault injection experiment, d be the maximum degree of the graph. The performance of the CPT generation algorithm is $O(k * 2^d) + O(S * k * t_e)$. The time to complete the k fault injection experiments (samples) at each data / branch node of the graph takes $O(S * k * t_e)$ time. The time taken to compute the CPT entries at each node takes $O(k * 2^d)$ time.

5.7 Implementation

In this section, we discuss our methodology to use the approximable loads and branches to improve runtime performance of programs. A schematic of our work is shown in Figure 5.9. It consists of two main steps, the *pre-execution analysis* and an *architectural simulation*. The pre-execution phase identifies approximable load / branch instructions and the architectural simulation phase simulates rollback-free execution of a load / branch instruction. For approximable load instructions, the load value is not fetched from the next level memory in the event of a D1 cache miss in the first level data cache, instead, an approximate value in the register is used, without causing a pipeline stall. No load value verification and no rollback is associated with such instructions. Similarly, no pipeline flush and reload is performed when a misprediction is detected in an approximable branch. We implement the misprediction penalty free execution scheme in the Sniper architectural simulator [97].

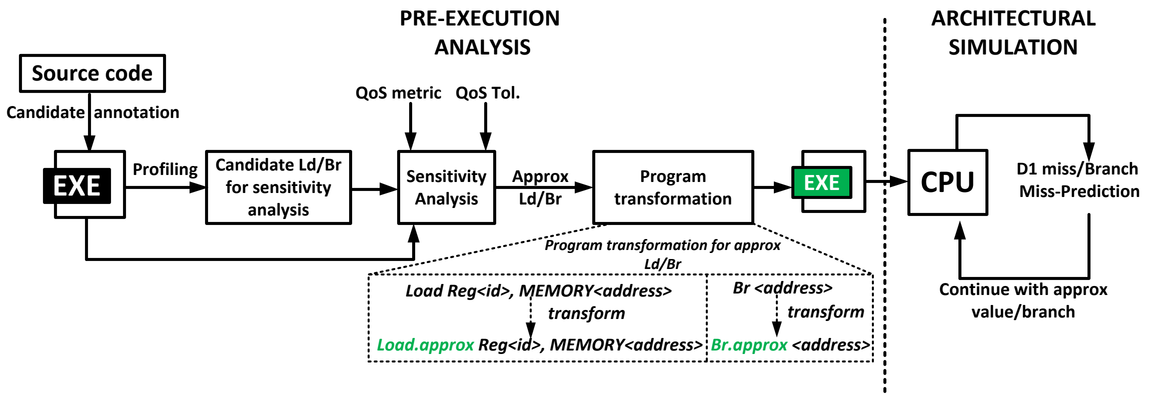


Figure 5.9: A schematic of the implementation

5.7.1 Pre-execution analysis

As mentioned earlier, the *pre-execution analysis* in the figure consists of three main steps, namely *Candidate annotation for sensitivity analysis*, *sensitivity analysis*, and *program transformation*.

Candidate annotation for sensitivity analysis: This step uses tools to identify the candidate instructions for sensitivity analysis. Cache-miss heavy load instructions are identified by cache profiling using *Cachegrind* [107]. Hard to predict branches are identified by collecting misprediction statistics using the Pin tool [34] implementing the *two bit saturating counter* branch predictor [108]. Instructions which show high cache miss and branch misprediction rate are considered as candidates for approximation. In the case of load instructions, we select the ones for sensitivity analysis that have at least 100 cache misses. For branch instructions, the sensitivity analysis is carried out similarly for those instructions that have at least 100 mispredictions.

Sensitivity analysis: We perform fault injection experiments of the SPRT routine using binary instrumentation, implemented on top of the Pin tool [34]. A fault is injected in a load-instruction by injecting an erroneous load value in the respective register. A fault is injected in a branch-instruction by flipping the status flags in the *EFLAGS register* which are necessary to negate the outcome of the branch.

Program transformation: The set of approximable load / branch instructions are replaced with their approximable counterparts in the program. This is an extension we propose to add to the Instruction Set Architecture (ISA) such that the processor is able to differentiate approximable instructions from the normal ones, and exercise the roll-back policy accordingly. For instance, Load *Reg <id>, MEMORY <address>* is replaced with *Load.approx Reg <id>, Memory <address>* and *Br <address>* is replaced with *Br.approx <address>*. In order to pass the information to the hardware, we propose an extension to the ISA similar to [109] to implement the idea of misprediction penalty-free execution of approximable load and branch instructions. An additional bit is set in the opcode of a load, branch instruction to indicate to the micro-architecture that it is approximable.

After the pre-execution analysis, an architectural simulation of the approximate program is performed. The implementation details of the architectural simulator is discussed in the following.

5.7.2 Architectural Simulation

In this section, we discuss the hardware implementation of our proposed approach. When a load/branch instruction is not approximable, it executes normally. Moreover, an approximable load executes precisely when the load address is present in the cache. The approximate execution of the load is triggered only in the event of a cache miss on the load address, when the pipeline proceeds with the content of the destination register as the approximate load value. Similarly, the approximable branch instruction executes normally when no branch misprediction is detected. In the event of a branch misprediction, the pipeline proceeds with no flushing.

Hardware Design

Figure 5.11 shows the hardware implementation of our design using a 5-stage pipeline architecture. We introduce two 32-bit registers - *Ap_Ld_Reg* and *Ap_Br_Reg* to store the load and branch instruction types respectively of all the loads and branches in the pipeline. We intend to use these registers as a FIFO (First In First Out) queue to know the load / branch types later in the execution stage. When a new load / branch instruction is in the fetch stage of the pipeline, the MSB of *Ap_Ld_Reg*/*Ap_Br_Reg* is set if it is approximable. In order to have a FIFO access of the instruction types from the LSB of these registers, we perform an appropriate number of right shift operations on the MSB bit. For example, the type of the first load / branch instruction in the pipeline is inserted at the MSB (31st bit). We then perform 31 right shift operations to shift the MSB to LSB (0th bit) position.

Similarly, the type of the second load / branch instruction in the pipeline is set at the MSB and we now perform 30 right shift operations to shift the MSB to the 1st bit position and so on. We keep an additional 16-bit register *Cntr_Reg* to keep the count of the load / branch instructions in the pipeline. The 8-bits from its LSB are used to store the load instruction

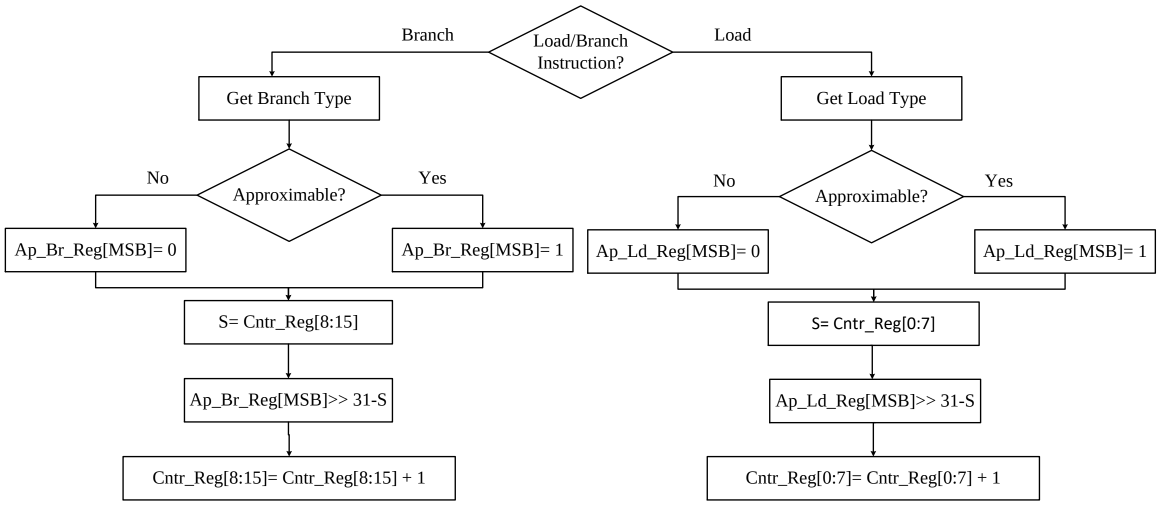


Figure 5.10: Flowchart to implement the fetch stage for a branch and a load instruction

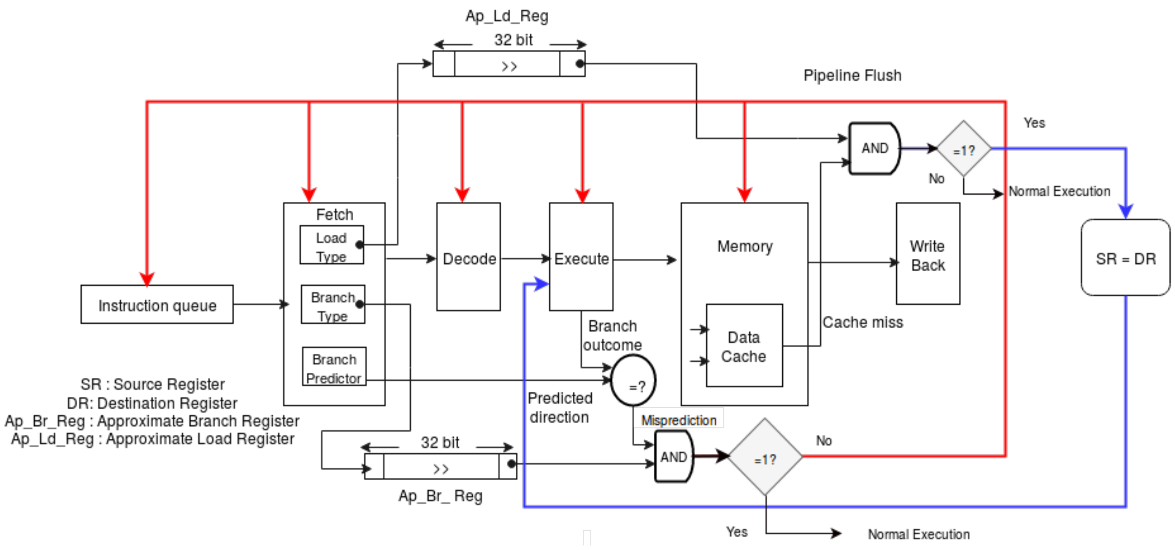


Figure 5.11: Hardware architectural design

count and the 8-bits from its MSB are used to store the branch instruction count. When a load / branch enters the pipeline, the corresponding *Cntr_Reg* value is incremented. It is decremented when the instruction exits from the pipeline. In general, if the load / branch count in *Cntr_Reg*[7:0]/*Cntr_Reg*[15:7] is S, then $(31 - S)$ right shift operations are performed on the MSB of *Ap_Ld_Reg*/*Ap_Br_Reg* after being set. During the execution stage of a load / branch instruction, the LSB of *Ap_Ld_Reg* / *Ap_Br_Reg* is read to know the load / branch type (approximable or precise). After reading, the register content is shifted one position right to have the effect of deleting the last read value. Figure 5.10 shows the execution steps in setting the registers and the counter in the fetch stage.

For a branch instruction, the branch predictor predicts the branch direction at the **Fetch stage**. When the branch condition is later resolved at the **Execution stage**, the actual branch direction is compared with the predicted direction as shown in the figure. The pipeline proceeds normally when there is no mis-prediction. In the event of a mis-prediction, the branch-type is read from the *Ap_Br_Reg* register to decide whether to flush the pipeline.

For a load instruction, when a cache miss occurs at the **Memory stage**, the load-type is read from the *Ap_Ld_Reg* register. Depending on the load type, either the next level of memory is accessed to get the exact data needed for the execution or the pipeline proceeds with the execution using the content of the destination register (DR). Instead of a random value, we use the content of DR for approximable load instructions. Overall, the proposed hardware design additionally requires three comparators, two AND gates, two 32 bit registers and a 16 bit counter register.

Example 5.5 Let us consider a simple assembly code as shown in Figure 5.12 containing 2 conditional branches and 1 unconditional branch. We walk through the execution of the branch instructions in the proposed hardware pipeline.

```
        BNE loc1
        ....
        CMP R0, #0 ; check if ro==0
        BNE_APPROX loc2 ; if R0!=0 branch to loc2
        ADD R1,#1 ; r1=r1+1
        BR next ; unconditional branch to next
        loc2 :      ADD R2,#1 ; R2=R2+1
        ....
        loc1 :      ....

```

Figure 5.12: Assembly instructions containing one approximable branch (BNE_APPROX)

- When the instruction **BNE loc1** is in the fetch stage of the pipeline, we set $Ap\_Br\_Reg[MSB] = 0$ to signify that it is a non-approximable branch. 31 right shift operations are performed in this register to shift the MSB to the LSB position.
- The $Cntr\_Reg[8:15]$ is then incremented by 1 to signify that one branch instruction is present in the pipeline.
- Since this is a non-approximable branch, in the event of a misprediction of this branch during the execution stage, the pipeline is flushed and reloaded with the content of the correct branch.
- When the instruction exits the pipeline, the $Cntr\_Reg[8:15]$ register is decremented.
- For the branch instruction **BNE loc1**, we assume that its outcome is false and hence the execution follows the path that goes to the next instruction outside the branch.

- When the branch **BNE_APPROX loc2** at Line 4 is in the fetch stage of the pipeline, the register $Ap\_Br\_Reg[MSB]$ is assigned 1 since it is an approximable branch.
- The $Cntr\_Reg[8:15]$ is then incremented by 1.
- If this branch is mispredicted during the execution stage, since this is an approximable branch, the execution in the pipieline proceeds as usual, causing no roll-back and reloading of the pipeline.
- When the instruction exits the pipeline, the counter $Cntr\_Reg[8:15]$ is decremented. □

5.8 Evaluation

We present an evaluation of our sensitivity analysis techniques, followed by a performance and energy evaluation of the proposed rollback-free execution of approximable *load* and *branch* instructions on a subset of three benchmark suites, AxBench [72], ACCEPT [99] and Parsec 3.0 [98]. We first briefly introduce the applications considered for evaluation and the corresponding QoS metric.

5.8.1 Applications for Evaluation

We consider *JPEG-Encoder*, a lossy compression algorithm, *Sobel* - an image processing application for edge detection, *Inversek2j* - an inverse kinematics application for a 2-joint arm and *x264* - a video-stream encoder, from the AxBench suite of benchmarks [72]. In the ACCEPT benchmark suite [99], we have *Stream-cluster* - an algorithm to cluster a set of data points, and *Blackscholes* - an application which predicts the prices of European stock options analytically using Black-Scholes partial differential equations. From the Parsec 3.0 benchmark suite [98], we consider the *Bodytrack* application. It is a computer vision application to track the posture of a human body from a sequence of video frames. We also evaluate on *Jmeint* and *Raytracer* mentioned in Chapter 3 Section 3.5.1.

5.8.2 QoS Metric

In Chapter 3 Section 3.5.2, we discussed some QoS metrics associated with each application domain taken for evaluation. In addition to those metrics, we use the following in this work to demonstrate the performance of our method.

Structural Similarity (*SSIM*): The *Structural Similarity* index is used for measuring the similarity between two images. It is used to measure the noise in an image with respect to the reference image in terms of the *SSIM* index. *SSIM* is an index valued between 0 and 1. *SSIM* index of 0 between two images means that they have no similarity, and an index of 1 means that they are exactly similar. We direct the readers to [100] for details on the computation of the *SSIM* index. We use this metric to compare the QoS distortion in *Sobel*, *Bodytrack* and *x264* applications. In particular, we find this to be a better metric than *PSNR* for measuring the QoS distortion in images introduced due to approximation, in the *Sobel* application. Some of the output images produced by *Sobel*, have a large perceptible noise cause by approximation, reported an acceptable *PSNR*. However, they have an unacceptable *SSIM* index. We use an *SSIM* index value of ≥ 0.8 as the acceptable threshold, which is the standard threshold for acceptable noise in images [100].

Cluster Center Distance (*CCD*): It is a metric to measure the collective distance between K pairs of vectors in the Euclidean space. We use this metric to measure the change in the cluster centers after introducing approximation in the *Stream-cluster* application. The computation of *CCD* is performed as:

$$\begin{aligned} CCD = minimize \sum_{i=1}^{K}\sum_{j=1}^{K} \frac{d_{i,j}x_{i,j}}{\sqrt{n}} \\ subject\ to \\ \sum_{i=1}^{K} x_{i,j} = 1,\ for\ all\ j = 1\ \dots K \\ \sum_{j=1}^{K} x_{i,j} = 1,\ for\ all\ i = 1\ \dots K \end{aligned} \tag{5.5}$$

where $d_{i,j}$ is the Euclidean distance between the i-th exact cluster center and the j-th approximate cluster center, among the k pair of exact and approximate cluster centers. n is the dimension of a cluster center.

Average Relative Error: It is a metric to measure the QoS distortion in applications that produce an array of values as output. It is given by:

$$Avg\_Err(X, X') = \frac{1}{N}\sum_{i=1}^{N}\frac{|X_i - X'_i|}{X_i} \tag{5.6}$$

X and X' are the reference and the approximate value respectively. N is the number of outputs.

5.8.3 Evaluation of Sensitivity Analysis by Hypothesis Testing

Application	LOC	QoS-metric	QoS Tol.	Load Sensitivity					Branch Sensitivity				
				% D1 miss	Load Tested	Load Appr.	% Appr.	Time (Hrs)	% Mis-pred.	Br. Tested	Br. Appr.	% Appr.	Time (Hrs)
JPEG encoder	482	PSNR	≥ 20	0.5	293	52	17.7	12	14.4	416	163	39	10.3
Stream-cluster	1465	Cluster center distance	≤ 0.1	0.7	186	45	24	5	7.5	208	112	53.8	8.5
Blackscholes	345	Average relative error	≤ 0.1	1.9	59	13	22	4	12.8	16	3	18.8	0.5
Sobel	149	SSIM	≥ 0.8	0.1	22	8	36.5	2.3	14.1	11	1	9	1
JmeInt	338	Matching	0	0.3	92	24	26	0.4	9.4	42	19	45	0.4
Bodytrack	16406	SSIM	≥ 0.8	5.7	280	27	9.6	8.6	5.4	161	84	51.1	7
x264	59982	SSIM	≥ 0.8	4.8	500	57	11.4	41	8.7	671	402	59	58
Raytracer	1006	PSNR	≥ 20	0.0	51	9	18	29	9.1	46	13	28.2	23
Inversek2j	77	Average relative error	≤ 0.1	0.76	12	0	0	2.7	5.3	11	6	54	2.4

Appr.:Approximable; %Appr. : Percentage of the approximable tested data and branches, Br. : Branches
We see that a considerable percentage of program data and branches tested are approximable.

Table 5.3: Sensitivity analysis results by hypothesis testing.

We first provide an evaluation of the proposed sensitivity analysis method for data and branches in a program by statistical hypothesis testing. Table 5.3 shows the results of our sensitivity analysis on program data load and branch instructions, performed with a confidence probability of $\theta \geq 0.5$. The 1st column shows the benchmarks, the 2nd shows the number of lines of code of each application, the 3rd column shows the metric used to measure the QoS distortion of the respective application. The 4th column shows the tolerance of QoS distortion in the respective application. The 5th to 9th columns show the cache miss percentage reported by the profiler on the level 1 (L1) data cache (D1), the number of load instructions analyzed for sensitivity, the number of load instructions identified as *approximable*, the percentage of the analyzed load instructions that are found to be approximable and the time to complete the analysis respectively. The cache configuration taken is shown in Table 5.5.

The 10th to 14th columns of Table 5.3 show the percentage of branch misprediction reported by the profiler, the number of branches analyzed for sensitivity, the number of branches classified as *approximable*, the percentage of the analyzed branches that are found to be approximable and the total analysis time respectively. Our experimental results show that a considerable number of data and branches in the applications are approximable when QoS distortion in permissible limits is acceptable. Evidently, we see a maximum of 36.5% of the tested data in the *Sobel* application and 59% of the tested branches in the *x264* application to be approximable, with a confidence probability of more than 0.5. This shows that there is a considerable scope of enhancing speculative executions with both load values and branches in these applications and these points of approximation can be automatically identified with our analysis.

We observe that performing the sensitivity analysis of data and branches one at a time using hypothesis testing is expensive, particularly in those applications which are expensive to execute to completion. We see that in applications like *x264* and *Raytracer*, testing the sensitivity of 500 and 51 load instructions takes nearly 41 and 29 hours respectively. Similarly, testing the sensitivity of 402 and 13 branches in *x264* and *Raytracer* takes nearly 58 and 23 hours respectively.

5.8.4 Evaluation of Sensitivity Analysis Using Bayesian Networks

In the applications, we construct the Bayesian network as discussed in Section 5.6.2, function-wise. Table 5.4 depicts the probability of a set of individually approximable branches to be jointly approximable, in certain functions from *x264*, *Stream-cluster*, *JPEG Encoder*, *Body-track*, *Raytracer* and *JMEint* applications. The **Function** column mentions the function name and the **Br** column shows the number of approximable branches in the corresponding function, identified using the hypothesis testing procedure. These branches essentially constitute the nodes of the Bayesian network. The **Pr.** column shows the probability of the approximable branches in the function to be jointly approximable, obtained from the joint probability distribution represented by the Bayesian network that we construct from our methodology discussed in Section 5.6. The **Time** column shows the time taken to construct the Bayesian network in order to compute the joint probability.

We see that out of all the tested branches, a maximum of 51 branches in *x264*, 12 branches in *Stream-cluster*, 20 branches in *Bodytrack* and 5 branches in *Raytracer* came out to be jointly approximable with a probability of 1. In the case of *Jpeg encoder* and *Jmeint*, we found a maximum of 11 and 14 branches to be jointly approximable with a probability of 0.93 and 0.52 respectively. These sets of branches in the applications are *jointly approximable* by our definition (Definition 5.5) with a confidence of $\theta \geq 0.9$ and with a QoS metric, distortion tolerance mentioned in Table 5.3. We observe that there are many potential sets of branches in these applications which can be jointly approximated. Performance-wise, the time taken to compute the joint probability on an application is the sum of the times taken to construct the Bayesian network for each function. The time taken to compute the joint probability in *x264*, *Stream-cluster*, *Jpeg encoder*, *Bodytrack*, *Raytracer* and *Jmeint* is approximately 96.93, 6.41, 3.95, 10.5, 12.4 and 0.6 hours respectively.

	Function	#Br.	Pr.	Time (s)
x264	get_ref	21	0.76	80064
	mc_chroma	4	1	3604
	mc_luma	13	1	8617
	mc_weight_w16	3	1	2818
	refine_subpel	33	0.7	23967
	refine_subpel.constprop.0	22	1	19328
	x264_analyse_update_cache	4	1	4848
	x264_cabac_block_residual_8x8_rd_c	6	1	6543
	x264_cabac_block_residual_c	2	1	1392
	x264_cabac_block_residual_rd_c	2	1	1857
	x264_frame_expand_border	3	1	3767
	x264_frame_expand_border_filtered	2	1	2871
	x264_frame_expand_border_lowres	3	1	3412
	x264_intra_rd	2	1	2812
	x264_macroblock_analyse	51	1	37294
	x264_macroblock_cache_load_progressive	16	0.62	12807
	x264_macroblock_cache_save	4	0.74	4171
	x264_macroblock_encode	5	0.9	4957
	x264_macroblock_write_cabac	5	0.52	4941
	x264_mb_analyse_inter_p16x8	8	1	2529
	x264_mb_analyse_intra	17	1	13433
	x264_mb_analyse_intra_chroma.part.25	6	1	2289
	x264_mb_analyse_p_rd	5	1	3658
	x264_mb_analyse_transform_rd.part.26	4	1	2529
	x264_mb_encode_chroma	4	0.37	2762
	x264_mb_mc	5	1	3324
	x264_mb_predict_mv	12	1	6841
	x264_mb_predict_mv_16x16	4	1	3503
	x264_mb_predict_mv_ref16x16	2	1	2303
	x264_me_search_ref	13	1	10246
	x264_pixel_sad_x3_16x16	2	1	2303
	x264_rd_cost_mb	2	1	1424
	x264_predict_lossless_4x4	2	1	2045
	x264_predict_lossless_16x16	2	1	1870
	x264_slice_write	8	1	4882
	x264_rd_cost_mb	2	1	1424
	x264_pixel_sad_x4_16x16	4	1	2972
	x264_pixel_sad_x4_8x8	4	1	1422
	x264_pixel_x3_8x8_c	3	1	2321
	x264_sad_16x8c_p_c	4	1	3021
	x264_predict_lossless_16x16	2	1	1870
	x264_predict_lossless_4x4	2	1	2045
	x264_rd_cost_mb	2	1	1424
STREAMCLUSTER	__static_initialization_and_destruction_0	2	1	853
	_M_convert_to_wmask	6	1	3148
	moneypunct<char, true>::_M_initialize_moneypunct	2	1	1954
	moneypunct<char, false>::_M_initialize_moneypunct	2	1	1953
	moneypunct<wchar_t, true>::_M_initialize_moneypunct	2	1	1953
	_M_initialize_ctype	2	1	1941
	_M_install_facet	2	1	1200
	_M_initialize_numpunct	3	1	1182
	__ieee754_log_avx	3	1	1076
	__init_cpu_features	12	1	3165
	__run_exit_handlers	2	1	881
	init_cacheinfo	4	1	1372
	uselocale	2	1	897
	int_free	3	1	611
	IO_new_do_write	3	1	350
	IO_vfprintf	3	1	347
	printf_fp	10	1	1229
JPEGENCODER	image_compare	3	1	2730
	fread	2	0.90	1040
	build_huffman	11	0.93	3065
	create_png_image_raw	2	0.87	1306
	compute_quant_table	2	1	794
	intel_check_word	8	1	2960
	memcpy	3	0.69	1103
	decode_next_row	2	0.89	759
	quantize_pixels	2	0.91	512
BODYTRACK	FlexImage<unsigned char, 1>::Set	7	1	3356
	FlexLine	7	1	3406
	BodyGeometry::IntersectingCylinders	4	1	2306
	TrackingModel::CreateEdgeMap	7	1	3336
	TrackingModel::OutputBMP	9	1	4119
	TrackingModel::GetObservation	7	1	3365
	RandomGenerator::Rand	2	1	1492
	ImageMeasurements::InsideError	8	1	3779
	ImageMeasurements::ImageErrorEdge	2	1	1506
	ImageMeasurements::ImageErrorInside	2	1	1509
	ImageMeasurements::EdgeError	20	1	8382
	MultiCameraProjectedBody::ImageProjection	2	1	1492
RAYTR	Intersect	3	1	14289
	QueryScene	5	1	26895
	Raytrace	2	1	3712
JMEINT	tri_tri_intersect	14	0.52	1921
	vfprintf	2	1	521

Br. : Branches; Pr.: Probability; s.: seconds, RAYTR. : RAYTRACER

Table 5.4: Application's jointly approximable branches

5.8.5 Evaluation of Speculative Execution with Selective Approximation

We now present the benefits of sensitivity analysis in speculative execution for approximate computing benchmarks using rollback-free execution on the approximable load and branch instructions. Our comparison is against the baseline execution where a branch misprediction for any branch results in a penalty of pipeline flushing, and a cache-miss in the D1-cache for any data results in fetching the correct data from the memory. The system configuration of the simulation experiments in *Sniper* is shown in Table 5.5.

Architecture	x86 with clock frequency of 2.66GHz, single core
Pipeline	5 stage In-order pipeline of width 1
Branch Predictor	Pentium M, penalty: 15 cycles
Private L1 Cache	8KB, 4-way, 32 byte blocks, 3-cycles latency
Shared L2 Cache	32KB, 4-way, 32 byte blocks, 32-cycles latency
Main Memory	A miss in L2 Cache is considered as a hit in the Main memory with a miss penalty of 173-cycles, per controller bandwidth 2.5 GB/s
Power Model	McPAT [110]
VDD	1.6 Volts
Technology node	45 nm

Table 5.5: System Configuration

Performance Evaluation

In the evaluation, we assume that all the individually approximable branches and data are also jointly approximable. This assumption is to show the maximum possible benefit in terms of CPU cycles and the reduction in energy utilization that can be obtained with our rollback-free speculative execution method with our sensitivity analysis. We now define some terms to describe our experimental observations:

Definition 5.8 *The percentage reduction in CPU cycles is computed as* $\frac{(C_e - C_a)*100}{C_e}$*, where* C_e *and* C_a *are the CPU cycles with usual execution and with our proposed method on approximable data loads and branches respectively.* ■

Definition 5.9 *The percentage reduction in energy consumption is computed as* $\frac{(J_e - J_a)*100}{J_e}$*, where* J_e*,* J_a *are the energy footprints with usual execution and with our method respectively.* ■

Definition 5.10 *Drop rate is defined as the fraction of cache misses on approximable data loads that do not initiate memory access requests [109].* ■

The percentage reduction in CPU cycles and energy reduction are shown using 100% drop rate. This is a tunable parameter to observe the performance and energy benefits against QoS trade-off.

The configuration where a cache-miss on an approximable load memory request is never requested from the primary memory is referred to as 100% drop rate. Figure 5.13 and Figure 5.14 respectively show the percentage reduction in CPU cycles and energy consumption with our proposed method of rollback free execution on only the approximable loads, only the approximable branches, and both. We observe an average of 22.85% reduction in CPU cycles and 23.09% reduction in energy consumption over nine applications. We also observe that the performance of the application *Jmeint* deteriorates with our no roll-back proposal on approximable loads. We believe that the reason of such a deterioration is because of

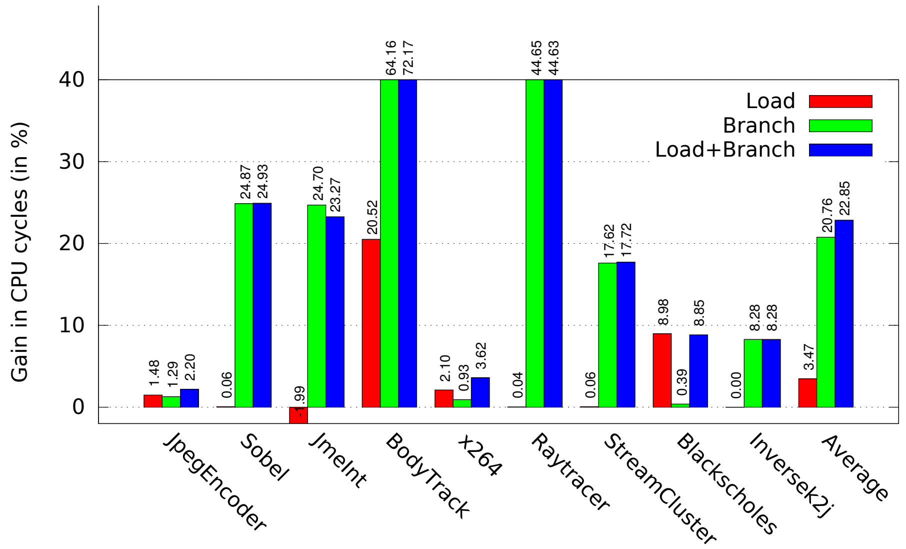


Figure 5.13: Percent gain in CPU cycles with selective no-rollback execution of approximable loads and branches.

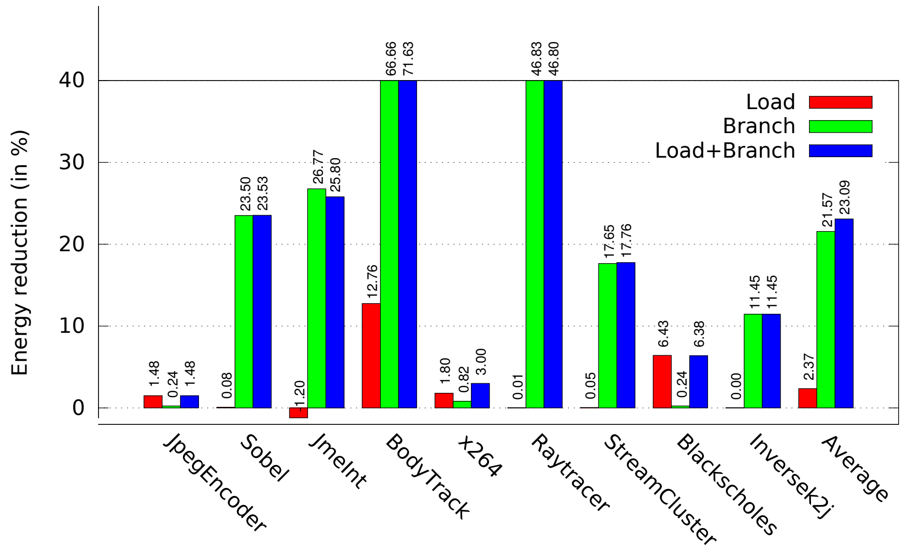


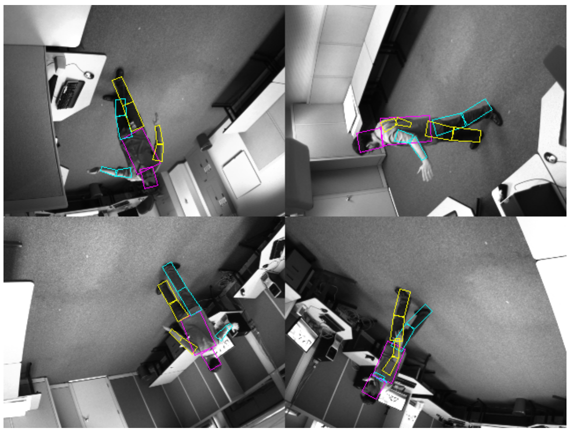
Figure 5.14: Percentage reduction in energy utilization with selective no-rollback execution of approximable loads and branches.

the wrong execution paths taken due to incorrect load values resulting in more expensive execution, either due to longer paths or expensive instructions in the alternate path.

QoS Evaluation

Fig. 5.15 shows the QoS trade-off with a speculative execution with no rollback policy on error tolerant loads and branches in the application *Bodytrack*. It is a computer vision application to track the 3D-posture of a human body from a sequence of images captured from multiple cameras. The application tracks the human pose in the image using the edges and foreground silhoutte as the features of the image. The application models the human body with 10 conic cylinders, which are superimposed on the input image to show the human pose [98].

Figure 5.15a shows the exact output consisting of an image with the application detected human body pose shown with 10 conic-cylinders. Note that the human torso and the head, the left hand and leg, and the right hand and leg are shown with Pink, Cyan and Yellow conic-cylinders respectively. Figure 5.15b shows the output with misprediction penalty-free execution of approximable loads and branches. We observe that the application is still able to detect most of the human body parts but the colour distinction is lost. For example, the human torso and head are shown in Yellow in the distorted image instead of Pink. The torso, the head and the left hand position are not detected in the top-left human in the image. We evaluate this loss of image quality in terms of SSIM (Structural Similarity Index) and it turns out to be 0.953. This is 4.7% loss [2] with respect to the exact image in terms on SSIM. Since our specified QoS tolerance is SSIM $\geq$ 0.8, this distorted image passes the QoS requirement. This shows that our load and branch sensitivity analysis performed with a QoS definition of SSIM $\geq$ 0.8, faithfully classifies the approximable loads and branches. A rollback-free execution of these loads and branches results in nearly 72% reduction in CPU cycles and energy consumption. In conclusion, we see that with an allowable degradation of 4.7% in terms of our QoS definition in *Bodytrack*, a substantial benefit of nearly 72% reduction in CPU cycles and energy consumption can be obtained.


(a) Exact Output

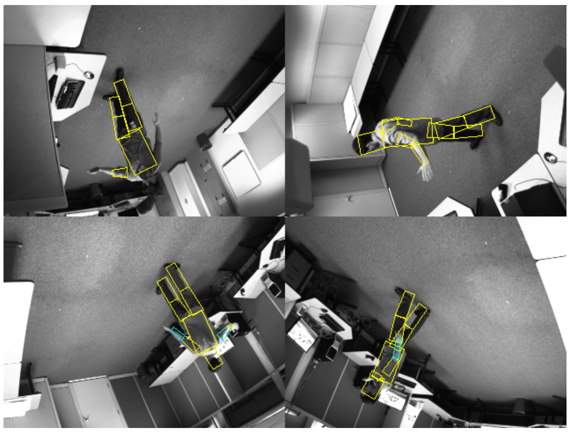

(b) Load & branch approximation output

Figure 5.15: QoS trade-off in Bodytrack with our method on loads and branches.

[2] $loss\% = \frac{SSIM(exact\_image)-SSIM(approx\_image)}{SSIM(exact\_image)} \times 100$, where exact_image and approx_image are images produced by exact and approximate execution of a Bodytrack application

5.8.6 Reliability Evaluation of the Bayesian Analysis–Based Method

In order to test the reliability of our Bayesian analysis based method, we perform an experiment to inject faults simultaneously at all the jointly approximable branches given by the analysis and observe the resulting distortion in the QoS. We perform 1000 such fault injection experiments and observe how many of these pass the QoS requirement. The QoS metric and tolerance measures considered in the reliability evaluation per application are the same as considered for sensitivity analysis, as mentioned in Table 5.3. The ratio of the passed experiments to the total number of experiments gives the probability of the branches to be jointly approximable. We compare this number with our computed joint probabilities using Bayesian analysis. This comparison is shown in Table 5.6. In the table, the first column mentions the application, the second column shows the number of branches identified as jointly approximable by our analysis in the corresponding application, the third column shows the empirically computed probability that the identified branches are jointly approximable, and the last column shows the computed probability of the identified branches to be jointly approximable using our Bayesian analysis based method. We observe that the empirically computed probability exactly matches with the probability computed by our method, in four of the applications. In two of the applications, namely *Jpeg encoder* and *x264*, the empirical and the computed probabilities differ by 0.01 and 0.3 respectively. This establishes the correctness of our analysis. In case of the *Jpeg encoder*, we similarly test the reliability of the analysis by injecting a fault simultaneously in all the 11 jointly approximable branches. On a test input image, this fault injection results in an image with no distortion with respect to the reference image. A comparison of the resulting image due to branch faults and the original image is shown in Fig. 5.16. Fig. 5.16a shows the reference image and Fig. 5.16b shows the resulting image with branch fault injections simultaneously in all the jointly approximable branches. This image shows no perceivable distortion. Fig. 5.16c shows the resulting image when faults are injected simultaneously in 146 branches (branches that cause memory error or segmentation fault have been removed) identified to be individually approximable using the hypothesis testing method. This image has a *PSNR* of 17 DB with respect to the reference image and does not meet the user-specified QoS distortion tolerance of $PSNR \geq 20$ DB. The distortion is clearly visible in the image. This establishes the reliability of our Bayesian network based sensitivity analysis.

For joint data sensitivity analysis, constructing the DDG (Data Dependency Graph) is a bottleneck. To address this, we resort to multiple fault injection-based hypothesis testing for joint data sensitivity analysis with a confidence of inference $\theta = 0.6$. The analysis returns the set of data variables which are jointly approximable with a probability ≥ 0.6. For reliability evaluation of our analysis, we perform 1000 data-fault experiments in all

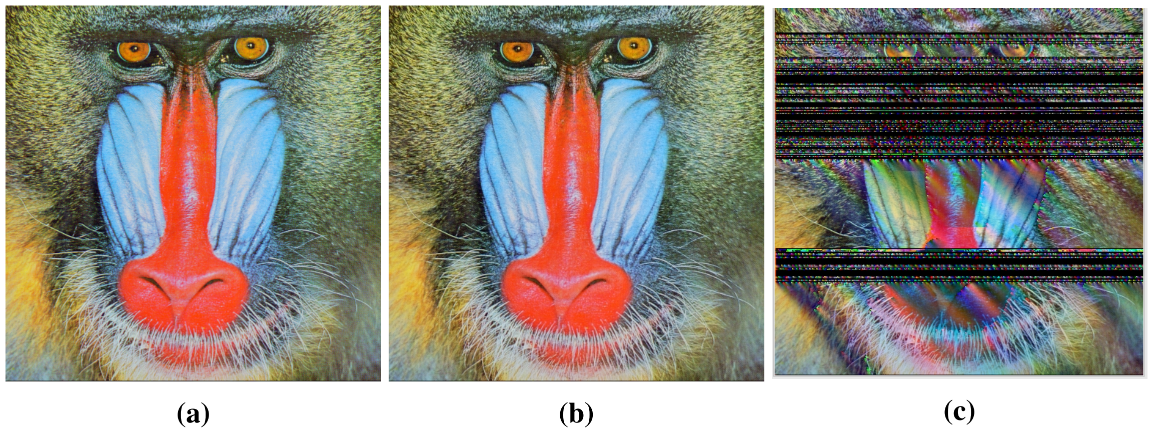


Figure 5.16: QoS comparison with branch faults in approximable branches. (a) Reference image (b) Image with faults simultaneously in all jointly approximable branches given by Bayesian analysis and (c) Image with faults simultaneously in all individually approximable branches given by hypothesis testing

Application	#branches jointly approx.	Passed runs/1000	Pr. jointly approx. by BA
JPEG encoder	11	0.94	0.93
Stream-cluster	12	1	1
JmeInt	9	1	1
Bodytrack	20	1	1
x264	33	1	0.7
Raytracer	5	1	1

Table 5.6: Reliability evaluation of the Bayesian analysis based method

Application	#Loads jointly approx.	Passed runs/1000
JPEG encoder	52	0.69
Stream-cluster	36	0.76
JmeInt	24	1
Bodytrack	27	1
x264	41	0.65
Raytracer	5	1

Table 5.7: Reliability evaluation of the jointly sensitive data analysis with confidence $\theta = 0.6$

the data simultaneously in the set of approximable loads given by the analysis and observe the number of experiments which pass the QoS requirement. The ratio of the number of passed experiments to 1000 gives us an estimate of the probability that the data-set is jointly approximable. The experimental observations are shown in Table 5.7. The first column of the table shows the benchmark application, the second column shows the size of the jointly approximable data set returned from hypothesis testing and the last column shows the empirically computed probability that the data set is jointly approximable by data-fault experiments. We observe that the empirically computed probability in the last column is

≥ 0.6 for all the benchmark applications. This is an expected result because the hypothesis testing with confidence of inference $\theta = 0.6$ should return only that data-set which is jointly approximable with a probability ≥ 0.6.

5.9 Summary

In this chapter, we present a method for sensitivity analysis of data and branches in a program, in order to infer the ones which are approximable. The analysis is based on statistical methods and provides an inference with a probabilistic reliability measure. We provide a method to identify loads / branches in a program using hypothesis testing and show the reliability of our analysis with empirical results. A hardware design of a processor pipeline which can exploit the sensitivity knowledge of instructions in order to implement speculative execution with a selective no rollback policy is presented. Architectural simulations show benefits in term of reduction in the consumed CPU cycles and energy with our speculative execution scheme on a number of applications from approximate computing benchmarks. We believe that our work will have important consequences on the application of approximate computing in modern processors.

Our proposed sensitivity analysis using hypothesis testing and Bayesian analysis are both based on sampling (executing) the application with random inputs chosen by following a uniform distribution. In certain applications, generating such random inputs is hard. For example, in a *Jpeg-encoder* application, we need random images to execute the application and generating such random images is difficult. For such applications, we create a pool of representative inputs and select inputs from the pool uniformly during the sampling process. Considering inputs only from the finite pool may be a cause of inaccuracy in our analysis. We attempt to improve on this process going forward to be able to mark more data and instructions as approximation tolerant.

In this chapter, we have proposed a mechanism to utilize the output of our approximability analysis in a speculative execution environment. While this is of great value to any processor execution environment, an even further optimization is possible by extending the same philosophy to a multi-core execution environment which are common place today, and have more knobs of optimization. This is what we take up in the following chapter. In particular, we assume a multi-core shared memory setup where processors have local private caches, and typically synchronize on a shared cache and primary memory to ensure consistency. A significant number of messages are exchanged to ensure consistency among processor cores for the shared data elements. These messages are overheads that any consistency management protocol induces to ensure coherent and consistent computation. In an exact computation environment, none of these messages can be dispensed with, considering that correctness of the computation may be affected, and is usually not tolerated. However, where approximate computing is the computation model of choice, this leaves us a good room for optimization with a possibility of reduction in the number of messages exchanged, considering the ones that are exchanged to maintain synchrony on the approximable data elements. This is what we leverage in the next chapter, and we show through simulation the significant simulation advantage that our proposal brings in.

Chapter 6

Approximate Computing for Multithreaded Programs in Shared Memory Architectures

In this chapter, we explore the use of our sensitivity analysis techniques in the context of multi-core architectures. Specifically, our goal is in identifying approximable instructions accessing shared data in shared memory multi-core systems. In multi-core and multi-cached architectures, cache coherence is ensured with a coherence protocol. However, the performance benefits of caching diminish due to the cost associated with the protocol implementation. In this chapter, we propose a novel technique to improve the performance of multi-threaded programs running on shared-memory multi-core processors by embracing approximate computing that is built on the foundations of our sensitivity analysis methods.

6.1 Introduction

The performance of applications running on single-core processors automatically improves with newer generations of processors due to increasing clock speed. In multicore processors, however, clever parallel implementation of applications is required to effectively exploit its computational capability. Multithreaded programming has therefore gained importance with the abundance of multi-threaded and multicore processors, and research on multi-core processors, programming languages and runtime executions have been key topics in the research community. Caching in multicore processors mitigates the memory access latency but introduces architectural challenges in ensuring the correct execution of multithreaded programs. A memory location content can be present in multiple caches where it may be updated by the individual cores. This may prevent the cores to see the same global order of all memory updates - a primary condition for correctly executing parallel programs [111]. A multicore processor satisfying this condition is called *sequentially consistent* [111]. Additionally, the updates made on a memory location content present in multiple caches may result in an inconsistent value in the memory if the updates are not communicated appropriately to the sharer caches. This is the *cache coherence* problem. Processors implement a cache coherence protocol (e.g. MESI) to ensure that the caches remain coherent. Implementing such a protocol in the architecture incurs an additional cost and therefore, ensuring cache coherence diminishes some of the performance benefits of caching in multicore processors.

In this chapter, we propose a strategy to improve the performance of multithreaded programs running on shared-memory multicore processors by embracing approximate computing. Our idea is to relax the coherence requirement on memory locations selectively in order to reduce the cost associated with any cache-coherence protocol. In particular, we consider instructions that update shared data and denote them as SWAPs (shared-write-access-points), a name borrowed from [112]. We then perform a systematic sensitivity analysis of the effect of having *coherence faults* at the SWAPs in a number of program executions. A coherence fault refers to the non-communication of the update on a shared data by the SWAP, to the sharer cores in a multi-core processor. Note that write-propagation is an essential requirement for cache-coherence so that all the cores in a processor observe a consistent value in a memory location. Essentially, our analysis observes the resiliency of an application to coherence faults in a SWAP.

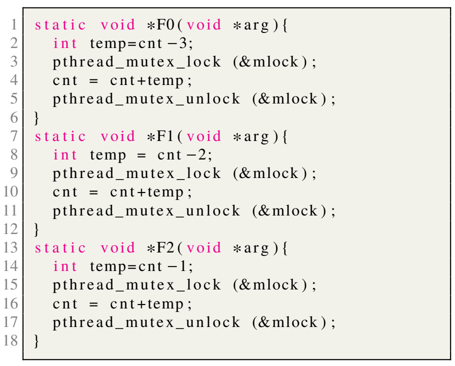
Our contributions in this chapter can be summarized as follows:

- We detect instructions in a multi-threaded program that writes to shared data (SWAPs) and propose a statistical analysis extending our earlier contributions to infer SWAPs which are tolerant to coherence faults. A coherence fault tolerant SWAP is one on which relaxing the coherence requirement in order to reduce the cost associated with a cache-coherence protocol ensures a bounded QoS degradation with a probabilistic reliability guarantee.

- We propose an adapted cache-coherence protocol that relaxes the coherence requirement on stores from approximable SWAPs. Additionally, our protocol uses stale values for cache misses of load data due to coherence. The stale value is the data at the time of cacheline invalidation. We show that our adapted protocol reduces the number of cache-line invalidations during program execution and is therefore efficient.

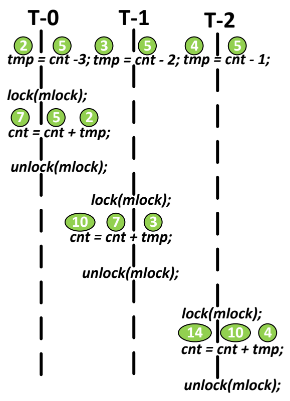
6.2 Motivating Example

The goal of our analysis is to identify write instructions on shared data (SWAPs) for which updates, if not communicated to the other cores, do not cause the computation to deviate from exactness beyond an acceptable limit. The analysis infers a SWAP to be either *sensitive* or *approximable*. By a sensitive SWAP, we mean that the cache-coherence requirement cannot be relaxed on the shared-data updated by the SWAP, if the accuracy of the program is to be kept within acceptable limits. It means that in an invalidation-based cache-coherence protocol, an update by a sensitive SWAP on a shared data should invalidate the cache-lines in the sharer caches containing the same shared data, if the program has to keep its accuracy acceptable. Similarly, in an update-based cache coherence protocol, it would mean that an update on the shared data should be communicated and updated in all other sharer caches for the desired program accuracy. On the contrary, when we infer a SWAP as *approximable*, we mean that the cache-coherence requirement on the shared data that is updated by the SWAP can be relaxed and the program accuracy will remain acceptable with a user-defined probability confidence. We illustrate with our idea with a simple example shown in Example 6.1.

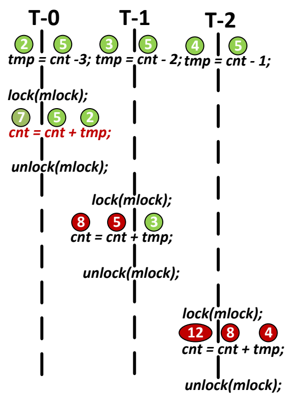


```
static void *F0(void *arg){
  int temp=cnt-3;
  pthread_mutex_lock (&mlock);
  cnt = cnt+temp;
  pthread_mutex_unlock (&mlock);
}
static void *F1(void *arg){
  int temp = cnt-2;
  pthread_mutex_lock (&mlock);
  cnt = cnt+temp;
  pthread_mutex_unlock (&mlock);
}
static void *F2(void *arg){
  int temp=cnt-1;
  pthread_mutex_lock (&mlock);
  cnt = cnt+temp;
  pthread_mutex_unlock (&mlock);
}
```

(a) An example of a multithreaded program



(b) Exact execution



(c) Approximate execution

Figure 6.1: Exact and approximate execution with a given interleaving.

Example 6.1 We illustrate our idea with a simple multithreaded program running on 3 cores as shown in Figure 6.1. Let us assume that the functions $F0$, $F1$ and $F2$ are executed by threads T-0, T-1 and T-2 respectively. We assume that thread T-0, T-1, and T-2 execute on Core-0, Core-1, and Core-2 respectively. The program contains two variables of interest, the shared variable *cnt* and the thread local variable *temp*. We ignore the *mlock* variable for the time being.

Figure 6.1b shows the exact execution of the program under a thread scheduling policy (the execution sequence of the instructions is from top to bottom). In this execution, we make an assumption that a copy of the memory location of *cnt* is present in the local caches of the three cores. The Green circles 🟢 associated with the program variables shown above the program statements depict the value of the variable after executing the respective statement. For example, after executing the statement $tmp = cnt - 3$ in thread T-0, the values of *tmp* and *cnt* are 2 and 5 respectively, shown in Green circles 🟢 above the variables. On completion of the program, the value of *cnt* is 14.

Consider the statement $cnt = cnt + tmp$ in thread T-0 that writes to the shared variable. This is a Shared Write Access Point (SWAP) in our terminology. Figure 6.1c shows the approximate computation of the program, under the same thread scheduling (top to bottom) as in the exact computation. For the sake of illustration, we assume in this approximate execution that any instruction colored Red in thread T-0 is an approximable SWAP.

On executing this SWAP, the value of *cnt* is updated to 7 in the local cache of Core-0, however, the copies of *cnt* in the other caches are not updated / invalidated. Therefore, Thread T-1 will be using the stale value of *cnt*, i.e., 5 instead of 7 and execute the statement $cnt = cnt + temp$. Since this statement in T-1 is not an approximable SWAP, the new value 8 of *cnt* is updated in the other caches. Now, the last update statement of T-2 is executed by Core-2 with the new value of *cnt* from its local cache, resulting in the final value of *cnt* to be 12. Observe that the result of our approximate computing differs from the exact computation by a value of 2. If this error in computation is acceptable to the user, we have an approximate program execution that avoids some of the overhead for cache-coherence and can still produce an acceptable result. This is the main idea of our work, as explained in the following sections. □

6.3 Detailed Methodology

Our proposed methodology consists of three main steps:

- The first step is a dynamic *thread sharing analysis* in order to identify SWAPs in a multithreaded program [113], [114].
- The second step is for the *sensitivity analysis* of applications in the presence of coherence faults in program SWAPs. A statistical analysis [115] is then carried out on the computational results to determine the sensitivity of SWAPs. The analysis injects coherence faults at a SWAP under test and its effect on the application output is evaluated on a number of executions with varying inputs. The result of the analysis is the inference of SWAPs as either *approximable* or *sensitive*. The analysis also provides

a probabilistic measure of confidence by which we can ascertain that the program output will remain within a user-specified acceptable QoS range in the presence of coherence faults on approximable SWAPs. Our experiments show the presence of approximable SWAPs in considerable numbers in applications from the approximate computing domain, which are resilient to limited computational errors.

- The third and final step is to replace the SWAPs in the machine code of a program that is found approximable, with an approximable counterpart. An extended Instruction Set Architecture for the processor is proposed to represent approximable SWAP instructions in a program. At runtime, the coherence requirement is then relaxed for data of the approximable SWAP instructions. We propose an adaptation of the MESI cache-coherence protocol [116] to benefit from the sensitivity knowledge obtained from sensitivity analysis. The modified protocol attempts to reduce the number of cache-line invalidations and coherence message exchanges between the cores during an execution of a multithreaded program. Additionally, our protocol uses stale values for load misses due to coherence, the stale value being the version at the time of invalidation, by having them in an auxiliary cache.

In the following sections, we describe each of the steps in more detail. However, before we describe our overall methodology for approximate execution in concurrent programs, we briefly discuss a directory-based cache coherence protocol in order to address the cache coherence problem in multicore processors, and forms the basis of our work.

6.3.1 Cache Coherence for Multicore Processors

Shared memory multicore processors typically have private caches for each core and a shared main memory in the last level of the hierarchy [116]. In such a memory organization, it is possible to have a copy of some data in the main memory to exist simultaneously in multiple private caches. This poses particular challenges since any core's local update on the data shared in multiple caches needs to be communicated to the sharers in order to keep the data consistent. A cache coherence protocol ensures this consistency requirement and is implemented in the architecture. Our proposal in this work is based on a directory-based cache coherence protocol, where private caches send memory read / write requests to a central memory controller called the *directory*. The directory maintains the information of which data is present in which core's private cache and acts as a mediator for all memory operations. We consider a write-back, invalidation based cache-coherence protocol. In a write-back cache, a write-request hit is updated to the (local) cache but the same update to the memory waits until the eviction of the block from the cache [117]. We now discuss the MESI cache-coherence protocol where each cache-block is associated with a state.

6.3.2 MESI Cache Coherence Protocol

The MESI Cache Coherence Protocol is a common invalidation-based protocol that supports write-back caches. The state of a cache-block can be one of M, E, S or I. The semantics associated with these states are:

- **Modified (M)** : The block is valid [1], owned [2], and potentially dirty [3]. The cache has the only valid copy of the block and must respond to requests for the block.
- **Exclusive (E)**: The block is valid and clean [4]. The cache has the only read-only copy of the block.
- **Shared (S)** : The block is read-only, valid and clean, present in more than one cache.
- **Invalid (I)**: The block is invalid and ready for eviction.

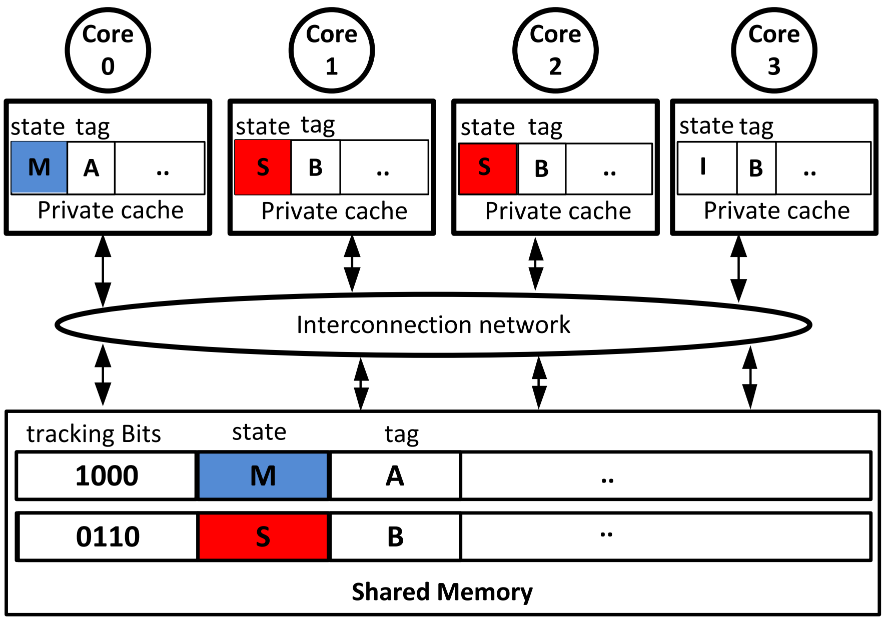


Figure 6.2: A 4-core shared memory architecture. The directory is located in the shared memory. Each block in the memory is associated with a coherence state and presence bits.

Example 6.2 Figure 6.2 explains a simple directory-based protocol on a 4-core processor with a shared L2 cache. Each core owns a private cache and share a common memory via the interconnection network. The directory is present in the shared memory. This type of architecture is typically implemented in modern processors such as in Intel Core-i7 [118], [119]. The directory keeps presence bits, one bit per core, for each cache-block in order to

[1] A valid block contains an updated data

[2] The owner core of the block is responsible for responding to load/store requests for that block from other cores

[3] A block is dirty if its data is updated but differs from the stale value in the memory

[4] A clean block contains the same data as in the memory

mark the presence / absence status of the cache block in the respective core's private cache. Together with the presence bits, each cache-block in the directory also maintains a state. As depicted in the figure, the block A is cached by core 0 only and is in the state M (Modified). The corresponding presence bits in the directory store *1000* to show that block A is present in the cache of core-0 and absent in the other core caches. Here, *1* depicts the presence of the cache-block in the private cache of a core. Similarly, the Block B is cached by both core 1 and core 2 and is in the Shared (S) state. The corresponding presence bits in the directory is given as *0110*. When a core issues a load or store request that misses in its private cache, it issues a request message to the directory. Depending on the block's coherence state and the presence bits, the directory either responds directly with the data or forwards the request to one or more cores that ought to respond to the request. □

State Diagram

Figure 6.3 shows a description of the MESI directory protocol via state-diagrams borrowed from [117]. The state diagram has three types of transition messages, namely request messages, forwarded-request messages and response messages.

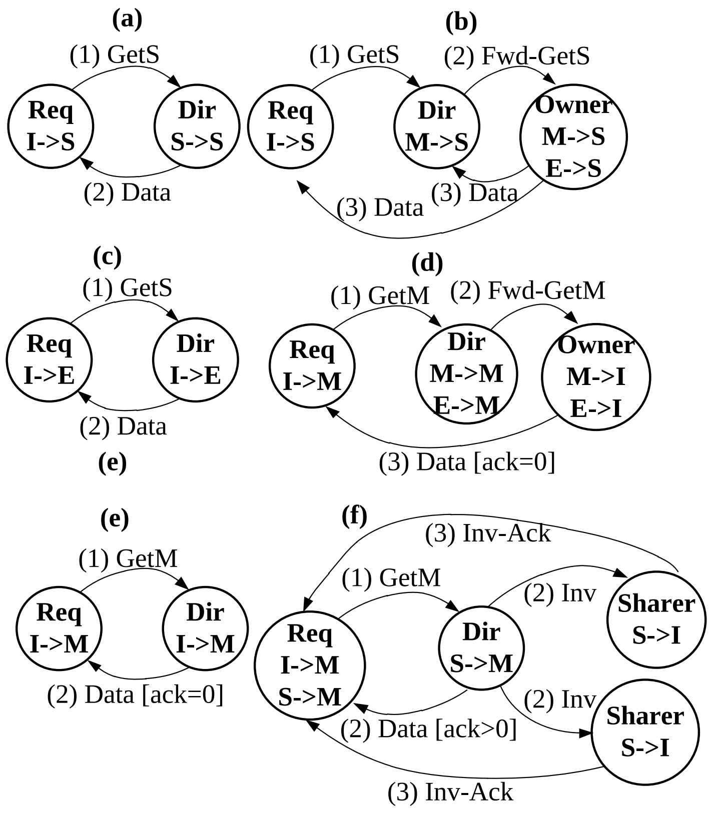


Figure 6.3: State-transitions in a MESI protocol.

The request messages are as follows :

- **GetS**: Obtains block in read-only mode.

- **GetM**: Obtains block in read-write mode.

The forwarded-request messages are :

- **Fwd-GetS**: Forwards **GetS** request to owner.
- **Fwd-GetM**: Forwards **GetM** request to owner.
- **Inv**(alidation): To change a block to invalid state.

and the response messages are:

- **Data**: The updated data of a cache block, and
- **Inv-Ack**: An acknowledgment of block invalidation in a cache.

In Figure 6.3, the cache controller that requests a memory transaction is denoted by **Req** and the directory controller is denoted by **Dir**. The owner of the block is denoted by **Owner** and the sharers of the block are denoted by **Sharer**.

A brief summary of the transitions in the state-diagram is discussed below.

- Figure 6.3a: A requester core sends a **GetS** request for a cache-block to the directory. If the state of the block in the directory is shared (S), then the requester obtains the block in its cache from the directory (shown with the transition labelled with the response message - Data) and marks it in the Shared (S) state locally.
- Figure 6.3b: A requester core sends a **GetS** request for a cache-block to the directory. If the state of the block in the directory is modified (M), then the directory controller sends a **Fwd-GetS** request to the owner. The owner upon receiving the **Fwd-GetS** request, sends the updated block-data to the directory as well as to the requester core. The directory upon receiving the updated block-data, changes the block state to shared (S). The requester also changes the state of the block to shared (S) upon receiving the data from the owner.
- Figure 6.3c: A requester core sends a **GetS** request to the directory. If the directory state is Invalid (I), the directory sends a request to the memory to fetch the line and updates the state in the directory to Exclusive (E). Data is then forwarded to the requester core and the line is updated to Exclusive (E).
- Figure 6.3d: A requester core sends a **GetM** request to the directory. If the state of the requested block in the directory is Exclusive (E) or Modified (M), a forward request message **Fwd-GetM** is sent by the directory to the owner. This changes the block state to modified (M). The owner then sends the block to the requester and invalidates (I) its own copy.

- Figure 6.3e: A requester core sends a **GetM** request to the directory. If the state of the requested block in the directory is Invalid (I), the directory sends a request to memory to fetch the line and updates the state in the directory to Modified (M). Data is then forwarded to the requesting core and the line is updated to Modified (M).

- Figure 6.3f: A requester core sends a **GetM** request to the directory for a block that is either in Invalid (I) or Shared (S) state. The directory has the block in shared (S) state. The directory then forwards an invalidation request (Inv) to all the sharing cores having the requested block in their private cache. As a response, the directory sends the Data and the count of the sharers to the requester. The sharers invalidate the cache-block copy upon receiving the Inv message from the directory and sends an Inv-ack response message to the requester core. The requester counts the number of received Inv-acks and transitions the block to modified (M) state when Inv-acks from all the sharers are received.

Example 6.3 To illustrate the working principle of the MESI protocol, let us consider a simple 3-cores processor system, each having its own private cache as shown in Figure 6.4. The coherence protocol is implemented using a directory-based cache coherence protocol. Consider 3 instructions that are executed by the cores. On executing each instruction, a series of actions are required to complete the transaction. The notation for the action executed by each core are as follows:

- Black circle denoted by ● are actions by Core-1.
- Yellow circle denoted by 🟡 are actions by Core-2.
- Red circle denoted by 🔴 are actions by Core-3.

Consider the following ordered actions by Core-1:

- ❶ **Rd X** issues a read request to its own local cache. The request is a miss since the line containing the data **X** is not in the cache.
- ❷ **GetS** is a request message to the directory for the line containing the data **X**.
- The directory at present does not have the line and it needs to be fetched from the memory. ❸ is a **GetS** request and ❹ is a data reply from memory.
- ❺ the directory adds an entry of the requested line and marks the state as **E** and ❻ returns the data to the requester core i.e., Core-1.
- ❼ The requested line is then added to the local cache of Core-1 with the state **E**.

Similarly, consider the following ordered actions by Core-2:

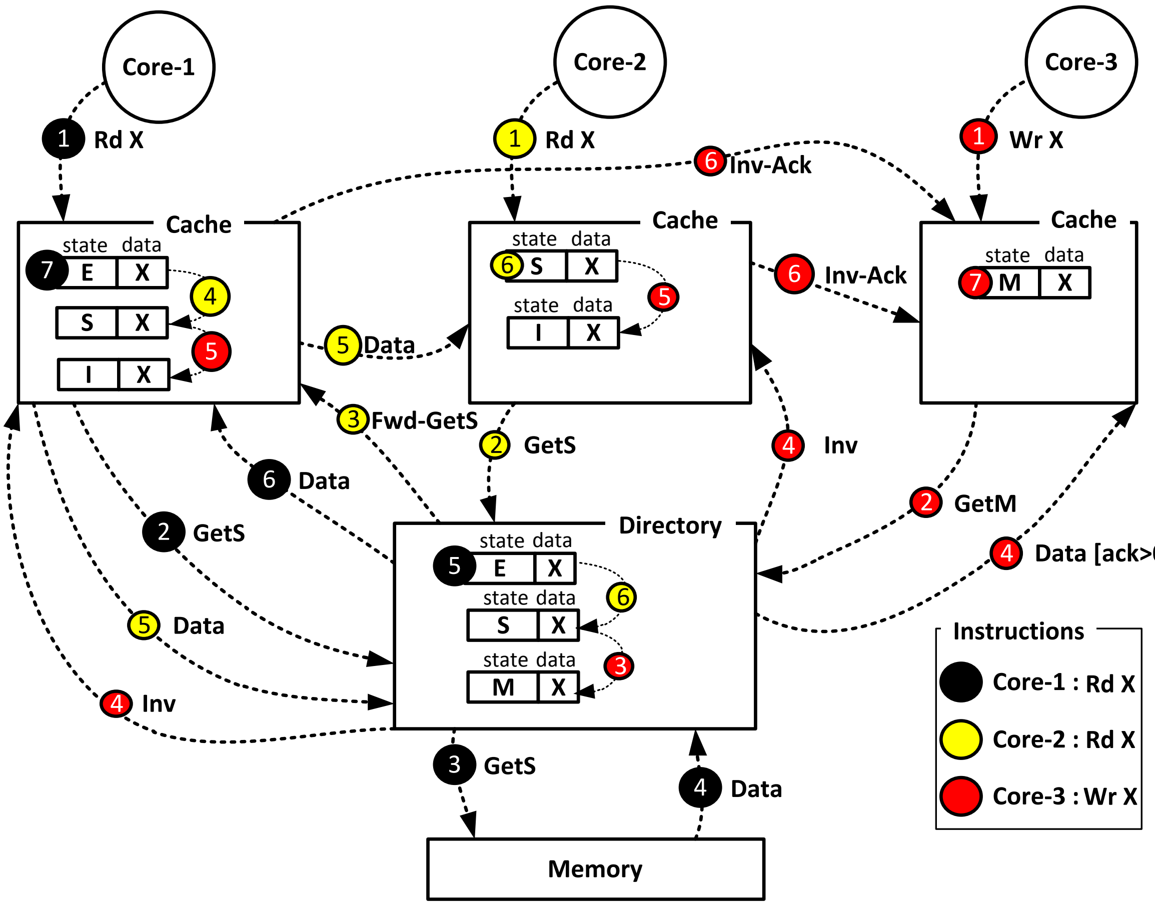


Figure 6.4: A simple 3-cores processor each with its own local caches managed by a directory-based MESI cache coherence protocol

- (1) **Rd X** issues a read request to its own local cache. The request is a miss since the line containing the data **X** is not in the cache.
- (2) **GetS** is a request message to the directory for the line containing the data **X**.
- (3) **Fwd-GetS** is a forward message by the directory to the owner core, i.e., Core-1. On receiving the request, (4) Core-1 updates the line state to **S**. The line is then forwarded to both Core-2 ((5) **Data**) and Directory ((5) **Data**).
- Finally, (6) updates the line state to **S**.

Finally, consider the following ordered actions by Core-2:

- (1) **Wr X** issues a write request, the request is a write miss in the local cache of Core-3.
- (2) **GetM** is a read/write request to the directory.
- (3) The directory changes the state of the requested line to **M** and sends an invalidation message ((4) **Inv**) to Core-1 and Core-2.

- Core-1 and Core-2 upon receiving the invalidation request, invalidates their own copies (5) and each core sends an invalidation acknowledgment (6 **Inv-Ack**) to Core-3. □

In this work, we consider the directory based MESI cache-coherence protocol and propose modifications to the protocol to incorporate the philosophy of approximate computing. In a nutshell, with feedback from the sensitivity analysis step, for SWAPs that are deemed to be approximable, we propose to drop the coherence messages to cores.

6.3.3 Our Approach

Figure 6.5 shows a block diagram of our methodology. It consists of two main phases, the *pre-execution analysis* and the *execution (architectural simulation)* phase. The pre-execution phase is performed one time, which transforms the application to its approximate version. The execution phase executes the approximate version in an architecture which supports the relaxed cache coherence protocol. In the following, we discuss these two phases in detail.

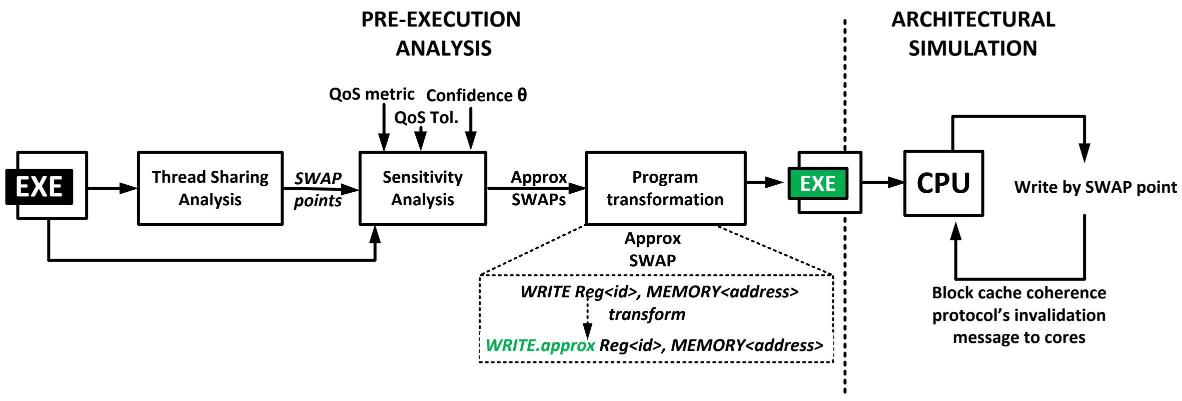


Figure 6.5: Block-diagram of our pre-execution analysis by sensitivity analysis and ISA extension.

Pre-execution Analysis

The pre-execution analysis phase consists of the following three main steps:

- The first step is to identify SWAPs in the program using a technique called Thread Sharing Analysis (TSA) [112].

- The second step is the *sensitivity analysis* of SWAPs. The analysis studies the fault tolerance of a multi-threaded program when *coherence faults* are injected in a SWAP. Our idea is to systematically study the deviation of the computational result from the expected correct one, in the presence of these faults. In particular, we propose statistical hypothesis testing to identify SWAPs where fault injections do not cause an unacceptable deviation from the expected result. We call such SWAPs *approximable*.

- The third step is a *program transformation* where store / write instructions corresponding to approximable SWAPs are replaced with their approximate counterparts.

Thread Sharing Analysis

The primary step in our analysis is to detect SWAPs in a multi-threaded program. For this, we use Dynamic Thread Sharing Analysis (TSA) proposed in [112]. The analysis is performed at runtime for a representative input set. For each program location (write instructions in our case), the analysis algorithm tracks the memory location that the executing thread accesses. If the same location is accessed by two different threads from two different program locations or twice from the same program location, with at least one for writing, the location is marked as shared. Consequently, all write instructions with a write-access on shared data are classified as SWAPs.

Algorithm 6.1 Dynamic Thread Sharing Analysis

```
1: $MemSet \leftarrow \emptyset$
2: $MemSet \leftarrow MemSet \cup BSS \cup DATA$ // .bss and .data sections
3: function OnMemoryAllocation($start$, $end$)
4:     $memArea \leftarrow MemArea(start, end)$
5:     $MemSet \leftarrow MemSet \cup memArea$
6: end function
7: function OnMemoryAccess($addr$, $thread\_id$, $inst$)
8:     memArea ← getMemArea(addr)
9:     if isRead(inst) then
10:        memArea.RdThread(thread_id)
11:        memArea.RdInst(ins)
12:    end if
13:    if isWrite(inst) then
14:        memArea.WrThread(thread_id)
15:        memArea.WrInst(inst)
16:    end if
17: end function
18: function onExecutionEnd( )
19:    $\mathcal{S} \leftarrow \emptyset$ // Empty SWAPs Set
20:    for all $M \in MemSet$ do
21:        if isShared(M) then
22:            $\mathcal{S} \leftarrow \mathcal{S} \cup M.getWrInst()$
23:        end if
24:    end for
25: end function
```

Algorithm 6.1 presents an overview of Dynamic Thread Sharing Analysis. Our analysis is designed using dynamic binary instrumentation and hence relies on events triggered by a running program. On such events, the analysis routines capture useful information such as *instruction*, *thread ID*, *memory address* and other architectural states of the running application. All routines except isShared(), which is a helper routine are callback routines [5]. The algorithm first initializes the $MemSet$ to an empty set as shown in Line-1. $MemSet$ is a set that stores memory allocation information. Static regions such as block-started-by-symbol (*bss*) and *data* section are added to $MemSet$ as shown in Line-2. In the following we discuss each routine in detail:

OnMemoryAllocation() : The routine triggered when a dynamic memory allocation is invoked such as *malloc()*, *calloc()*, *realloc()* etc. This routine captures the memory area and then inserts it to $MemSet$.

OnMemoryAccess() : The routine is triggered when a memory read / write is invoked. This routine records the read/write into the memory areas.

onExecutionEnd() : This routine is triggered when a program is about to exit. The job of this routine is to extract SWAPs from the memory areas.

isShared() : This is a helper routine. It checks if the memory area is a shared memory allocation.

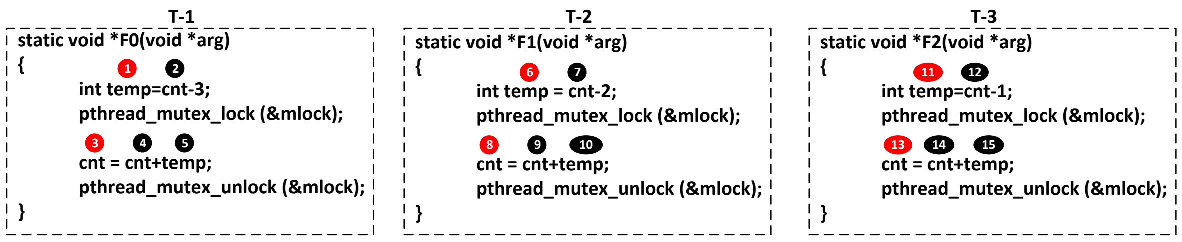

T-1
```
static void *F0(void *arg)
{
        int temp=cnt-3;
        pthread_mutex_lock (&mlock);
        cnt = cnt+temp;
        pthread_mutex_unlock (&mlock);
}
```
T-2
```
static void *F1(void *arg)
{
        int temp = cnt-2;
        pthread_mutex_lock (&mlock);
        cnt = cnt+temp;
        pthread_mutex_unlock (&mlock);
}
```
T-3
```
static void *F2(void *arg)
{
        int temp=cnt-1;
        pthread_mutex_lock (&mlock);
        cnt = cnt+temp;
        pthread_mutex_unlock (&mlock);
}
```

Figure 6.6: A multi-threaded program with 3 threads accessing variables temp and cnt. Reads are shown in Black and writes are shown in Red.

Example 6.4 Figure 6.6 shows a simple hypothetical multi-threaded program for illustrating the thread sharing analysis problem. The program contains three threads executing functions F0, F1 and F2. These threads access a shared variable *cnt* and their own thread local variable *temp*. There are in total 15 read / write operations on these variables, marked with numbers from **1** to **15**. The goal of the thread sharing analysis is to find out which of these 15 accesses are to thread-shared data. In the above example 3, 8 and 13 are writes to the shared data *cnt* and 2, 4, 7, 9, 12, and 14 are reads on the shared data *cnt*. The thread sharing analysis identifies 3, 8 and 13 as Shared Write Access Points (SWAP) □

[5]Routine invoked when certain event happens

Sensitivity Analysis

Our analysis is based on framing a statistical hypothesis and then testing the hypothesis with a statistical hypothesis testing procedure like in the previous chapters, except that we now carry out the analysis on shared variable writes in concurrent programs. We elaborate the steps in the following:

Fault Injection Experiment

The analysis performs fault injection experiments on a SWAP, these faults are called coherence faults. The definition of a coherence fault is as follows:

Definition 6.1 *Let $\mathcal{P}$ be a multi-threaded program and* s *be a SWAP in $\mathcal{P}$. s writes on a shared data x, shared between k threads scheduled to execute on k cores of the processor. Let e be the execution of the program $\mathcal{P}$ on an input I. A fault injection experiment at the SWAP s is the resulting execution e' with the same input I when the write of data v into x by s is not communicated to the sharers of x. We refer to the non-communication of a write by a SWAP to the sharers as a coherence fault.* ■

Coherence faults are usually expected to lead to erroneous execution due to incoherence of the data on the shared variable x, across the caches of cores. Our goal of this fault experiment by inserting a coherence fault is to observe the resiliency of the multithreaded program to a relaxed coherence requirement. For example, if we observe that the quantitative difference between the result of the correct execution e and the faulty execution e' for the same input is within the acceptable tolerance, then the fault experiment provides us with an evidence that the shared write access point s is indeed tolerant to coherence faults. We now discuss how we collect observations from many fault experiments to statistically infer the fault tolerant SWAPs with a probabilistic confidence.

To emulate coherence faults during an execution, we associate a buffer to each thread during execution as shown in Figure 6.7. The buffer stores values that are not propagated to other threads. Our coherence fault model works as follows:

- We denote the SWAP that is being analyzed for sensitivity as the *target SWAP*. The other SWAPs in the program are referred as *non-target SWAPS*. Given a target SWAP that writes to a shared memory location / data variable **X**, for each thread executing the target SWAP, the value of X is written to a thread local buffer.
- For all subsequent reads of **X**, the value of **X** is fetched from the local cache until it is written by non-target SWAPs.

- When non-target SWAPs write to **X**, it is written to memory and the content of **X** from all buffers are removed.

The following example explaining an instance of our coherence fault model.

Example 6.5 Consider the simple example shown in Figure 6.7. We assume two threads **T0** and **T1** executing on Core-1 and Core-2 with the order of execution of the instructions shown from 1 to 8. Consider an instruction 2 **Wr X** executed by thread **T0** that writes the value of **X** as the target SWAP for analysis. On writes by a target SWAP of a thread, our fault-injection model blocks the writes to memory but instead writes to a local buffer of a thread, in order to emulate a non-communication of the write to other sharer caches. When the same thread reads the data, it is then fetched from the local buffer. When other SWAPs write to **X**, it is propagated to all threads. In the following, we will walk through a simple example as shown in the figure:

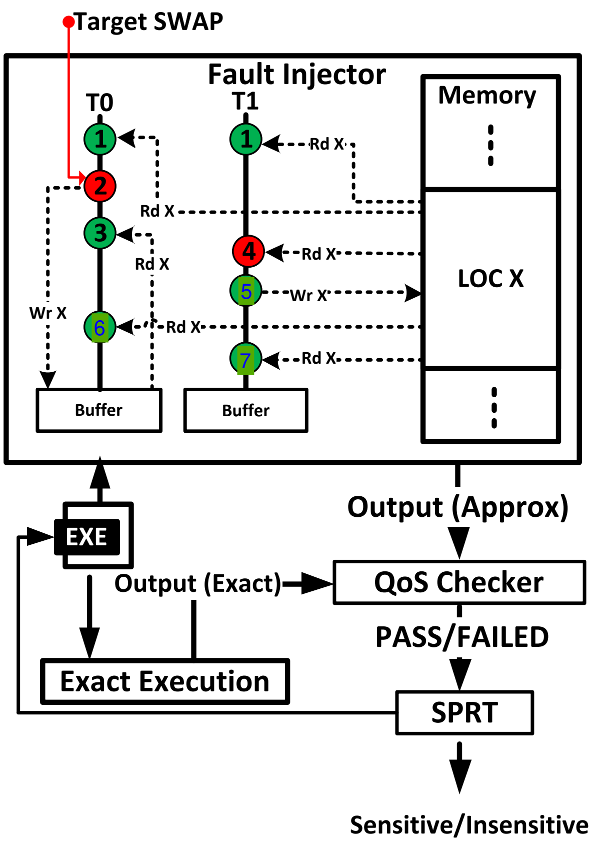


Figure 6.7: A demonstration of coherence fault model

- 1 Thread **T0** and **T1** read X.
- 2 Thread T0 issues a write (**Wr X**). Since this is a target SWAP, it is written in the thread local buffer.
- 3 Thread **T0** reads X. The value of **X** is fetched from the local buffer.

- 🟥4 Thread T1 reads X. The value of X is fetched from memory (stale value).
- 🟩5 Thread T1 writes X. The value of X is written to memory and all buffers having the value of X are cleared.
- 🟩6 Thread T0 reads X. The value of X is fetched from memory (correct value).
- 🟩7 Finally, thread T1 reads X from memory.

It may be noted that the thread T1 executes with the stale value at 🟥4 **Rd X**. □

Hypothesis Testing

This is similar to the one proposed in Chapter 3, but here, we consider hypothesis of the form *"A SWAP in a multi-threaded program is approximable"*. The definition of an approximable SWAP is as given below.

Definition 6.2 *Given a confidence of inference θ and an acceptable QoS distortion measure α, a SWAP s in a multithreaded program $\mathcal{P}$ is said to be approximable if and only if for all executions e of $\mathcal{P}$, the probability that the program output remains within the acceptable QoS distortion α in the presence of a coherence fault in s, is at least θ.* ■

Our methodology then proceeds with testing the hypotheses for the SWAPs of interest, with a hypothesis testing procedure. We frame the hypothesis in the same way as discussed in Chapter 3 Section 3.3.3. The Sequential Probability Ratio Testing procedure is used to test an hypothesis as discussed in Chapter 3 Section 3.3.4.

Program Transformation

The final step is a program transformation by replacing the $\langle write \rangle$ instructions corresponding to approximable SWAPs by an approximate counterpart $\langle write.approx \rangle$. This extended ISA allows for selectively switching to an approximate implementation of the cache-coherence protocol that we discuss in the following section.

6.4 Relaxed Cache-Coherence Protocol

We propose a modified cache coherence protocol to exploit the sensitivity information from our analysis for performance. We now elaborate the details below.

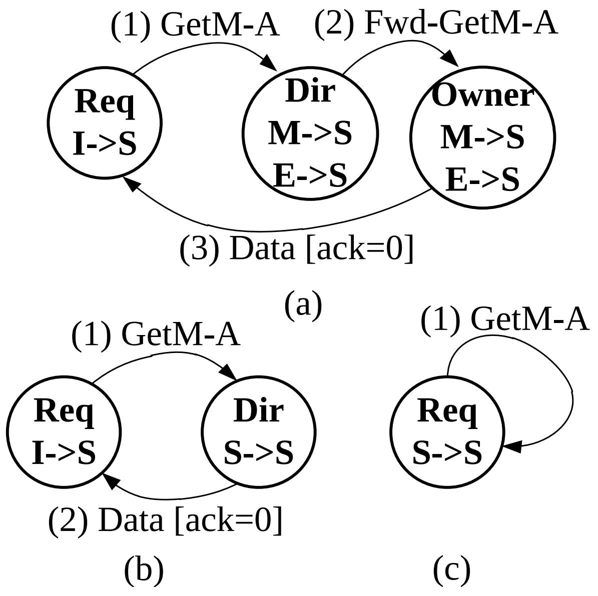


Figure 6.8: Modified MESI protocol

Modified Cache Coherence Protocol

We now describe the modifications on the MESI directory-based cache coherence protocol that we carried out in order to relax the coherence requirement selectively with an objective of mitigating the coherence overhead in multi-threaded applications. The proposed modifications in the state-diagram of the MESI directory-based cache coherence protocol discussed in Section 6.3.1 are shown in Figure 6.8. We emphasize in Figure 6.8a that approximable write-requests (GetM-A) by requester cores on cache-blocks, which are either in modified (M) or in exclusive (E) state in the directory, are forwarded to the block owner to be sent to the requester. Instead of the owner invalidating their own copy and then sending the data to the requester, the owner now transitions to the shared (S) state and sends the data to the requester. Essentially, the sharers continue their computation with a stale copy of the cache-block, not updated with the latest write by the requester for blocks for which our sensitivity analysis ensures that it is computationally acceptable to do so. Figure 6.8b shows that if a core puts an approximable write-request on a cache-block to the directory, and the directory finds that the cache-block is shared between cores, then instead of the sharers invalidating their local copies, the modified protocol allows the sharers to keep their stale local copies. In this way, many cache invalidates and invalidation message exchanges are saved. In Figure 6.8c, it is shown that in our modified protocol, approximable writes are not communicated to the sharers and the sensitivity analysis ensures that the loss of accuracy in computation will be within acceptable limits. Therefore, many coherence messages are avoided with our adapted protocol.

Example 6.6 To illustrate our modified MESI protocol, we consider an example shown in

Figure 6.9. This is similar to Example 6.3, except that the last instruction is an approximate write request by Core-3. The first and second instructions by Core-1 and Core-2 have the same actions as in Example 6.3. The SWAP instruction **Wr_approx X** by Core 3 marked in ●, is found to be approximable. The subsequent actions are as follows:

- 1 **Wr_approx X** is an approximate write request by Core-3.
- 2 **GetM-A** is an approximate request message to the directory requesting the line containing **X**.
- The directory returns the line (3 Data[ack=0] [6]) and this is inserted in the local cache 4.

It can be observed that some of the messages and cache evictions are avoided. □

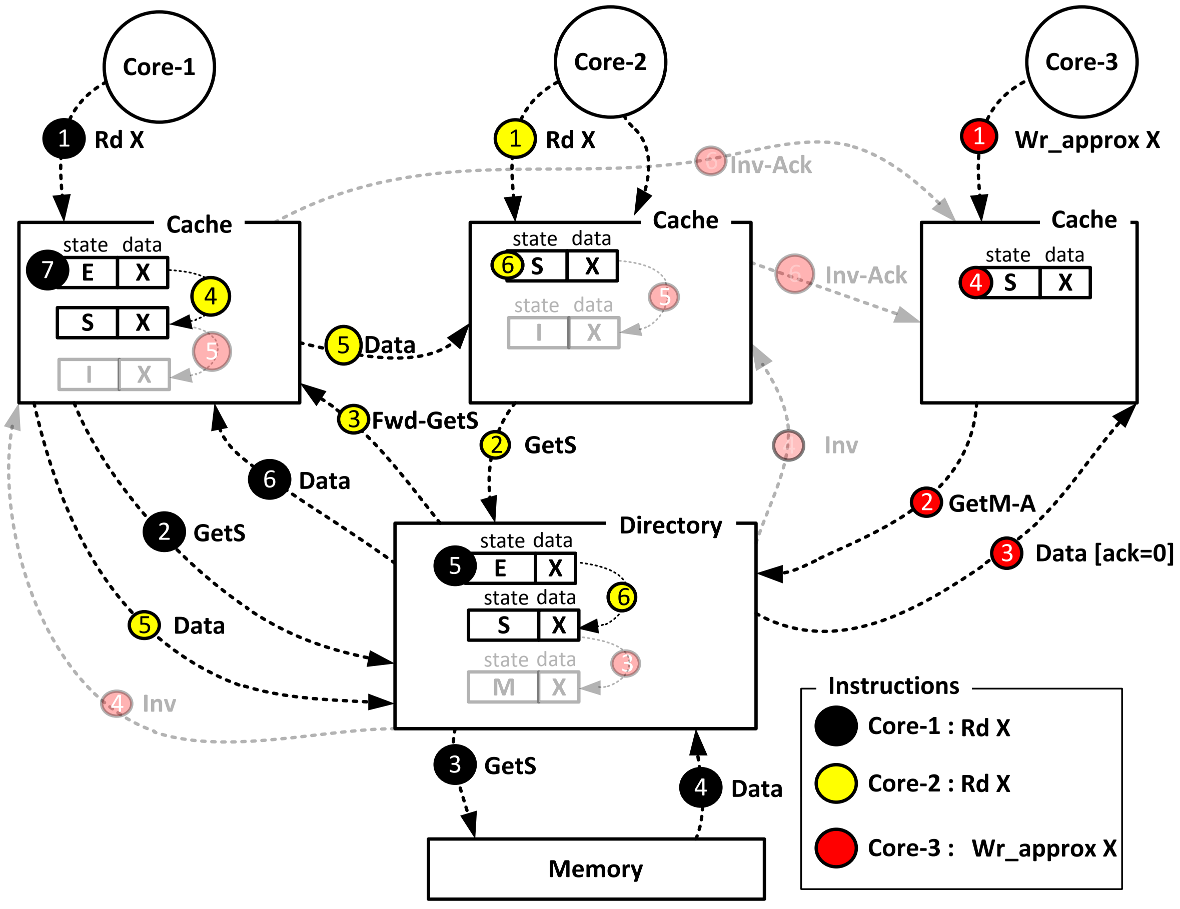


Figure 6.9: A simple 3-core processor managed by a directory-based MESI cache coherence protocol. The protocol behaviour for an approximable-write instruction is depicted here.

[6]No acknowledgment

Exploiting Staleness

In addition to the modified coherence protocol, we adopt the idea of using stale data for coherence load misses, as proposed in [65]. This work proposes to approximate coherence-related load misses by returning stale values, i.e., the version at the time of invalidation. The authors propose a small Stale Victim Cache (SVC) with only 8 lines to hold invalidated lines upon invalidation in the L1 data (L1-D) cache. To avoid the possibility of data getting very stale, they further propose to time-out these lines from SVC. Using this technique, they report an average speedup of 10-15% on the SPLASH-2 applications with the maximum error of at most 0.08% across the entire suite. We adopt the idea of using SVC but without any time-out.

Stale Victim Cache (SVC)

SVC is a small cache that holds only those invalidated lines that are evicted due to coherence actions from L1-D. The following three actions are carried out in the SVC.

- *Insertion* : When a cache line is invalidated due to coherence conflicts by the cache replacement policy, it is inserted into the SVC.
- *Lookup* : When a core issues a load request to the L1-D cache, SVC is looked up concurrently. If the line exists in the cache (valid or stale), the data is returned to the core. Otherwise, if the line is present in the SVC, the data from SVC is returned to the core. A request is then sent to fetch the updated data using the usual process.
- *Eviction*: An element in the SVC can get evicted in two ways: (a) When the data is brought in from the memory hierarchy, the line is updated in the L1-D cache (in valid state) and it is then removed from SVC, or (b) due to the replacement policy.

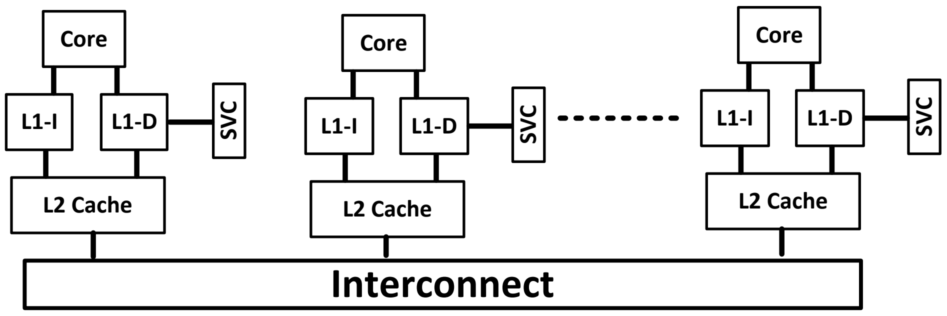


Figure 6.10: Architectural configuration with SVC attached to L1-D in each core

Figure 6.10 shows the architectural configuration of the system with the SVC attached to the L1-D cache on each core. The L2 cache is shared across all the cores. In the event of a load miss in the L1-D cache, if the requested cache-line is present in invalid state, then the stale data is given to the requester core. Concurrently, a memory request is sent to the next level of the memory hierarchy. However, the requested core continues the execution with the stale value and as a result does not incur any miss penalty. The size of SVC is 256KB, having 8 lines of 32 KB each and uses the Least-Recently-Used (LRU) replacement policy. We consider this cache to have similar delay as that of the L1 cache.

6.5 Evaluation

We perform our proposed pre-execution analysis on 9 multithreaded applications from the SPLASH 3.0 multi-threaded benchmarks suite [120]. We first describe briefly the applications followed by the Quality of Service (QoS) metric and tolerance used in the evaluation.

6.5.1 Applications for Evaluation

- **Fast Multipole Method (FMM)** simulates interactions of a system of bodies in two dimensions over a number of time-steps.
- **Ocean** is an application from the domain of oceanography. It studies the role of eddy and boundary currents in influencing large-scale ocean movements [121].
- **Raytracer** is an image processing application that renders a three-dimensional scene using the ray tracing algorithm [122].
- **Cholesky** factors a sparse matrix into the product of a lower triangular matrix and its transpose [122].
- **FFT**, **LU** kernels and **Raytracer** are parallel versions of applications mentioned in Chapter 3 Section 3.5.2.

6.5.2 QoS Metrics

To evaluate the performance and accuracy of our approach with respect to an exact computation, we adopt the QoS metrics as discussed in Chapter 3 Section 3.5.2. We compare the outputs produced out of each benchmark application with respect to one or more of the QoS metrics, that is standard in the approximate computing community.

6.5.3 Evaluation of Sensitivity Analysis

In our evaluation for sensitivity analysis, we only consider those SWAPs that have at least 10 writes. Table 6.1 shows our observations. The confidence probability factor θ of hypothesis testing taken for our analysis is 0.5. In the table, the 1^{st} column shows the application, the 2^{nd} column shows the Lines of Code (LoC), the 3^{rd} column shows the QoS metric, the 4^{th} column shows the QoS tolerance in terms of the respective QoS metric, the 5^{th} column shows the number of SWAPs tested, the 6^{th} column shows the number of approximable SWAPs classified by our analysis, the 7^{th} column shows the percentage of approximable SWAPs classified by our analysis and the 8^{th} column shows the time taken by our analysis to complete the classification. The minimum and maximum number of SWAPs taken for analysis are 190 (*Cholesky*) and 27 (*LU-C*) respectively. The lowest and highest percentages of approximable SWAPs are 41% (*Ocean-C*) and 79% (*FFT*) respectively. The analysis took the longest time of 50 hours to complete on *FMM*. The best time to complete the analysis is 30 minutes for *LU-NC*. The analysis time depends on the application's execution time, the number of samples tested by SPRT, probability factor θ and the inputs considered for sample tests in SPRT. We observe that an average of 57% of the tested SWAPs are approximable in the considered applications from the SPLASH 3.0 applications, showing their tolerance to coherence faults.

Application	LoC	QoS Metric	QoS Tol.	SWAPs Tested	SWAPs Appr.	% Appr.	Time (hrs)
FMM	2945	Average Relative Error	≤ 0.1	144	9	6.25	50
OCEAN NC	2731	Average Relative Error	≤ 0.1	92	63	68.3	15.6
OCEAN C	4283	Average Relative Error	≤ 0.1	88	36	40.9	16.6
RAYTRACER	5783	PSNR	≥ 20	156	98	62.8	17
CHOLESKY	3847	Percent Error	≤ 0.1	190	85	44.7	8.10
FFT	668	Normalized Mean Error	≤ 0.1	24	19	79.1	0.82
LU NC	494	Normalized Mean Error	≤ 0.1	28	20	71.4	0.51
LU C	916	Normalized Mean Error	≤ 0.1	27	20	74	0.52
RADIX	654	matching	0	31	21	67.7	0.8

Appr.:Approximable; %Appr. : Percentage of the tested SWAPs inferred approximable; LoC: Lines of Code; QoS: Quality of Service; Tol. : Tolerance

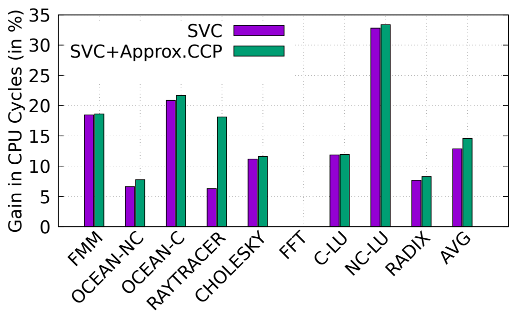
Table 6.1: Sensitivity analysis of SWAPs by hypothesis testing.

6.5.4 Performance Evaluation

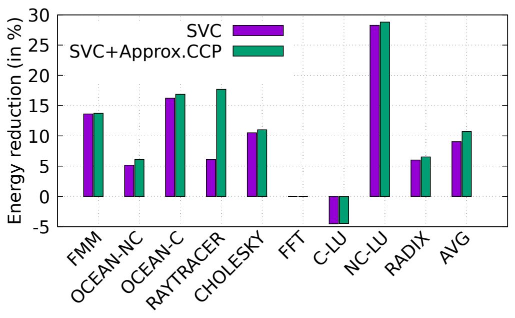
In this section, we show the benefits of our approximate computing method using the modified cache coherence protocol (referred to as CCP in the figure), in comparison with an exact baseline execution. We use *Sniper* to carry out the architectural simulation. Table 6.2 shows the simulation parameters configured for our simulation. Figure 6.11a shows the performance gain in terms of CPU cycles and energy reduction in comparison with the baseline execution.

Parameters	Values
Cores	16 and 32 cores, Nahalem 2.24GHz
L1	32KB I Cache, 32 KB D Cache, 32 Cache block size, private per core, 2 way associativity, 2 cycles hit latency, MESI coherence protocol
L2	128KB per core,S-NUCA, 8 way, 20 cycles hit latency.
Network on Chip (NoC)	Mesh 4 × {4,8}
Memory Controllers	4

Table 6.2: Architectural configuration used for simulations

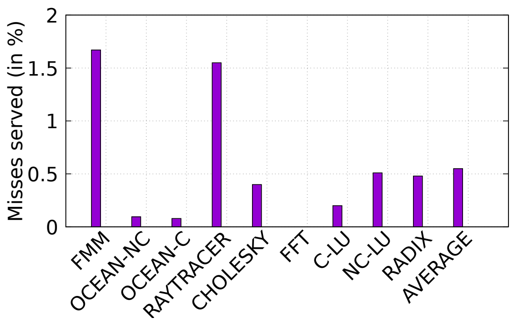


(a) Gain in CPU cycles

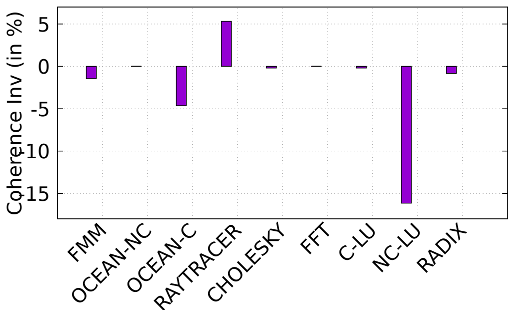


(b) Energy reduction (in Joules)

Figure 6.11: Performance gain (in %) with Stale Victim Cache (SVC) and SVC + Approx. CCP



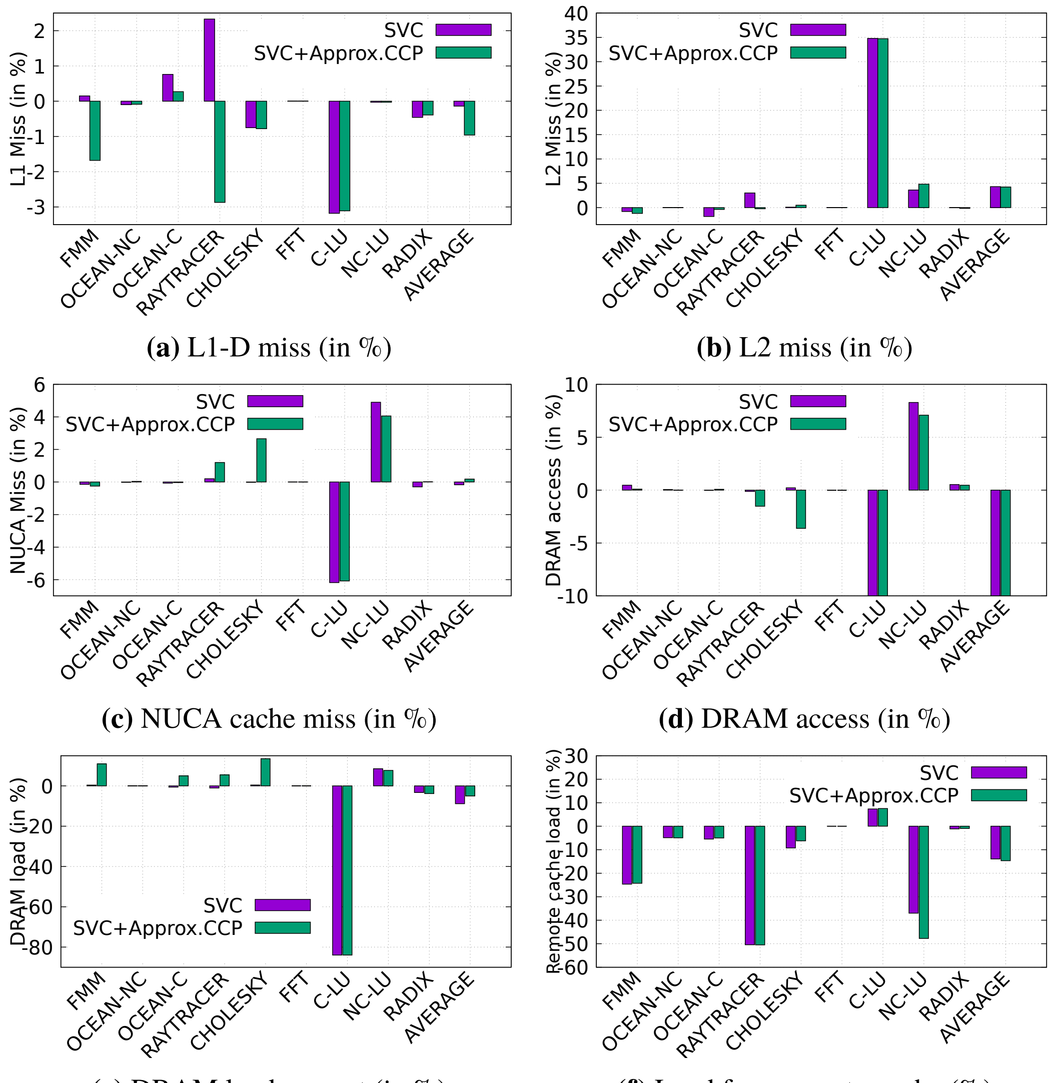
(a) Miss served by SVC and L1-D on cache misses (in %)



(b) Coherence invalidation reduction in Approx.CCP in comparison with exact execution

Figure 6.12: Cache-misses served by SVC and L1-D cache in SVC, and cache-misses in our approach (Approx-CCP), and coherence invalidation in our approach

On an average, our approach shows 15.5% gain in terms of CPU cycles and 11.5% reduction in energy. Figure 6.13f shows a comparison with [65], that proposes to serve stale values on load misses due to coherence invalidation. With our approach, we observe a further 1.72% gain in CPU cycles and 1.7% reduction in energy utilization. Figure 6.12 shows the memory sub-system statistics. We observe on an average, 0.26% reduction in L1-D misses, 1.45% reduction in invalidation messages due to coherence and 15.46% reduction in request to load data messages from remote caches. This demonstrates the achieved reduction in the overhead



(a) L1-D miss (in %)

(b) L2 miss (in %)

(c) NUCA cache miss (in %)

(d) DRAM access (in %)

(e) DRAM load request (in %)

(f) Load from remote-cache (%)

Figure 6.13: Architectural simulation statistics for SVC and Approx CCP in comparison with exact execution

of ensuring cache-coherence, with our approximate execution scheme.

6.6 Summary

In this chapter, we present a sensitivity analysis technique for instructions in a multi-threaded program, in order to determine the ones which are approximable. We show that a substantial number of thread instructions deem fit to be approximable. In order to harmonize the hardware-software co-design, we propose a modified approximation aware cache coherence protocol that triggers on every approximable write miss, and acts differently as compared to a classical exact execution. We show an application of our analysis by not sending the invalidation messages triggered by such approximable writes on shared data. We present performance benefits in terms of CPU cycles and energy benefits using an architectural simulator. Our experiments make us believe that strict coherence requirement may be dispensed with for many approximate computing applications, giving us benefits in performance and energy.

Chapter 7

Conclusion and Future Directions

The objective of this thesis is to study the evolving paradigm of approximate computing from two different perspectives. On one side, we propose strategies to automatically identify approximable data and instruction elements inside application programs, such that controlled inaccuracies in these identified elements do not affect the correctness of computation beyond acceptable limits. On the other side, we combine the benefits of approximate computation with speculative execution inside modern processors to derive significant benefits, as we demonstrate through extensive experiments on architecture workloads.

In this thesis, we begin our study on using statistical sampling techniques for automatic classification / identification of application data that are resilient to limited errors, in other words, minor errors in those data elements are still acceptable in the application domain they are used in. Indeed, a wide range of application domains today, ranging from machine learning, computer vision, signal processing to planning and robotics, exhibit an intrinsic resilience towards minor errors, and are thus, suitable candidate domains for approximate computing to be applied to. The challenge in most of these domains is in identifying automatically the correct spots that lend themselves to approximation, considering the scale and complexity of modern workloads, and the complexity of the identification task thereof. Identifying insensitive error resilient data of an application is considered a non-trivial task, especially when the application is large and has substantial data and control dependencies. Manual annotation of data may not be reliable and may result in unacceptable output even when one data is mis-annotated as insensitive / approximable in approximation aware programming languages like EnerJ. While a number of methods for automated resilience quantification and identification have been proposed in literature, the key novelty of our approach is in combining static and dynamic analysis with statistical sampling, that gives us an unique edge over existing methods in literature. Additionally, we show the reliability of our analysis with empirical results. Indeed, as we demonstrate through extensive experiments on real workloads, the quality of the computation resulting out of approximations professed by our methods has a distinct advantage over ones produced using existing approaches or even manual annotations. We believe that our proposal has the potential to widen the horizon of approximate computing to a wider application domain going forward.

In our second contribution as outlined in Chapter 5, we extend the framework for approximability analysis to program instructions. While our main objects of interest in Chapter 3 are around data elements in a program, the focus in this chapter shifts to approximability analysis of program instructions. While approximation in data elements can give us some performance benefits, the main motivation behind this work is to be able to connect approximable instructions to the execution runtime and explore ways to expedite or derive benefits therein. In particular, we propose a technique that can automatically classify program data and load / branch instructions that are amenable for approximation. Further, a Bayesian analysis methodology is proposed to identify the loads / branches that are jointly approximable. As we witness through our experiments, our intuitions are indeed well justified. It is often the case that a good number of instructions in an application in these domains are approximable as well, however, these are outside the scope of optimization of even the most sophisticated compilers. Our approach is able to identify these instructions in an automated way, building on the foundation of approximability analysis outlined earlier, and therefore, leads to substantial performance benefits during execution.

In the subsequent chapters, we build on the benefits that our approximability analysis framework provides, and connect to modern processor pipelines. In particular, we propose the hardware design of a processor pipeline that can exploit the approximability knowledge of instructions to implement speculative execution with a selective no rollback policy. The paradigm of speculative execution is a defacto performance enriching technique in modern processors, whereby modern processors hide the latency overheads of long latency instructions by suitably speculating values or program paths, without holding the computation for long latency instructions to respond with relevant values. However, when the speculation fails, a rollback is needed to always keep program executions on the correct path. Our novel contribution in this context is a proposal to dispense rollback for approximable instructions, by allowing the computation on the predicted path with predicted values irrespective of a failure. The fact that these instructions are provably approximable, as deemed so by our earlier analysis, helps us leverage on this no rollback model of computation, that has the potential of being able to derive substantial performance benefits in terms of latency and energy since a significant number of instruction executions are done away with. In particular, we look at loads for which cache misses usually dictate a memory access, and a branch, for which, a misprediction mandates a rollback and fetch and execute from the correct path. Based on our identification of approximable loads and branches, we allow the processor to continue with random / stale values for cache misses and on the false path in case of a misprediction. Indeed, as results show, both these techniques lead to substantial performance gains as expected, and significant latency and energy savings.

As the final contribution of this thesis, we extend our approximability analysis technique to instructions in a multi-threaded execution context and automatically identify concurrent writes that are approximable. This has a significant performance promise in multi-threaded concurrent workloads, considering the fact that a significant number of messages are exchanged in this context for maintaining coherence of shared data across the different processors on which the threads execute. As our experiments confirm, in this case as well, a substantial number of such instructions deem fit to be approximable, and do not require the processors to synchronize to be on the correct execution path within allowable error limits. In order to bridge the hardware / software co-design, we propose a modified approximation aware cache coherence protocol that triggers on every approximable write miss, and acts differently as compared to a classical exact execution. We show an application of our analysis by not sending the invalidation messages triggered by such approximable writes on shared data. Our experiments confirm that in this case as well, our intuitions are well justified, and the fact that strict coherence may be dispensed with for many approximate computing applications executing on multiple processors, with guaranteed benefits in performance and energy.

This thesis opens up a lot of avenues for future exploration around approximate computing. We outline some of the directions below.

- *Extending the approximability analysis framework for more fine-grained analysis:* In our current work, we rely on statistical sampling techniques for identification of approximable instructions. Our analysis is tightly coupled with static and dynamic program analysis techniques that exploit the syntactical structure of the program in terms of the control and data flow to derive better insights for approximability. While our analysis is effective in practice and leads to quite accurate results in terms of resilience analysis, we believe that it is often possible to derive more accurate insights by considering the semantic structure of the program, the data elements and the data structures they are part of. In particular, it may often be the case that a data element is not approximable when the end to end program input space is considered, however, it may be resilient to errors only within certain input domains. Our current analysis is not able to identify such fine-grained approximable points, and works only at the entire program boundary considering the entire program space when the degree of deviation of an approximated execution is compared with the correct execution. Further, we consider instructions as individual entities in our approximability analysis. Going ahead, we wish to extend our framework to be able to provide a quantitative assessment of the contribution of each block of code to the quality of the final result. Such fine-grained analysis can derive insights like if an input or intermediate variable x is perturbed within an interval, and, as a result, the variation of the value of an output variable y is small, then x is insignificant to y. We wish to explore techniques from interval analysis and algorithmic differentiation to derive such refined insights, something that is beyond the scope of our current method. We believe this will enrich our current framework even further, with an ability to automatically quantify the significance of computations and to detect variations in significance among parts of code.

- *A study on combining approximations:* In our thesis, we have focused primarily on optimizing on single points of approximability in the execution stack and to show benefits in energy or execution time. Going ahead, we wish to investigate if combining multiple approximation techniques spanning more than one layer of the system stack compound or reduce the benefits, and if these benefits / reductions are generic across different application domains. In order to provide a concrete demonstration, we plan to focus on three approximation categories: skipping computations, approximation of arithmetic instructions, and approximation of communications between computational elements. As representatives of each category, we may evaluate loop perforation, reduced arithmetic precision, and relaxation of synchronization, along similar lines as in our current work. If our findings are positive, meaning that these indeed generate compounded benefits, we wish to revisit the design of the execution runtime stack to be able to make way for approximations all across. This, we believe, will make way for the design of tightly integrated approximate accelerators. This will enable moving applications into a paradigm in which the architecture, programming model, and even the algorithms used to realize the application are all fundamentally embraced and tuned for approximate computing.

- *Quantifying approximations with formal guarantees:* An important direction of future research is on formally proving correctness of the approximations derived using our methods. In particular, we wish to utilize the rich foundations of formal methods, probabilistic / quantitative reasoning to derive correctness guarantees on the quality of the approximations with respect to their deviations from the correct output. In the present scenario, the guarantees provided are based on the technique used to derive approximability, either through controlled sampling or using data flow analysis. Our plan is to derive formal guarantees on the extent of correctness compromised by the approximations introduced by our method. We believe this can be an important aid for a designer to embrace approximation, that is missing in the current design flow.

We believe that approximate computing research has a significant role to play in the next era of computer system design. With the semi-conductor industry increasingly embracing the fact that traditional approaches to scaling performance are losing steam and hardware techniques to ensure perfect reliability are growing to be prohibitively expensive, approximation will start getting adopted in mainstream design flows. We believe that our contributions outlined in this thesis will have an useful impact going forward as we accept to embrace the approximate computing paradigm in our routine computations.

Appendix A

Publications

A.1 List of Publications

- **B. Nongpoh**, R. Ray, M. Das, and A. Banerjee, **Enhancing speculative execution with selective approximate computing**, ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 24, no. 2, 26:1-26:29, 2019.
- **B. Nongpoh**, R. Ray, S. Dutta, and A. Banerjee, **Autosense: A framework for automated sensitivity analysis of program data**, IEEE Transactions on Software Engineering vol. 43, no. 12, pp. 1110–1124, 2017.
- **B. Nongpoh**, R. Ray, and A. Banerjee, **Approximate computing for multithreaded programs in shared memory architectures**, In Proceedings of the 17th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE) 2019, pp. 11:1-11:9

A.2 Awards and Recognition

- Google Travel Grant award amount $2000 towards attending **the 11th joint meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)** Paderborn, Germany, September 04-08, 2017.

- ACM SIGSOFT CAPS Travel Grant award amount $430 towards attending **ESEC/FSE 2017** Paderborn, Germany, September 04-08, 2017.

Bibliography

[1] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing", in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–9. doi: `10.1145/2463209.2488873` (page 1).

[2] A. Agrawal, J. Choi, K. Gopalakrishnan, S. Gupta, R. Nair, J. Oh, D. A. Prener, S. Shukla, V. Srinivasan, and Z. Sura, "Approximate computing: Challenges and opportunities", in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, 2016, pp. 1–8. doi: `10.1109/ICRC.2016.7738674` (page 3).

[3] E. Kreyszig, *Advanced Engineering Mathematics*. USA: John Wiley and Sons, INC., 2006, isbn: 978-0-471-48885-9 (pages 6, 50).

[4] A. Wald, "Sequential tests of statistical hypotheses", *Ann. Math. Statist.*, vol. 16, no. 2, pp. 117–186, Jun. 1945. doi: `10.1214/aoms/1177731118`. [Online]. Available: `http://dx.doi.org/10.1214/aoms/1177731118` (pages 7, 53–55, 81).

[5] M. Naik. (2018). Introduction to Software Analysis, [Online]. Available: `https://www.cis.upenn.edu/~alur/CIS673/isil-plmw.pdf` (visited on 01/01/2018) (pages 8, 9, 14, 15).

[6] I. Dillig. (2014). A Gentle Introduction to Program Analysis, [Online]. Available: `https://www.cis.upenn.edu/~alur/CIS673/isil-plmw.pdf` (visited on 01/21/2014) (page 8).

[7] M. Pistoia and U. Erlingsson, "Programming languages and program analysis for security: A three-year retrospective", *SIGPLAN Not.*, vol. 43, no. 12, pp. 32–39, Feb. 2009, issn: 0362-1340. doi: `10.1145/1513443.1513449`. [Online]. Available: `http://doi.acm.org/10.1145/1513443.1513449` (page 8).

[8] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, "Comparing bug finding tools with reviews and tests", in *Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems*, ser. TestCom'05, Montreal,

Canada: Springer-Verlag, 2005, pp. 40–55, ISBN: 3-540-26054-4, 978-3-540-26054-7. DOI: `10.1007/11430230_4`. [Online]. Available: `http://dx.doi.org/10.1007/11430230_4` (page 8).

[9] J. Laski and W. Stanley, *Software Verification and Analysis: An Integrated, Hands-On Approach*, 1st ed. Springer Publishing Company, Incorporated, 2009, ISBN: 1848822391, 9781848822399 (page 8).

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811 (page 8).

[11] S. Midkiff, *Automatic Parallelization: An Overview of Fundamental Compiler Techniques*. Morgan and Claypool, 2012, ISBN: 9781608458424. [Online]. Available: `https://ieeexplore.ieee.org/document/6813266` (page 8).

[12] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, “Tricorder: Building a program analysis ecosystem”, in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15, Florence, Italy: IEEE Press, 2015, pp. 598–608, ISBN: 978-1-4799-1934-5. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2818754.2818828` (page 8).

[13] E. Dijkstra. (2005). On the reliability of programs, [Online]. Available: `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html` (visited on 06/27/2015) (page 9).

[14] A. Moller and M. I. Schwartzbach. (2018). Static Program Analysis, [Online]. Available: `https://cs.au.dk/~amoeller/spa/` (visited on 10/01/2018) (pages 9, 16–18, 20, 21).

[15] W. Landi, “Undecidability of static analysis”, *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, Dec. 1992, ISSN: 1057-4514. DOI: `10.1145/161494.161501`. [Online]. Available: `http://doi.acm.org/10.1145/161494.161501` (page 9).

[16] (page 9).

[17] N. Ayewah and W. Pugh, “The google findbugs fixit”, in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA ’10, Trento, Italy: ACM, 2010, pp. 241–252, ISBN: 978-1-60558-823-0. DOI: `10.1145/1831708.1831738`. [Online]. Available: `http://doi.acm.org/10.1145/1831708.1831738` (page 9).

[18] Synopsys. (2019). Coverity Scan Tool, [Online]. Available: `https://scan.coverity.com/` (visited on 05/14/2019) (page 9).

[19] Facebook. (2019). Facebook Infer : A static analyzer for Java, C, C++, and Objective-C, [Online]. Available: `https://github.com/facebook/infer` (visited on 05/14/2019) (page 9).

[20] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, “Slam and static driver verifier: Technology transfer of formal methods inside microsoft”, in *Integrated Formal Methods*, E. A. Boiten, J. Derrick, and G. Smith, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–20, ISBN: 978-3-540-24756-2 (page 9).

[21] D. R. Cok and J. R. Kiniry, “Esc/java2: Uniting esc/java and jml”, in *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 108–128, ISBN: 978-3-540-30569-9 (page 9).

[22] F. E. Allen, “Control flow analysis”, in *Proceedings of a Symposium on Compiler Optimization*, Urbana-Champaign, Illinois: ACM, 1970, pp. 1–19. DOI: `10.1145/800028.808479`. [Online]. Available: `http://doi.acm.org/10.1145/800028.808479` (page 9).

[23] F. E. Allen and J. Cocke, “A program data flow analysis procedure”, *Communications of the ACM*, vol. 19, no. 3, p. 137, 1976 (pages 12, 13).

[24] “Reaching definition analysis”, in *Reasoning About Program Transformations: Imperative Programming and Flow of Data*, J.-F. Collard, Ed. New York, NY: Springer New York, 2003, pp. 77–122, ISBN: 978-0-387-22461-9. DOI: `10.1007/978-0-387-22461-9_5`. [Online]. Available: `https://doi.org/10.1007/978-0-387-22461-9_5` (page 12).

[25] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999, ISBN: 3540654100 (pages 12, 17, 20, 69, 70, 73).

[26] `https://web.cs.wpi.edu/~kal/PLT/PLT9.4.html` (page 12).

[27] B. Meyer. (2019). Soundness and Completeness: With Precision, [Online]. Available: `https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-with-precision/fulltext` (visited on 04/20/2019) (page 15).

[28] Y. Yang. (2013). Note for Introduction to Lattice Theory, [Online]. Available: `http://www.math.ucla.edu/~yy26/works/Lattice%20Talk.pdf` (visited on 05/18/2013) (page 17).

[29] J. B. Kam and J. D. Ullman, “Monotone data flow analysis frameworks”, *Acta Inf.*, vol. 7, no. 3, pp. 305–317, Sep. 1977, ISSN: 0001-5903. DOI: `10.1007/BF00290339`. [Online]. Available: `http://dx.doi.org/10.1007/BF00290339` (pages 18, 20).

[30] A. Platzer. (2010). Lecture Notes on Monotone Frameworks and Abstract Interpretation, [Online]. Available: `https://www.cs.cmu.edu/~aplatzer/course/Compilers/27-monframework.pdf` (visited on 11/30/2010) (pages 19, 20).

[31] M. D. Ernst, “Static and dynamic analysis: Synergy and duality”, in *IN WODA 2003: ICSE WORKSHOP ON DYNAMIC ANALYSIS*, 2003, pp. 24–27 (page 21).

[32] T. Ball, “The concept of dynamic analysis”, *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 216–234, Oct. 1999, ISSN: 0163-5948. DOI: `10.1145/318774.318944`. [Online]. Available: `http://doi.acm.org/10.1145/318774.318944` (page 22).

[33] J. R. Larus and T. Ball, “Rewriting executable files to measure program behavior”, *Software Practice and Experience*, vol. 24, pp. 197–218, 1994 (page 22).

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation”, in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05, Chicago, IL, USA: ACM, 2005, pp. 190–200, ISBN: 1-59593-056-6. DOI: `10.1145/1065010.1065034`. [Online]. Available: `http://doi.acm.org/10.1145/1065010.1065034` (pages 22, 23, 97).

[35] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation”, in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07, San Diego, California, USA: ACM, 2007, pp. 89–100, ISBN: 978-1-59593-633-2. DOI: `10.1145/1250734.1250746`. [Online]. Available: `http://doi.acm.org/10.1145/1250734.1250746` (page 22).

[36] C. Lattner and V. Adve, “Llvm: A compilation framework for lifelong program analysis and transformation”, in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, IEEE, 2004, pp. 75–86 (page 22).

[37] E. Kuleshov, *Using the asm framework to implement common java bytecode transformation patterns*, 2007 (page 22).

[38] S. Chiba and M. Nishizawa, “An easy-to-use toolkit for efficient java bytecode translators”, in *Generative Programming and Component Engineering*, F. Pfenning and Y. Smaragdakis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 364–376, ISBN: 978-3-540-39815-8 (page 22).

[39] A. Commons, “Bcel: Byte code engineering library”, [Online]. Available: `https://commons.apache.org/proper/commons-bcel/index.html` (pages 22, 57).

[40] ARM. (2019). Arm documentation, [Online]. Available: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0222b/ch01s01s01.html` (visited on 05/03/2019) (page 24).

[41] T. Ball and J. R. Larus, “Branch prediction for free”, in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI ’93, Albuquerque, New Mexico, USA: ACM, 1993, pp. 300–313, ISBN: 0-89791-598-4. DOI: `10.1145/155090.155119`. [Online]. Available: `http://doi.acm.org/10.1145/155090.155119` (page 25).

[42] M. Chang and Y. Chou, "Branch prediction using both global and local branch history information", *IEE Proceedings - Computers and Digital Techniques*, vol. 149, no. 2, pp. 33–38, 2002, ISSN: 1350-2387. DOI: `10.1049/ip-cdt:20020273` (page 25).

[43] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons", in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, ser. HPCA '01, Washington, DC, USA: IEEE Computer Society, 2001, pp. 197–, ISBN: 0-7695-1019-1. [Online]. Available: `http://dl.acm.org/citation.cfm?id=580550.876441` (page 25).

[44] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, "Design tradeoffs for the alpha ev8 conditional branch predictor", in *Proceedings 29th Annual International Symposium on Computer Architecture*, 2002, pp. 295–306. DOI: `10.1109/ISCA.2002.1003587` (page 26).

[45] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction", *SIGPLAN Not.*, vol. 31, no. 9, pp. 138–147, Sep. 1996, ISSN: 0362-1340. DOI: `10.1145/248209.237173`. [Online]. Available: `http://doi.acm.org/10.1145/248209.237173` (pages 27, 39).

[46] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks", in *Proceedings of ICS'06*, ser. ICS '06, Cairns, Queensland, Australia: ACM, 2006, pp. 324–334, ISBN: 1-59593-282-8. DOI: `10.1145/1183401.1183447`. [Online]. Available: `http://doi.acm.org/10.1145/1183401.1183447` (page 29).

[47] M. C. Rinard and M. S. Lam, "The design, implementation, and evaluation of jade", *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 3, pp. 483–545, May 1998, ISSN: 0164-0925. DOI: `10.1145/291889.291893`. [Online]. Available: `http://doi.acm.org/10.1145/291889.291893` (page 29).

[48] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation", *SIGPLAN Not.*, vol. 46, no. 6, pp. 164–174, Jun. 2011, ISSN: 0362-1340. DOI: `10.1145/1993316.1993518`. [Online]. Available: `http://doi.acm.org/10.1145/1993316.1993518` (pages 30, 34, 40, 44, 71).

[49] P. Roy, R. Ray, C. Wang, and W. F. Wong, "Asac: Automatic sensitivity analysis for approximate computing", in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14, Edinburgh, United Kingdom: ACM, 2014, pp. 95–104, ISBN: 978-1-4503-2877-7. DOI: `10.1145/2597809.2597812`. [Online]. Available: `http://doi.acm.org/10.1145/2597809.2597812` (pages 32, 34).

[50] R. E. Rodrigues, V. H. Sperle Campos, and F. M. Quintão Pereira, "A fast and low-overhead technique to secure programs against integer overflows", in *Proceedings of*

the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2013, pp. 1–11. doi: `10.1109/CGO.2013.6494996` (page 32).

[51] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code", *Technometrics*, vol. 21, no. 2, pp. 239–245, 1979, issn: 00401706. [Online]. Available: `http://www.jstor.org/stable/1268522` (page 33).

[52] S. K. Palaniappan, B. M. Gyori, B. Liu, D. Hsu, and P. S. Thiagarajan, "Statistical model checking based calibration and analysis of bio-pathway models", in *Computational Methods in Systems Biology*, A. Gupta and T. A. Henzinger, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 120–134, isbn: 978-3-642-40708-6 (page 33).

[53] M. Carbin and M. C. Rinard, "Automatically identifying critical input regions and code in applications", in *Proceedings of ISSTA'10*, ACM, 2010, pp. 37–48 (pages 34, 44).

[54] W. H. E. Day and H. Edelsbrunner, "Efficient algorithms for agglomerative hierarchical clustering methods", *Journal of Classification*, vol. 1, no. 1, pp. 7–24, 1984, issn: 1432-1343. doi: `10.1007/BF01890115`. [Online]. Available: `https://doi.org/10.1007/BF01890115` (pages 35, 36).

[55] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing", in *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, ACM, 2013, 113:1–113:9, isbn: 978-1-4503-2071-9. doi: `10.1145/2463209.2488873`. [Online]. Available: `http://doi.acm.org/10.1145/2463209.2488873` (pages 36, 44).

[56] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation", in *Proceedings of FSE'11*, ser. ESEC/FSE '11, Szeged, Hungary: ACM, 2011, pp. 124–134, isbn: 978-1-4503-0443-6. doi: `10.1145/2025113.2025133`. [Online]. Available: `http://doi.acm.org/10.1145/2025113.2025133` (pages 37, 90).

[57] J. S. Miguel, M. Badr, and N. E. Jerger, "Load value approximation", in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 127–139. doi: `10.1109/MICRO.2014.22` (page 39).

[58] A. Yazdanbakhsh, B. Thwaites, H. Esmaeilzadeh, G. Pekhimenko, O. Mutlu, and T. C. Mowry, "Mitigating the memory bottleneck with approximate load value prediction", *IEEE Design Test*, vol. 33, no. 1, pp. 32–42, 2016, issn: 2168-2356. doi: `10.1109/MDAT.2015.2504899` (pages 39, 40).

[59] L. Ceze, K. Strauss, J. Tuck, J. Torrellas, and J. Renau, "Cava: Using checkpoint-assisted value prediction to hide l2 misses", *ACM Trans. Archit. Code Optim.*, vol. 3,

no. 2, pp. 182–208, Jun. 2006, ISSN: 1544-3566. DOI: `10.1145/1138035.1138038`. [Online]. Available: `http://doi.acm.org/10.1145/1138035.1138038` (page 39).

[60] F. G. et. al., *Speculative execution based on value prediction*. Technion-IIT, 1996 (page 39).

[61] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing", *IEEE Micro*, vol. 31, no. 5, pp. 7–17, 2011, ISSN: 0272-1732. DOI: `10.1109/MM.2011.89` (page 40).

[62] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A case for toggle-aware compression for gpu systems", in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 188–200. DOI: `10.1109/HPCA.2016.7446064` (page 40).

[63] M. Burtscher and B. G. Zorn, "Hybrid load-value predictors", *IEEE Transactions on Computers*, vol. 51, no. 7, pp. 759–774, 2002, ISSN: 0018-9340. DOI: `10.1109/TC.2002.1017696` (page 41).

[64] Y. Sazeides and J. E. Smith, "The predictability of data values", in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30, Research Triangle Park, North Carolina, USA: IEEE Computer Society, 1997, pp. 248–258, ISBN: 0-8186-7977-8. [Online]. Available: `http://dl.acm.org/citation.cfm?id=266800.266824` (page 41).

[65] P. V. Rengasamy, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Exploiting staleness for approximating loads on cmps", in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 343–354. DOI: `10.1109/PACT.2015.27` (pages 42, 131, 134).

[66] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors", *SIGARCH Comput. Archit. News*, vol. 18, no. 2SI, pp. 15–26, May 1990, ISSN: 0163-5964. DOI: `10.1145/325096.325102`. [Online]. Available: `http://doi.acm.org/10.1145/325096.325102` (page 42).

[67] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "Autosense: A framework for automated sensitivity analysis of program data", *IEEE Trans. Software Eng.*, vol. 43, no. 12, pp. 1110–1124, 2017. DOI: `10.1109/TSE.2017.2654251`. [Online]. Available: `https://doi.org/10.1109/TSE.2017.2654251` (pages 43, 68, 90).

[68] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware", in *ACM SIGPLAN Notices*, ACM, vol. 48, 2013, pp. 33–52 (page 44).

[69] P. Roy, R. Ray, C. Wang, and W. F. Wong, "Asac: Automatic sensitivity analysis for approximate computing", *SIGPLAN Not.*, vol. 49, no. 5, pp. 95–104, Jun. 2014,

ISSN: 0362-1340. DOI: `10.1145/2666357.2597812`. [Online]. Available: `http://doi.acm.org/10.1145/2666357.2597812` (pages 44, 63).

[70] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation", in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, San Jose, California, USA: ACM, 2011, pp. 164–174, ISBN: 978-1-4503-0663-8. DOI: `10.1145/1993498.1993518`. [Online]. Available: `http://doi.acm.org/10.1145/1993498.1993518` (pages 44, 60, 64, 65).

[71] R. Pozo and B. Miller, *Scimark 2.0*. [Online]. Available: `http://math.nist.gov/scimark2/` (pages 44, 58, 59).

[72] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs", in *MICRO 2012, Canada, December 1-5, 2012*, IEEE Computer Society, 2012, pp. 449–460, ISBN: 978-1-4673-4819-5. DOI: `10.1109/MICRO.2012.48`. [Online]. Available: `http://dx.doi.org/10.1109/MICRO.2012.48` (pages 44, 58, 59, 101).

[73] W.-F. Wong, P. Roy, R. Ray, and N.-M. Ho, "Compilation and other software techniques enabling approximate computing", in *Approximate Circuits: Methodologies and CAD*, S. Reda and M. Shafique, Eds. Cham: Springer International Publishing, 2019, pp. 443–463, ISBN: 978-3-319-99322-5. DOI: `10.1007/978-3-319-99322-5_22`. [Online]. Available: `https://doi.org/10.1007/978-3-319-99322-5_22` (pages 45, 46).

[74] A. Sampson. (2015). Hardware and Software for Approximate Computing, [Online]. Available: `https://homes.cs.washington.edu/~djg/theses/sampson_thesis.pdf` (visited on 01/01/2010) (page 46).

[75] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto, *Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models*. New York, NY, USA: Halsted Press, 2004, ISBN: 0470870931 (page 46).

[76] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: an intermediate code-level fault injection tool for hardware faults", in *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5, 2015*, IEEE, 2015, pp. 11–16, ISBN: 978-1-4673-7989-2. DOI: `10.1109/QRS.2015.13`. [Online]. Available: `http://dx.doi.org/10.1109/QRS.2015.13` (page 50).

[77] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation", in *Proceedings of CGO'04)*, Palo Alto, California, 2004 (page 50).

[78] H. L. S. Younes, “Verification and planning for stochastic processes with asynchronous events”, PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania., 2005 (pages 50, 52, 53, 61).

[79] H. L. S. Younes and R. G. Simmons, “Probabilistic verification of discrete event systems using acceptance sampling”, in *In Proc. of CAV’02, volume 2404 of LNCS*, Springer, 2002, pp. 223–235 (pages 50, 81).

[80] E. W. Weisstein, “Hypothesis testing. from mathworld–a wolfram web resource.”, *URL http://mathworld.wolfram.com/HypothesisTesting.html*, (page 50).

[81] W. Feller, *An Introduction to Probability Theory and Its Applications*. Wiley, 1968, vol. 1, ISBN: 0471257087. [Online]. Available: `http://www.amazon.ca/exec/obidos/redirect?tag=citeulike04-20{and}path=ASIN/0471257087` (pages 50, 86).

[82] E. L. Lehmann and J. P. Romano, *Testing Statistical Hypotheses*. Springer-Verlag New York, 2005, ISBN: 978-0-387-98864-1. DOI: `10.1007/0-387-27605-X` (page 52).

[83] A. Legay, B. Delahaye, and S. Bensalem, “Statistical model checking: An overview”, in *RV 2010*, 2010, pp. 122–135. DOI: `10.1007/978-3-642-16612-9_11`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-642-16612-9_11` (page 53).

[84] “Java instrumentation library”, *URL http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html*, (page 57).

[85] J. Korhonen and J. You, “Peak signal-to-noise ratio revisited: Is simple beautiful?”, in *2012 Fourth International Workshop on Quality of Multimedia Experience*, 2012, pp. 37–38. DOI: `10.1109/QoMEX.2012.6263880` (page 60).

[86] A. Askarov and A. Myers, “A semantic framework for declassification and endorsement”, ser. ESOP’10, Paphos, Cyprus: Springer-Verlag, 2010, pp. 64–84, ISBN: 3-642-11956-5, 978-3-642-11956-9. DOI: `10.1007/978-3-642-11957-6_5`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-642-11957-6_5` (page 70).

[87] B. Nongpoh, R. Ray, M. Das, and A. Banerjee, “Enhancing speculative execution with selective approximate computing”, *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 2, 26:1–26:29, Feb. 2019, ISSN: 1084-4309. DOI: `10.1145/3307651`. [Online]. Available: `http://doi.acm.org/10.1145/3307651` (page 79).

[88] B. R. Rau *et al.*, “Instruction-level parallel processing: History, overview, and perspective”, *The journal of Supercomputing*, vol. 7, no. 1-2, pp. 9–50, 1993 (page 80).

[89] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious”, *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995, ISSN: 0163-5964. DOI: `10.1145/216585.216588`. [Online]. Available: `http://doi.acm.org/10.1145/216585.216588` (page 80).

[90] J. González and A. González, “Speculative execution via address prediction and data prefetching”, in *ICS*, 1997, pp. 196–203, ISBN: 0-89791-902-5 (page 80).

[91] D. I. August *et al.*, “Integrated predicated and speculative execution in the impact epic architecture”, in *ISCA*, 1998, pp. 227–237, ISBN: 0-8186-8491-7 (page 80).

[92] A. Mendelson and F. Gabbay, “Speculative execution based on value prediction”, EE Department TR 1080, Technion - Israel Institue of Technology, Tech. Rep., 1996 (page 80).

[93] M. H. Lipasti *et al.*, “Value locality and load value prediction”, *ACM SIGPLAN Notices*, vol. 31, no. 9, pp. 138–147, 1996 (page 80).

[94] A. N. Eden *et al.*, “The yags branch prediction scheme”, in *MICRO*, 1998, pp. 69–77, ISBN: 1-58113-016-3 (page 80).

[95] I.-C. K. Chen *et al.*, “Instruction prefetching using branch prediction information”, in *ICCD*, 1997, pp. 593–601 (page 80).

[96] G. S. Tyson, “The effects of predicated execution on branch prediction”, in *MICRO*, 1994, pp. 196–206 (page 80).

[97] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, “An evaluation of high-level mechanistic core models”, *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014, ISSN: 1544-3566. DOI: `10.1145/2629677` (pages 82, 97).

[98] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications”, in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’08, Toronto, Ontario, Canada: ACM, 2008, pp. 72–81, ISBN: 978-1-60558-282-5. DOI: `10.1145/1454115.1454128`. [Online]. Available: `http://doi.acm.org/10.1145/1454115.1454128` (pages 82, 101, 108).

[99] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, “Accept: A programmer-guided compiler framework for practical approximate computing”, 2015 (pages 83, 85, 101).

[100] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, “Image quality assessment: From error visibility to structural similarity”, *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004, ISSN: 1057-7149. DOI: `10.1109/TIP.2003.819861` (pages 83, 102).

[101] N. Kanopoulos, N. Vasanthavada, and R. L. Baker, “Design of an image edge detection filter using the sobel operator”, *IEEE Journal of Solid-State Circuits*, vol. 23, no. 2, pp. 358–367, 1988, ISSN: 0018-9200. DOI: `10.1109/4.996` (page 85).

[102] V. K. Rohatgi and A. M. E. Saleh, *An introduction to probability and statistics*. John Wiley and Sons, 2015 (page 89).

[103] N. Wang *et al.*, “Y-branches: When you come to a fork in the road, take it”, in *PACT*, 2003, pp. 56–66 (page 90).

[104] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003, ISBN: 0137903952 (page 92).

[105] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo Method (Wiley Series in Probability and Statistics)*, 2nd ed. 2007, ISBN: 0470177942, 9780470177945 (page 93).

[106] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph", *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 1, pp. 121–141, Jan. 1979, ISSN: 0164-0925. DOI: `10.1145/357062.357071`. [Online]. Available: `http://doi.acm.org/10.1145/357062.357071` (page 94).

[107] J Seward *et al.*, *Cachegrind: A cache-miss profiler*, 2004 (page 97).

[108] J. L. Hennessy *et al.*, *Computer architecture: A quantitative approach*, 2011 (page 97).

[109] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "Rfvp: Rollback-free value prediction with safe-to-approximate loads", *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, 62:1–62:26, Jan. 2016, ISSN: 1544-3566. DOI: `10.1145/2836168`. [Online]. Available: `http://doi.acm.org/10.1145/2836168` (pages 98, 106).

[110] S. Li *et al.*, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures", in *MICRO*, 2009, pp. 469–480, ISBN: 978-1-60558-798-1 (page 106).

[111] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979, ISSN: 0018-9340. DOI: `10.1109/TC.1979.1675439`. [Online]. Available: `https://doi.org/10.1109/TC.1979.1675439` (page 114).

[112] J. Huang, "Scalable thread sharing analysis", in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 1097–1108. DOI: `10.1145/2884781.2884811` (pages 114, 123, 124).

[113] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for java", *Acm Sigplan Notices*, vol. 34, no. 10, pp. 1–19, 1999 (page 116).

[114] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for java programs", *ACM Sigplan Notices*, vol. 34, no. 10, pp. 187–206, 1999 (page 116).

[115] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "Autosense: A framework for automated sensitivity analysis of program data", *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1110–1124, 2017, ISSN: 0098-5589. DOI: `10.1109/TSE.2017.2654251` (page 116).

[116] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011, ISBN: 012383872X, 9780123838728 (page 117).

[117] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, 1st. Morgan and Claypool Publishers, 2011, ISBN: 1608455645, 9781608455645 (pages 117, 119).

[118] M. Thomadakis, "The architecture of the nehalem processor and nehalem-ep smp platforms", *JFE Technical Report*, Mar. 2011 (page 118).

[119] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay", *Commun. ACM*, vol. 55, no. 7, pp. 78–89, Jul. 2012, ISSN: 0001-0782. DOI: `10.1145/2209249.2209269`. [Online]. Available: `http://doi.acm.org/10.1145/2209249.2209269` (page 118).

[120] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research", in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016, pp. 101–111. DOI: `10.1109/ISPASS.2016.7482078` (page 132).

[121] J. P. Singh, W. Weber, and A. Gupta, "Splash: Stanford parallel applications for shared-memory*", Stanford, CA, USA, Tech. Rep., 1992 (page 132).

[122] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations", in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ser. ISCA '95, S. Margherita Ligure, Italy: ACM, 1995, pp. 24–36, ISBN: 0-89791-698-0. DOI: `10.1145/223982.223990`. [Online]. Available: `http://doi.acm.org/10.1145/223982.223990` (page 132).