Low-overhead Online Code Transformations

by

Michael A. Laurenzano

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2016

Doctoral Committee:

Assistant Professor Jason Mars, Co-Chair
Assistant Professor Lingjia Tang, Co-Chair
Professor Scott Mahlke
Associate Professor Kevin P. Pipe

For Alexander, Mia and Laura.

ACKNOWLEDGEMENTS

I am fortunate enough to have unparalleled mentors and teachers, now and in the past. I thank my dissertation committee – Scott Mahlke, Jason Mars, Kevin Pipe and Lingjia Tang – for their insights and guidance in constructing this dissertation. My advisors, Jason and Lingjia, have taught me how to think about intellectual endeavors, research, and execution. Thank you both for always asking tough questions and giving me the opportunity to succeed. All of the members of the Clarity Lab, you have helped me grow as a scientist and learn how to teach. Laura Carrington, thank you for the many opportunities you've given me and for always taking my ideas seriously. Allan Snavely, you were a mentor and a friend. Thank you for recognizing my potential and having the foresight to drag me back into the research world in 2007. The world has been a little bit darker since you left it. To all of my other teachers, especially Mike Rongitsch, Lukasz Pruski, Lynne Small and Jane Friedman. Your mentorship and passion for the technical influences me to this day.

I am also fortunate enough to have the greatest family in the world. Steve and Mariana Laurenzano, my parents – thank you for putting up with everything, working tirelessly to make sure I started life on second base, and giving me the tools to get to home plate. My siblings – Angela, Matthew and Stephen. Thank you for toughening me up and for a lifetime of friendship. And most of all, thank you to Laura, Mia, and Alexander. You put up with me during the late nights, the good times and the bad times, and everything else along the way. Life would be empty without you. Thanks for making it all worthwhile.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

ABSTRACT

Low-overhead Online Code Transformations

by

Michael A. Laurenzano

Chairs: Jason Mars and Lingjia Tang

The ability to perform *online code transformations* – to dynamically change the implementation of running native programs – has been shown to be useful in domains as diverse as optimization, security, debugging, resilience and portability. However, conventional techniques for performing online code transformations carry significant runtime overhead, limiting their applicability for performance-sensitive applications. This dissertation proposes and investigates a novel low-overhead online code transformation technique that works by running the dynamic compiler asynchronously and in parallel to the running program. As a consequence, this technique allows programs to execute with the online code transformation capability at near-native speed, unlocking a host of additional opportunities that can take advantage of the ability to re-visit compilation choices as the program runs.

This dissertation builds on the low-overhead online code transformation mechanism, describing three novel runtime systems that represent in best-in-class solutions to three challenging problems facing modern computer scientists. First, I leverage online code transformations to significantly increase the utilization of multicore dat-

acenter servers by dynamically managing program cache contention. Compared to state-of-the-art prior work that mitigate contention by throttling application execution, the proposed technique achieves a 1.3-1.5× improvement in application performance. Second, I build a technique to automatically configure and parameterize approximate computing techniques for each program input. This technique results in the ability to configure approximate computing to achieve an average performance improvement of 10.2× while maintaining 90% result accuracy, which significantly improves over oracle versions of prior techniques. Third, I build an operating system designed to secure running applications from dynamic return oriented programming attacks by efficiently, transparently and continuously re-randomizing the code of running programs. The technique is able to re-randomize program code at a frequency of 300ms with an average overhead of 9%, a frequency fast enough to resist state-of-the-art return oriented programming attacks based on memory disclosures and side channels.

CHAPTER I

Introduction

Compilation is the process of turning code written in a high-level language into machine code instructions and data that can be understood by the machine. The job of the compiler is to ensure *correctness*, producing machine code that faithfully implements the source code specified by the programmer. However, correctness is just the beginning of what the modern compiler must do. There may be many machine code implementations of source code that are correct, and today's programmers rely heavily on the compiler to produce machine code that executes efficiently on the hardware platform. To achieve this efficiency the compiler makes a number of assumptions about the execution environment of the program. For example, differences in the L1 cache sizes among hardware platforms may cause the compiler to optimize locality by structuring code to use a different memory access pattern via a customized tiling optimization [99, 177]. The assumptions a compiler must make about a program's runtime environment are not limited just to the features of the hardware platform, also including the makeup and size of program inputs as well as the impact of other running programs and services on the system.

As a result of these assumptions and the rigidity they impose on software in terms of dealing with different runtime environments, there has been a wealth of prior work aiming to enable *online code transformations* to allow the assumptions made by the

compiler, and the resulting machine code, to be changed as the program's execution environment changes [16, 17, 29, 40, 41, 59, 91, 92, 98, 107, 109, 143, 152, 164, 169, 170, 173, 184, 190]. Although the roots of online code transformation are in performance optimization, there has since been a wide spectrum of other important use cases that have been shown in the literature. The main classes of use cases for online code transformations is depicted in Figure 1.1. Online code transformations have been used to enact security measures [175], improve the resilience of software to errors [64], to build debugging tools [87], and to facilitate software portability, enabling programs compiled for one machine to run on another [43, 53, 59].

Despite these prior efforts, dynamic compilation has not been widely adopted and put into continuous operation in performance-sensitive production and commercial domains. This dissertation argues that the key to facilitating adoption is to realize a mechanism for online code transformation that introduces low performance overhead, is platform-agnostic, has the full transformative power of a static compiler, and is capable of extrospectively examining the program's execution environment as the program runs. This motivates us to design a new approach to online code transformations that overcomes these challenges, and to investigate the implications of our approach on two important use cases for online code transformations – optimization and security.

1.1 Motivation

This section motivates the need for a deployable, low-overhead technique for online code transformations in the context of two critical problems that exist in today's computer systems.

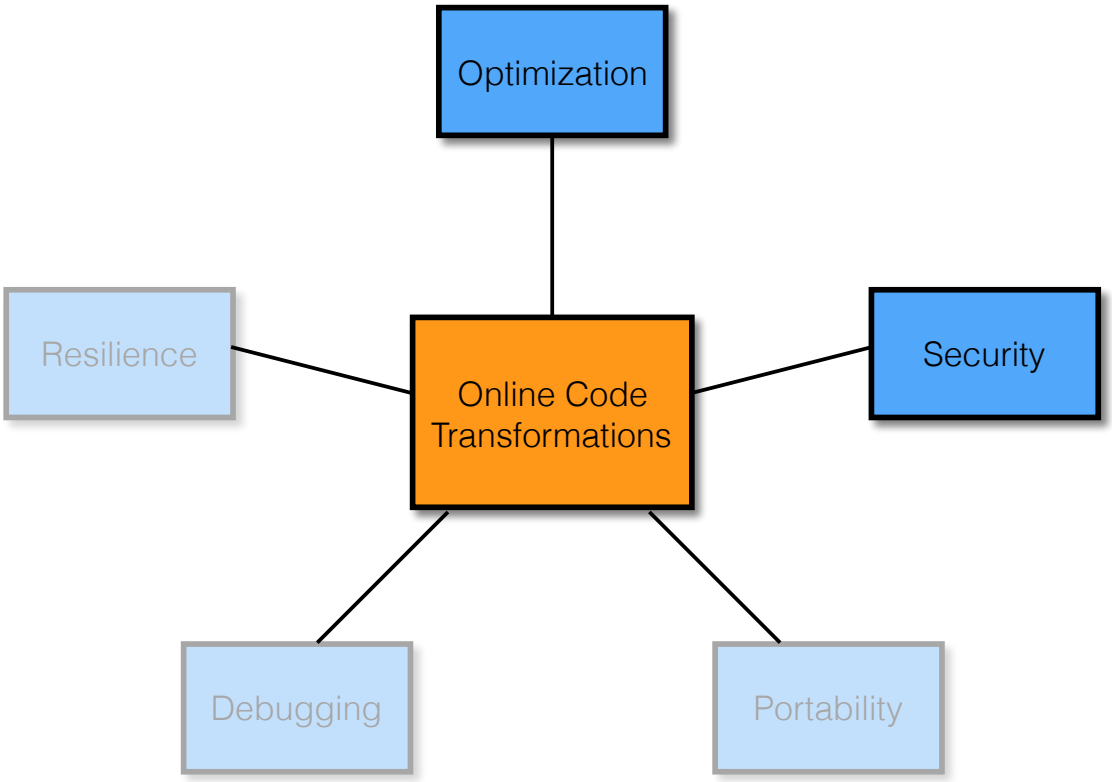


Figure 1.1: Online code transformations have a range of uses, including optimization, security, portability, resilience and debugging. This dissertation proposes a new low-overhead online code transformation technique and its implications on performance and security

1.1.1 Datacenter Server Utilization

Large enterprises such as Google and Facebook build and maintain large datacenters known as *Warehouse Scale Computers* (WSCs) dedicated to hosting popular user-facing webservices along with a variety of support applications. These datacenters are expensive and resource-intensive, with price tags now being measured in the billions of dollars [118, 121] and energy demands that require dedicated power plants. Maximizing the efficiency of compute resources in modern WSCs is an important challenge rooted in finding ways to consistently maximize server utilization to minimize cost.

Many datacenters run a mix of high-priority, often latency-sensitive, applications, such as web search and social networking, along with low-priority applications. The strategy of co-locating multiple applications on a single server has proved critical for maximizing utilization and minimizing cost [54, 113–115]. However, a significant challenge that emerges from the unpredictable dynamism in WSCs and limits our ability to co-locate is the threat of violating the *quality of service* (QoS) of user-facing latency-sensitive applications. Sources of dynamism include:

1. fluctuating user demand (load) for user-facing applications,
2. highly variable co-locations between user-facing and batch applications on a given machine, and
3. constant turnaround on each server; when an application completes, new applications are mapped to the server.

To deal with the threat of QoS violations, sophisticated software systems have been used to mitigate the effects of dynamism, acting to maximize server utilization while minimizing QoS violations [54, 114, 115, 170, 179, 180]. State-of-the-art runtime systems [170,180] solve the QoS problem by introducing short naps into the execution

stream of the low-priority application, unilaterally reducing the pressure on all shared resources in order to allow the high-priority application to make faster progress. ReQoS [170], for example, is a static compiler-enabled dynamic approach that throttles low-priority applications to allow them to be *safely* co-located with high-priority co-runners, guaranteeing the QoS of the high-priority co-runners and improving server utilization.

Such software systems are well-suited to improving server utilization, allowing some progress to be made on low-priority batch applications while guaranteeing QoS in latency-sensitive applications. This style of approach works well at meeting application QoS targets because there is some level of nap intensity that hinders low-priority applications enough to allow high-priority applications to meet their QoS targets. However, due to the inability to transform application code online, these approaches are limited to using the heavy handed approach of putting the batch application to sleep, i.e., napping, to reduce pressure on shared resources.

A capability missing in the WSC system software stack is the ability to dynamically transform and re-transform executing application code, which limits the design space when designing solutions to deal with the dynamism found in WSCs and leads to missed optimization opportunities. An example of such an optimization is to apply software non-temporal memory access hints to an application code to reduce its cache allocation and protect the QoS of its user-facing latency-sensitive co-runners. Modern ISAs, such as x86 and ARMv8 [2, 3], include prefetch instructions that hint to the processor that a subsequent memory access should not be cached. This instruction provides a mechanism that can cause an application to occupy more or less shared cache, and thus can enable higher throughput co-locations while protecting the QoS of high priority co-runners. However, it is difficult to leverage these hints effectively without a continuously-available low-overhead a mechanism to dynamically add and remove them in response to changing conditions on the server.

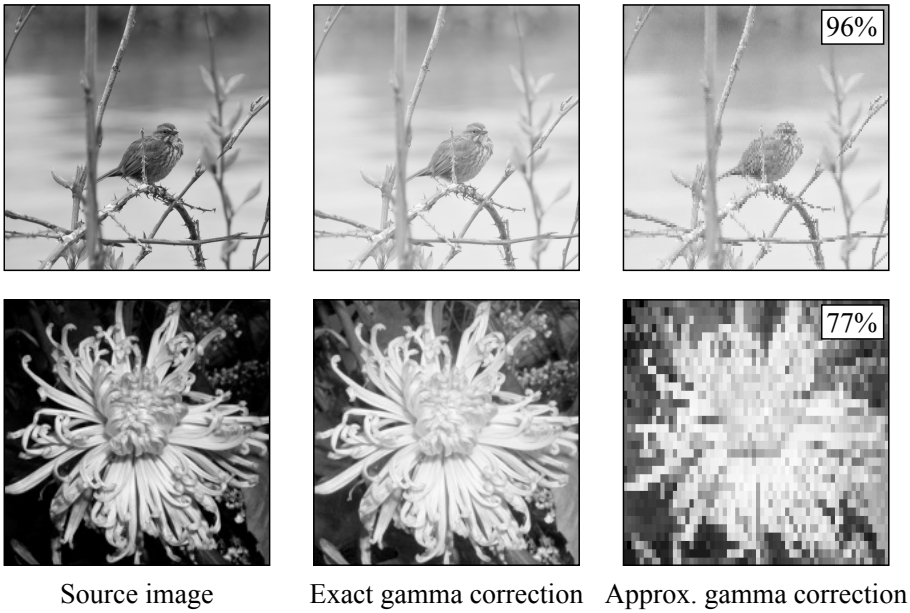


Figure 1.2: One approximation approach (16x8 tiling [146]) produces outputs of very different quality across inputs

1.1.2 Approximate Computing

The emergence of applications in the domains of image and sound processing, computer vision, machine learning, and data mining significantly increase the processing demands on compute infrastructure as the usage of wearable technologies [71,117,128] and intelligent personal assistants [9, 72, 119] rises. These emerging applications rely heavily on *regularly-structured computations* on inputs such images, video, and sound, and have loose constraints on the quality of output. The need for significant improvements in processing throughput for these applications along with loose quality constraints make them ideal candidates for *approximate computing*, where small amounts of output accuracy can be traded for large improvements in performance or energy.

The general purpose software-based approximate computing techniques typically applied to regularly-structured computations, such as loop perforation [84, 140], algorithm selection [8, 55], and numerical approximation [81], have been important and successful vehicles for realizing approximation in practice. These approaches can

be realized on commodity hardware, apply to a variety of problem types, and are straightforward for programmers to implement. However, the performance improvements achieved by prior work, 1.1× to 4× [15, 84, 85, 116, 142, 146, 147, 159], has been regarded as the ceiling on the performance and energy gains that are possible [35].

To breach this ceiling and realize the full potential of approximate computing, this work reconsiders *how* to approximate and observes that prior work falls into two categories.

Category #1: Calibration Intensive. Calibration computes both exact and approximate results and compares them to measure the accuracy of the approximate approach on some (set of) problem inputs. Calibration has been used to drive offline approaches [84], runtime systems [147], and within systems that use a combination of the two [15]. Because it is expensive to compute the exact solution and the accuracy of the approximate solution(s) on every input, calibration must be used sparingly. This approach encumbers the flexibility of approximation and ultimately the performance gains that can be realized.

Category #2: Profile Dependent. Profile guided approaches [63, 85, 159] make approximation decisions based on average or worst case behavior of a set of training inputs. This class of approaches relies on training with inputs that are representative of real-world inputs, which may be difficult to achieve in practice.

The key insight of this work is that, common to both classes of approaches, one approximation is used to cover multiple inputs. Those that focus on worst case accuracy can result in overly conservative approximation for many inputs, while those that focus on average case accuracy may be overly aggressive and fail to deliver sufficient accuracy in the worst case. As we show in this work, designing approximation systems that discount the differences between inputs hinders both the performance and accuracy of software-based approximation.

In this work, we present an approach to addressing this limitation that is guided by

two observations. Firstly, the accuracy of approximate programs can depend heavily on program input. Consider the example presented in Figure 1.2, which shows two images that have been processed by an identical approximate gamma correction with results that are of wildly different quality. Typical approaches to dealing with this difference in quality would dial down the aggressiveness of the approximation for both images, sacrificing performance for the first to produce satisfactory accuracy for the second. Addressing this problem requires a low-overhead mechanism that allows approximate computing techniques to be tuned and customized dynamically and for each specific problem input.

1.1.3 Code Reuse Attacks

Traditionally, operating systems have been designed to manage hardware and software resources in computing systems with the implicit assumption that the system does not have visibility into, nor control over, the code within executing processes. From the perspective of the OS, processes are black boxes for which to provide services such as thread management, scheduling, interrupt handling, memory management, file systems, device drivers, I/O, networking and security, among others.

Despite the efforts of system and software developers to build secure systems within this design paradigm, a class of execution hijack attacks called *code reuse attacks* – including a sophisticated form of code reuse known as return-oriented programming (ROP) – remain a viable method of orchestrating attacks that subvert the intent of running programs [34, 70, 157, 162]. It was recently reported that ROP has been used to facilitate more than 95% of the known Windows exploits over the last two years [136] and has been used as a key component in recent high-profile attacks on industrial targets [60].

The difficulty in preventing such attacks lies in their very nature – programs have a rich set of functionality in their executable code to efficiently perform useful

computation, effectively handing attackers a rich set of code widgets that can be subverted to achieve the attacker's ends. Without visibility into application code itself, and without a capability of modifying application code at runtime, modern operating systems are limited in their ability to defend against these attacks.

Current defenses against code reuse attacks fall into one of two categories: those that employ static Address Space Layout Randomization (ASLR), randomizing the locations of program components as execution begins [21, 77, 171, 174, 182], and those that enforce control-flow integrity (CFI) to stop unintended control flow changes and mitigate specific classes of code reuse attacks [39, 49, 50, 52, 160, 183]. Both of these categories of defenses have fundamental limitations.

- **Limitations of ASLR** — rudimentary types of code reuse attacks based on static program analysis are thwarted, or at least made mode difficult by, *static* ASLR. However, recent work has demonstrated that code reuse attacks can be fully orchestrated and executed *dynamically*, thus bypassing even the strongest forms of ASLR [156, 162].

- **Limitations of CFI** — recent literature [42, 52, 131] has shown that enforcing control-flow properties [4] (e.g., call-return parity) can defend against certain classes of code reuse attacks *when the method of attack is known in advance*. For instance, ROPDefender [52] maintains a shadow stack to ensure congruence between return targets and call sites to defend against return-based ROP attacks. However, ROP is general enough to be done without returns [37], and thus this defense is easily bypassed by sophisticated attackers.

The key motivation behind this work is that there is a third category of approaches that has not been well studied and requires rethinking the design of the modern OS. This approach is to design an OS that continuously has visibility into, and the capability to transform, application code as it runs — a *code transforming operating*

system. A system design that includes this visibility and transformation power over application code allows the system to *dynamically and continuously re-randomize native code* as applications execute, breaking some of the fundamental assumptions necessary to orchestrate a broad set of code reuse attacks.

Re-randomizing code is powerful because code reuse attacks are fundamentally bottlenecked by the ability of the attacker to gain visibility[1] into the executable bytes of the running program, and thus the time delay between the inception, construction, and execution of the attack is exploited to thwart such attacks. Code re-randomization constricts the window of time in which a program's bytes remain in one place, thus preventing an attacker that can gain visibility into the executable code of the program from making use of that code in an attack.

Prior work has acknowledged the benefits of code re-randomization [162, 174]. However, as valuable as it is to apply re-randomization, it is just as challenging to build a system to re-randomize code with low runtime overhead. This state of affairs was summarized in a recent paper describing a state-of-the art ROP attack:

> *"While [re-randomizing code pages] may be one way [to render our attack ineffective], we expect that re-randomization costs would make such a solution impractical."* [162]

1.2 Goals of Online Code Transformation

While the advantages of a low-overhead mechanism for online code transformation are clear, designing such a mechanism that is deployable in production environments has proved challenging. Despite a substantial body of prior work and having been shown to be useful in many problem domains [16, 17, 29, 40, 41, 59, 91, 92, 98, 107,

[1] *Visibility* means that the attacker has knowledge of the contents of some part of the memory. Such visibility can be gained, for example, via unintended memory disclosure bugs [162] or side channels [156].

109, 143, 152, 164, 169, 170, 173, 184, 190], dynamic compilation has not been widely adopted, particularly in production and commercial domains. Several challenges have prevented the realization of a *deployable* dynamic compilation mechanism:

- **Overhead** — It has been reported that companies such as Google tolerate no more than 1% to 2% degradation in performance to support dynamic monitoring approaches in production [137]. The high overhead that is common in traditional dynamic compilation frameworks has served as a barrier to adoption in these performance-critical environments.
- **Generality and Low Complexity** — To avoid hardware lockin and overly complex software maintenance, a deployable dynamic compilation system should impose little or no burden on application developers and should require no specialized hardware support.
- **Transformation Power** — Traditional dynamic optimizers raise native machine code to an intermediate representation before applying transformations. This approach limits the power of the transformations due to loss of source level information. Having the ability to apply transformations online that are as powerful as static compilation significantly impacts the flexibility of the dynamic compiler.
- **Continuous Extropsection** — In a highly dynamic environment where multiple applications co-run, specializing code to runtime conditions should be done both *introspectively*, based on a host program's behavior, and *extrospectively*, based on *external* applications that are co-located on the same machine. To accomplish this, a runtime code transformation system must be aware of changing conditions for both itself and its neighbors, applying or undoing transformations accordingly.

1.3 Design Overview

1.3.1 Protean Code

This dissertation describes the design of *protean code*, a general-purpose, near-free approach to monitoring, regenerating and replacing the code of running applications with semantically equivalent, specialized code versions that reflect the demands of the execution environment. Protean code is a co-designed compiler and runtime system built on top of LLVM [100]. At compile time, the program is prepared by a compiler pass that virtualizes a selected subset of the edges in its control flow and call graphs, providing hooks through which the runtime system may redirect execution. This novel mechanism allows the runtime system, including the dynamic compiler, to operate asynchronously while the application continuously runs. The compiler also embeds a copy of the program's intermediate representation (IR) into the data region, to be utilized by the runtime compiler for rapidly performing rich analysis and transformations on the program. The protean code runtime monitors all running programs on the system, generating and dispatching specialized program variants that are tailored to the particular conditions detected on the system at any given point in time. Protean code addresses the goals described in 1.2 in the following ways:

1. **Low Overhead** — Diverting program control flow through selectively virtualized points introduces near-zero (<1%) overhead and provides a seamless mechanism through which the runtime compiler introduces new code variants as they become ready.

2. **General and Flexible** — To enact optimizations, protean code requires no support from the programmer or any specialized hardware. The design of protean code optimizations is in the purview of compiler writers, and protean code can be deployed for large applications on commodity hardware.

3. **Transformation Power** — Protean code embeds the IR into the program at compile time, which in turn is used by the runtime compiler as the starting point for analysis and optimization. Using the IR gives the runtime compiler the flexibility of a static compiler in terms of the analysis and optimization options that are available.

4. **Continuous Extrospection** — The protean code runtime uses program counter samples along with inter- and intra-core hardware performance monitors to detect changes to both *host* and *external* applications co-located on a single machine. This approach allows the runtime to react to highly dynamic environments by revisiting compilation choices introspectively as program phases change or extrospectively as the environment changes.

1.3.2 Increasing Server Utilization in Datacenters

With the protean code mechanism in place, we design *Protean Code for Cache Contention in Datacenters* (PC3D), an approach that generates and deploys code transformations to change how applications consume shared cache resources. To tune cache occupancy based on dynamically changing system conditions, PC3D monitors changes in the behavior of the host program and its external co-running applications via a lightweight co-phase[2] analysis scheme. PC3D reacts to co-phase changes by using the online code transformation capability to generate, dispatch and evaluate code variants to discover how to mix cache pressure reduction transformations with napping in order to both meet the QoS of high-priority applications while maximizing the performance of low-priority applications. The search through the set of transformations is accomplished via a carefully designed greedy search algorithm that searches available code transformations to increase and decrease cache occupancy while en-

[2]A co-phase is defined as the combination of the currently running phases among a program and its co-runners.

forcing QoS targets.

1.3.3 Input Responsiveness in Approximate Computing

This work introduces *Input Responsive Approximation* (IRA), an approximation approach that leverages these insights to dynamically and automatically configure the approximation options for each problem input, including selecting which code regions to approximate and how to tune the approximations within those regions. IRA achieves this by creating a *canary input* — a much smaller representation of the full input — at the outset of the problem. The canary input is used to dynamically predict the accuracy and speedup characteristics of the full input for a number of approximation options, then to dynamically choose the fastest option that achieves the desired level of accuracy.

1.3.4 Resisting Code Reuse Attacks

On top of the protean code mechanism, we design *ProtOS*, an operating system architecture that overcomes the challenge of re-randomizing application with low-overhead to give the system a secure, efficient, transparent and robust mechanism for transforming running native applications. We build the dynamic compiler into the OS itself, which functions as a transparent service that operates on running programs. This design confers the security advantages of an OS service, in that critical structures in the dynamic compiler are as safe as other operating system services. Using the ProtOS system architecture, we carefully design a novel service for *continuous code re-randomization* that constantly re-positions and reorganizes the code of running programs throughout execution. While a conventional system architecture leaves code locations fixed throughout execution, the code re-randomization service in ProtOS continuously iterates over the code in the program to generate re-randomized variants of program code. Having an online code transformation capability allows ProtOS to

choose (and re-choose) from a range of code transformation techniques and select how to manage the security vs. overhead tradeoffs offered by each. Our prototype implementation supports mixes of two re-randomization strategies – *medium-grain* re-randomization that relocates functions without changing their structure, and *fine-grain* re-randomization that additionally randomizes the order of basic blocks within functions.

1.4 Summary of Contributions

This dissertation introduces a novel low-overhead online code transformation technique and leverages the online code transformation technique to design best-in-class solutions to two important problems in modern computer systems. A summary of the specific contributions are as follows:

- **Low-overhead Online Code Transformations** — We describe protean code, a fully functional co-designed compiler and runtime system for enacting general purpose online code transformations. We evaluate protean code on a real system for the SPEC [80] benchmarks, showing that is has an average overhead of less than 1% when the dynamic compiler runs on a separate core from the application (Chapter III).
- **Datacenter Server Utilization** — We describe Protean Code for Cache Contention in Datacenters (PC3D), a dynamic approach to mitigating cache pressure in software via online compiler transformations. This approach includes a runtime search algorithm that allows for the rapid discovery of an effective set of code transformations for cache pressure reduction, as well as how elements of the search generalize to other classes of online compiler transformations. We evaluate PC3D on a real system for a set of CloudSuite [66] webservice workloads, SPEC [80] and PARSEC [23] benchmarks, and SmashBench [115] microbenchmarks. We also perform an analysis of how deploying PC3D can impact energy and server provisioning

requirements within full-scale datacenters. Our results show that PC3D improves datacenter utilization by up to 2.9x and an average of 1.5x over the state-of-the-art software contention mitigation technique across a range of workloads, while meeting 98% QoS targets for high-priority latency-sensitive applications (Chapter IV).

- **Input Sensitivity in Approximate Computing** – we perform a thorough study to demonstrate the extent to which approximation accuracy can depend on problem input, showing also that approaches used in prior work to conservatively target worst-case behavior sacrifice the full performance potential of approximation (Chapter V).
- **Input Responsive Approximation** – we introduce Input Responsive Approximation (IRA), a framework for automatically configuring approximation for every input supplied to a problem. IRA determines where to approximate, automatically selecting which code regions are most amenable to approximation for each input, as well as configuring the approximation within those regions to the fastest configuration that meets a specified accuracy bound (Chapter V).
- **Code Transforming Operating System** — We investigate the advantages of a system architecture that changes the abstraction between operating systems and compiler technologies by including a code transformation capability in the operating system. Based on this investigation we introduce ProtOS, the first system architecture to host a service that provides a full-featured code transformation capability (Chapter VI).
- **Resisting Code Reuse Attacks** — We describe the design and implementation of a novel service built using the ProtOS system architecture that implements continuous code re-randomization, a transparent, low-overhead technique for undermining code reuse attacks. We perform a thorough investigation of the resource overhead and security features of our system on a spectrum of applications, including CPU and memory intensive applications that stress the performance as-

pects of our design. Our investigation demonstrates that program code can be re-randomized at a frequency of 300ms while imposing an average of 9% overhead across a wide range of applications, providing protection against state-of-the-art code reuse attacks that have been shown in recent literature with low enough overhead to be deployed in production systems (Chapter VI).

CHAPTER II

Background and Related Work

In this Chapter, we give background and survey the related literature to the topics covered in this dissertation. These include prior efforts in building online code transformation mechanisms, as well as techniques that have been used to enable co-location to improve datacenter server utilization and protect programs from code reuse attacks.

2.1 Online Code Transformations

The study of online code transformations is an important problem in the compiler research and development communities because this class of techniques has been shown to be useful in a number of problem domains [16, 17, 29, 40, 41, 59, 91, 92, 98, 107, 109, 143, 152, 164, 169, 170, 173, 184, 190]. However, there are several limitations that prevent the wide adoption of online code transformation techniques in production environments. These limitations include high runtime overhead, dependence on programmer support or specialized hardware, limitations on the available transformations, or inability to react to dynamic execution environments.

Many online code transformation techniques are based on *full virtualization*, where program execution is tightly controlled by the dynamic compilation infrastructure. In this model, the dynamic compiler acts as a shepherd to the program, taking control of

its execution frequently (usually at branches or other selectively-chosen control-flow operations) to dynamically compile its upcoming execution paths [16, 30, 40]. The main limitation of this class of techniques is that they impose high overhead. Due to the frequency of the dynamic compiler's intervention in the program's execution, the program spends a significant amount of time waiting for the dynamic compiler to produce and optimize code, and thus they have significant runtime overheads. Despite the effort put into minimizing this performance impact [16, 110], the state-of-the-art system from among this class of techniques still carries overheads of 10-30% [30], depending on the characteristics of target programs.

Others have proposed dynamic compilation techniques that uses hardware assistance to minimize overhead [109]. While these have produced interesting optimizations, they cannot generalize to most of the commodity hardware in use today. Others have proposed dynamic compilation techniques that apply rules or heuristics written by the programmer to determine how to generate code and undertake optimizations [12, 47, 61, 73, 111, 173], however these techniques require significant programmer support on a program-by-program basis.

A characteristic of most approaches to dynamic compilation systems is that they are designed to operate on a program's machine code, hoisting the machine code into an intermediate representation before applying transformations and optimizations. This approach leads to a loss of information needed to perform the full range of program transformations available to a static compiler as it operates on the program, such as variable types and other semantic information, limiting the flexibility of the dynamic compiler [44].

This dissertation describes an approach that overcomes these limitations by (1) keeping the program's intermediate representation along with the machine code used to implement the program, giving it the analysis and transformation capabilities of a static compiler, and (2) decoupling the execution of the dynamic compiler and the

program to allow the program to continuously execute at near-native speeds. This approach is described in detail in Chapter III.

2.2 Managing Shared Resources for Co-location

Prior work has pointed out that one of the keys to achieving high utilization in datacenter servers is to enable co-location, a technique by which multiple applications simultaneously run alongside one another on the same server to increase the utilization of that server [115, 190]. However, the challenge in running applications together on the same server is that they may contend for shared server resources such as disk, network, memory and caches, resulting in unsatisfactory performance in one or both of the applications. The primary challenge for enabling co-locations in the datacenter is guaranteeing the Quality of Service (QoS) of user-facing, often latency-sensitive, applications while they are co-running with other applications.

2.2.1 Predicting Safe Co-locations

In response to this challenge, there have been several approaches proposed in the literature [54, 114, 179, 187] to predict when co-locations are *safe* - that is, to predict which co-locations will eliminate or minimize application quality of service (QoS) violations and act on those predictions when scheduling jobs to machines. Unfortunately, these predictably safe co-locations may not always be available and it is difficult to encapsulate dynamic conditions into such predictions.

2.2.2 Dynamically Enabling Co-locations

To address this limitation, techniques have been proposed to make co-locations safe by dynamically throttling down the execution of low-priority applications by continuously introducing 'naps' of varying lengths and granularities into application execution [170, 180]. This approach has the effect of alleviating the pressure an ap-

plication places on shared server resources, which allows high-priority applications to consume a larger share of resources to meet their QoS targets. However, as we show in this work, applying naps is an overly blunt instrument and results in lower throughput than necessary to enforce QoS.

Another technique, cache partitioning, has been used to explicitly control cache resource allocations, mitigating cache interference among co-running applications to ensure co-location safety. Hardware-based partitioning [48,94,135,151] allows for fine-grain control on the assignment of partitions, however it requires customized hardware and therefore has not been deployed in production systems. Software-based cache partitioning has been enacted with page coloring [107, 163, 168, 185], which controls the parts of cache an application can access via its page assignment in the operating system. Unfortunately, dynamically changing an application's cache allocation incurs significant performance penalties due to the overhead of page migration [185]. In addition, page coloring cannot be used in the presence of large pages [107, 168].

2.2.3 ISA Support For Temporal Locality Hints

The importance of quantifying and managing cache contention has been shown by prior work [17, 91, 98, 107, 143, 152, 164, 169, 170, 190]. Temporal locality hints for memory accesses can be exploited to alleviate the pressure an application puts on the shared memory subsystem. Support for these hints is available across a broad range of instruction set architectures [67], including the modern high-performance platforms that appear in datacenter servers such as x86 [3] and ARMv8 [2].

Temporal locality hints can be employed in software to suggest how data should be cached. On the x86 instruction set architecture (ISA) family, the `prefetchnta` instruction hints to the microarchitecture that data should be prefetched in a way that minimizes cache pollution. The motivating premise behind supporting these hints is that there are cases where software can identify and take advantage of the fact that

a memory access lacks temporal locality – that is, it is likely to be evicted from cache before being used again. By hinting to the microarchitecture that a memory access lacks temporal locality, it may avoid evicting other, more useful data from the cache. In this dissertation, we leverage online code transformations to strategically insert temporal locality hints to dynamically change the cache pressure an application places on its co-runners.

2.3 Approximate Computing

There are many approaches for trading result accuracy for decreased execution time or energy, based on some combination of programmers [31], runtime systems [15,83,165], programming languages [8,148], middleware [5,69], compilers [147], and hardware [6, 62,62,75,108,126,149,181].

2.3.1 Approximation in Software

Some approaches to software-based approximation use formal analysis to provide worst-case guarantees [32, 33, 123, 153], while others use calibration offline [8, 84, 122, 124] or at runtime [5, 146, 147] to guide approximation. Others have proposed software [76,142] and hardware [96] systems to catch highly inaccurate approximations early in their execution. SAGE [147] uses a dynamic calibration interval coupled with steepest ascent decisions based on the result accuracy. Another body of related research analyzes the accuracy or robustness of programs in the event of faults [104, 106, 161] or uncertain input data [28, 150], which has been used to locate code regions to approximate or bound the accuracy of approximate computation [32,33,36,124].

Approximation has been performed by decreasing the number of iterations or tasks executed [84,116,159] or by replacing exact operations with less accurate versions [122,153]. One such replacement strategy is to relax synchronization in parallel

architectures [138, 147, 172]. Misailovic et al. [122] replace loops with parallel loops. ApproxHadoop [69] is an influential recent work that leverages statistical techniques to provide accuracy guarantees when applying approximation to MapReduce applications. Branch and data herding [153] eliminate warp divergence in GPGPUs, selecting the most common branch or memory access for the entire warp. Compilers and frameworks have been used to facilitate selecting between multiple programmer-supplied implementations [8, 55, 165, 189]. Loop perforation was used by Hoffmann et al. [84] and can incorporate extrapolation to correct bias in the result [140], similar to the work on task skipping by Rinard [139]. Discarding tasks is a similar method to loop perforation, but items in a queue are discarded rather than iterations in a loop [141].

2.3.2 Approximation in Hardware

In hardware, Yeh et al. [181] design an FPU with dynamic precision that uses resource sharing, trivialization, and memoization. Approximation was applied to non-volatile memory by Sampson et al. [149] and to volatile memory by Liu et al. [108]. EnerJ [148] is a language extension with a type system for approximate variables. Operations on these variables are carried out on the approximate logic and storage of specialized hardware. The language was applied to Truffle [62], a generalized architecture designed to support approximation at the instruction level. Running code to neural processing units (NPUs) was explored by Esmaeilzadeh et al. [63] and more recently by others [75, 126]. NPUs have also been designed using limited-precision analog components [6]. Online quality management using specialized hardware has been proposed by Khudia et al. [96].

Each of these solutions requires custom hardware to achieve improvement. IRA, on the other hand, is software-based and fully realizable on commodity hardware.

2.4 Code Reuse Attacks and Defenses

Early program subversion attacks were based on injecting malicious code into program memory then executing the injected code, resulting in widespread adoption of the `W` $\oplus$ `X` paradigm[1] in modern systems. `W` $\oplus$ `X` dictates that memory can either be writable or executable, but never both, and thus is also called Data Execution Prevention (DEP). So important is the `W` $\oplus$ `X` mechanism that many modern CPU implementations have begun support `W` $\oplus$ `X` via per-page execute disable permission bits [7, 10, 90]. The ubiquity of this paradigm has proven prohibitive to *code injection attacks*, spurring a shift in the focus of attackers to reusing executable code that already exists in the address space of the program. These *code reuse attacks* are challenging to defend against. After all, programs must be able to execute code to perform useful tasks. However, this useful code can be subversively reused by an attacker to achieve their own ends, and in principle the code that proves to be useful to attackers are limited only by their creativity.

A code reuse attack is enacted by first revealing the contents of execute-enabled memory to discover where a useful code sequence resides, followed by some method of redirecting execution to that location. This is often achieved by writing the address of that code sequence onto the stack to cause the program to return to it. Combined with the wide variety of powerful system interaction capabilities available in `libc` (e.g., changing memory protections or modifying file state) and the fact that nearly every Unix program links against it, these attacks came to be known as `return-to-libc` attacks. However, this classical model of `return-to-libc` attacks is somewhat limited, as it depends on the just the right sequence of instructions being present in the program.

[1] We use the nomenclature `R`, `W` and `X` to refer to memory read, write and execute permissions, respectively.

2.4.1 Return-oriented Programming

Shacham [157] extended the `return-to-libc` attack, demonstrating the feasibility of chaining together short sequences of bytes ending in return instructions – *gadgets* – from disparate locations in memory to execute arbitrary functionality on behalf of the attacker. Because control is passed from gadget to gadget using return instructions, this technique was dubbed a *return-oriented programming (ROP) attack*. The basic sequence of steps in executing a ROP attack is as follows:

1. **Locate Gadgets** — the attacker first gains visibility into the contents of executable program memory to discover where useful gadgets reside. This can be done via static analysis, side channels, or other information leaks such as memory disclosure bugs. Memory disclosure bugs that reveal the contents of memory are commonplace in production applications, such as in the OpenSSL Heartbleed vulnerability that affected as many as 55% of popular HTTPS websites [58]. Gadgets need not be sequential or even near one another, as control can be passed from gadget to gadget via the return instructions at the end of each gadget.

2. **Construct and Deliver Payload** — gadgets are chained together as primitive building blocks that produce some higher-order functionality that is of use to the attacker (e.g., setting up arguments and making a system call). Once the gadget chain is constructed, a payload containing the gadget addresses is placed onto the stack.

3. **Hijack Execution** — the execution of the gadget sequence can be triggered by hijacking the flow of execution (e.g., by overwriting a function pointer) to the first gadget. Once the first gadget is executed, control returns to the second gadget, which executes then returns to the third gadget, and so forth, until the functionality desired by the attacker has been executed in full.

ROP is the state-of-the-art technique in subverting program execution, and has produced a wide body of academic literature dealing with ever more sophisticated attack and defense techniques in recent years [25,27,34,37,42,51,52,68,70,82,89,131, 154,156,157,166,167,174,178,182,183]. Moreover, it was recently reported that ROP was used in 95%+ of the known Windows exploits over the last two years [136] and in attacks on a number of sensitive industrial firms [60].

ROP was originally conceived by Shacham [157], where he argues that any program linking against `libc` likely contains a rich enough set of ROP gadgets to achieve arbitrary functionality. Others have extended ROP and developed automated tools for identifying and constructing useful gadgets chains for ROP exploits [25, 56, 89, 154]. Checkoway et al. [38] demonstrate an attack on a widely used voting machine. Checkoway et al. [37] and Bletsch et al. [27] generalize ROP to circumvent stack-based defense mechanisms by constructing gadgets without the use of return instructions. Recently, in response to the popularization of various forms of Address Space Layout Randomization (ASLR), Snow et al. [162] demonstrate an exploit technique based on memory disclosures to map a program's randomized address space at runtime. Seibert et al. [156] present side-channel methods that an attacker can use to learn about code locations. Others have shown ROP attacks to exploit vulnerabilities in popular applications, including Adobe Flash, Mozilla Firefox and Internet Explorer [132,167].

2.4.2 ASLR and Its Limitations

One of the key components of a modern code reuse attack is in obtaining visibility into the executable bytes in program memory. With access to the program binary file, gaining this visibility into the program's code is trivial because the binary file contains a layout and description of where in memory those bytes will eventually reside. Similarly, code from dynamically-linked libraries loaded at fixed or predictable locations in memory are easily predicted and usable by attackers. This has resulted in the

widespread adoption of *coarse-grain Address Space Layout Randomization* (ASLR), a countermeasure that randomizes the base address of segments such as executable and library code, stack, and heap at each program invocation so that their exact positions are not known in advance [21, 77, 82, 88, 97, 130, 174, 182].

However, randomizing segment positions has been shown to be a weak defense because the locations of large swathes of bytes become visible when the location of anything in the segment is leaked. For example, a leaked function address effectively gives away all of the function's bytes because the function's structure is known in advance. Further extensions to ASLR randomize the locations of functions in the segment, known as *medium-grain* ASLR. Still others, dubbed *fine-grain* ASLR, randomize the locations of basic blocks within functions [174], the locations of instructions in blocks [82], or the contents of instructions themselves [130]. These techniques raise the bar and make attacks more difficult, however recent work has shown that even with the strongest forms of ASLR that statically change the content [130] and locations [82] of instructions, repeatedly exploitable vulnerabilities can be used by attackers to construct a map of memory on-the-fly, eventually allowing the attacker to find a sufficient number of gadgets to construct a ROP attack [156, 162].

2.4.3 Other Defense Mechanisms

Code reuse, and ROP in particular, are commonly used in real-world exploits [60, 136], and are an active area of literature. In response to the proliferation of these attacks, several other classes of defenses have emerged.

Control-Flow Integrity (CFI). CFI, first introduced in 2005 [4], thwarts attacks by preventing deviations from a program's intended control flow. CFI implementations in compilers and binary instrumentation strengthen control flow statically [105, 129] or dynamically [39, 52, 129, 175]. Others use sophisticated software and hardware mechanisms to strengthen protection and reduce overhead for CFI [26, 42,

131, 178, 182, 183]. Recently, Arther et al. proposed removing indirect branches from the ISA to enforce CFI [11]. However, CFI enforcement mechanisms are difficult to generalize beyond limited classes of attacks, are difficult to integrate into production systems [125, 182] or introduce significant runtime overhead [39, 52]. Moreover, recent attacks have compromised state-of-the-art CFI mechanisms [34, 70].

kBouncer [131] and ROPecker [42] are two recently proposed, highly effective runtime defense mechanisms against ROP. These works leverage hardware registers to capture anomalous behaviors that often lead to the invocation of sensitive system calls. However, the defense capabilities of these approaches can suffer from insufficient storage for monitoring long history and it rely heavily on the assumption that ROP gadgets behave in certain patterns.

Continuous Re-randomization. Others have recognized value of runtime re-randomization in thwarting code reuse attacks, while also recognizing that it is challenging to design re-randomization with low overhead [162, 174]. Guiffrida et al. re-randomize microkernel code and data to protect against kernel exploits [68], re-randomizing every 1 second with 40-50% overhead. Re-randomization in ProtOS focuses on efficiently re-randomizing native application code. Such techniques for protecting the OS are valuable and complementary to our work, protecting kernel structures while we protect userspace applications.

CHAPTER III

Protean Code: Low-overhead Online Code Transformations

This chapter introduces *protean code*, a novel approach for enacting arbitrary compiler transformations at runtime for native programs running on commodity hardware with negligible (<1%) overhead. The fundamental insight behind the underlying mechanism of protean code is that, instead of maintaining full control throughout the program's execution as with traditional dynamic compilers, protean code allows the original binary to execute continuously while the dynamic compilation mechanism works asynchronously and in parallel to the running application. In this approach, the dynamic compiler diverts control flow of the application only at a set of virtualized points, allowing rapid and seamless rerouting to the newly transformed code variants produced by the dynamic compiler. In addition, the protean code compiler embeds the compiler's intermediate representation (IR) with high-level semantic information into the program, empowering the dynamic compiler to perform rich analysis and transformations online with little overhead.

3.1 Protean Code

Protean code is a novel online code transformation system designed to address the challenges that prevent the wide adoption of traditional dynamic compilation techniques in production environments; it is low-overhead, requires no customized hardware, has the transformative capabilities of a full-fledged compiler, and can make compilation decisions based on the behavior of all running applications. Protean code consists of a co-designed compiler and runtime system, termed the *protean code compiler* (`pcc`) and the *protean code runtime*.

Presented in Figure 3.1, `pcc` readies the host program for runtime compilation by making two classes of changes to the program. First, it virtualizes a subset of the edges in the program's control flow and call graphs. These virtualized edges then serve as points in the program's control flow at which the runtime system may redirect execution. Second, the compiler embeds several metadata structures, including an Edge Virtualization Table (EVT) and intermediate representation of the program, within the program's data region, which are used to aid the runtime system in dynamically introducing new code variants into the running program.

Shown in Figure 3.2, the protean code runtime is responsible for monitoring a host program and its external execution environment in order to dynamically generate and dispatch code variants when needed. The runtime system first initializes by attaching to the program, discovering the program metadata and setting up a shared code cache from which the program can execute new code variants. To generate and dispatch a code variant, the runtime compiler, an LLVM-based compiler backend, leverages the IR. The new code variant is then inserted into the code cache and dispatched into the running host program by the EVT manager. During host program execution, the lightweight monitoring component of the runtime detects changes in both the host program phases and the external environment, including co-running

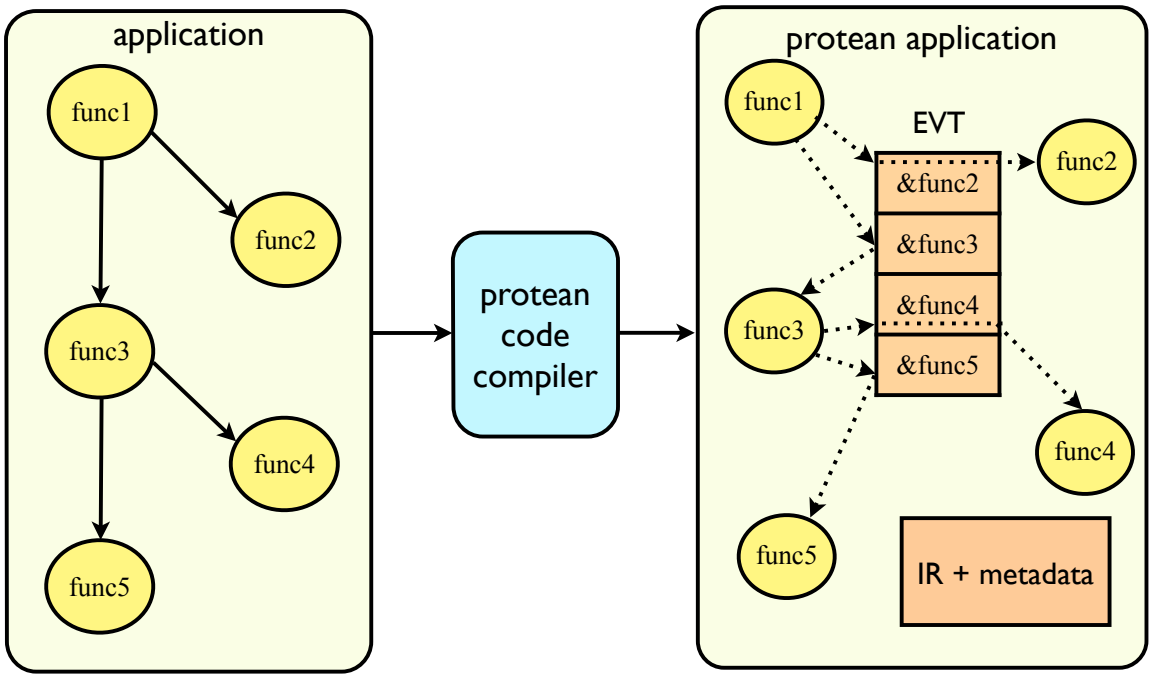


Figure 3.1: Overview of the protean code compiler

applications, using samples of program counters and hardware performance monitors. In response to phase and environment changes, a decision engine determines when and how to generate new code variants and selects the appropriate variant for the current execution phase.

3.1.1 Design Principles

The primary goal of protean code is to provide a dynamic code transformation solution that is deployable in production environments and is powerful enough to enable techniques such as the PC3D runtime described in Chapter IV and the continuous code re-randomization technique described in Chapter VI. There are three principles used in the design of protean code to realize this goal:

1. Maintaining absolute control of the program throughout execution, as in traditional dynamic compilers such as Dynamo [16] and DynamoRIO [30], leads to high overhead. Protean code instead allows the original binary to continuously

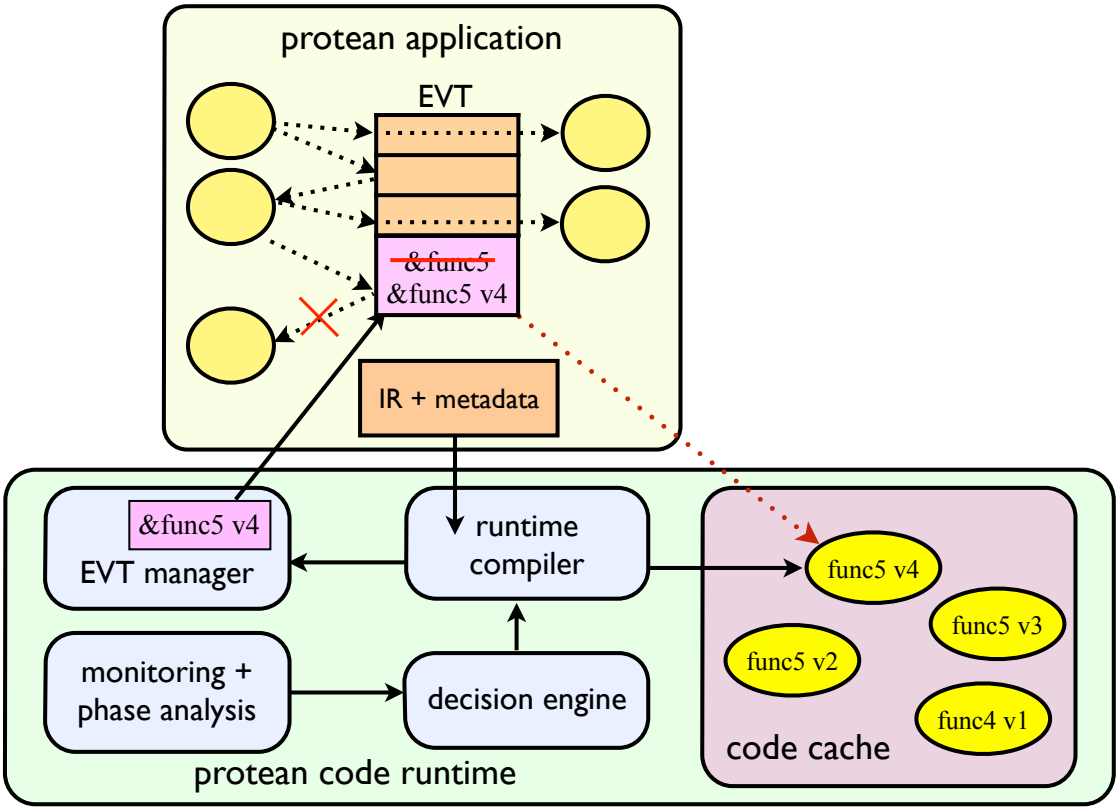


Figure 3.2: Overview of the protean code runtime

execute and diverts the program control flow at a set of virtualized points, introducing negligible overhead. The runtime compiler is invoked asynchronously at controllable granularity, which also limits the overhead.

2. Many traditional dynamic compilers hoist the native machine code into an intermediate format at runtime to perform analysis and transformation [16, 59, 109, 112, 155], leading to overhead and the loss of rich semantic information present in IR from the static compiler [44]. Protean code embeds the IR into the program binaries, allowing the dynamic compiler to perform powerful analysis and transformations online with little overhead.

3. Protean code requires no support from the programmer or any specialized hardware, allowing it to be seamlessly deployed for large applications on commodity hardware. It leverages hardware performance monitors for lightweight monitoring, phase analysis and transformation selection, further minimizing overhead. A useful property of the application binaries produced by `pcc` is that they can be run without the runtime system, incurring negligible extra runtime overhead. In addition, once compiled with `pcc`, any protean code runtime can be used. These are particularly useful features in modern production environments, where rapidly changing conditions may dictate applying different classes of optimizations in the pursuit of different objectives to the same application binary.

3.1.2 Protean Code Compiler

The Protean Code Compiler (`pcc`) readies the host program for runtime compilation by (1) virtualizing control flow edges and (2) embedding meta-data in the program binary.

3.1.2.1 Control Flow Edge Virtualization

`pcc` adds a compiler pass to convert a subset of the branches and calls in the program from direct to indirect operations. By virtualizing a subset of edges, `pcc` sets up those edges as points in the programs execution where its control flow path may be easily altered by the protean code runtime to route program execution to an alternate variant of the code.

There are some important considerations to be made when selecting which edges to virtualize. Selecting too many edges or edges that are executed too frequently may result in unwanted overheads because indirect branches are generally slightly slower than direct branches (the causes of this overhead are discussed shortly). On the other hand, selecting only edges that are rarely executed risks introducing large gaps in execution during which new code variants are not executed. Our current approach to selecting edges is to virtualize only function calls, and only those where the callee function has more than one basic block. We find that this approach works well in practice, resulting in negligible overhead while ensuring that execution is promptly routed to the new code variants.

Edge Virtualization Overheads. Protean code contains virtualized control flow edges to allow the runtime to redirect execution as those edges are executed. There are three sources of possible overhead that arise from edge virtualization. First, it may lead to increased cache/memory activity. Because the EVT may be updated at any point by the runtime, it is treated as volatile and its entries must be loaded from memory at each use. Third, the edge virtualization table resides in memory, which may impact program load time and memory memory footprint. Third, on certain platforms indirect branch and call instructions use more space than direct branches and calls. For example, direct and indirect calls with a 32-bit operand on x86_64 are 5 and 6 bytes long respectively. This may put slightly higher pressure on I-cache and decode resources in the CPU. We evaluate the overhead of edge virtualization

in Section 3.2, where we observe that the overhead of edge virtualization is small in practice ($< 1\%$ on average).

3.1.2.2 Program Metadata

Two types of program metadata are used by the protean code runtime to rapidly generate and dispatch correct, alternate code variants at runtime.

Edge Virtualization Table (EVT). A structure called the EVT contains the source and target addresses of the edges virtualized by `pcc`. The EVT is the central mechanism by which execution of the program is redirected by the runtime. To change execution, the runtime simply rewrites target addresses in the EVT to point to the new code variant.

Intermediate Representation (IR). `pcc` serializes, compresses and places the compiler's intermediate representation (IR) of the program into its data region, which the runtime decompresses then deserializes, leveraging it to perform analysis and transformations. Having direct access to the IR yields two significant advantages. First, it allows the runtime to avoid disassembling the binary, which can be difficult or impossible without access to fine-grain information about the executing code paths [102, 127]. Second, the alternative of hoisting the binary to IR, as is done in prior work, loses important semantic information and limits the flexibility of the compiler [44]. As an example of the utility of the IR, in this work PC3D gleans loop structure and nesting depth from the IR and uses that information to guide compilation decisions.

3.1.3 Protean Code Runtime

The protean code runtime is a set of mechanisms that work together to generate and dispatch code variants as the host program executes.

3.1.3.1 Runtime Initialization

Operating on an executable prepared by `pcc`, the runtime process begins by attaching to the program. It first discovers the locations of the structures inserted by `pcc` at compile-time, including the EVT and the IR. It then initializes a code cache, used to store the code variants generated by the dynamic compiler. Finally, because the EVT and code cache are structures that are shared between the program and the runtime and may be frequently accessed, the runtime sets up a shared memory region via an anonymous `mmap` to encompass both structures.

3.1.3.2 Code Generation and Dispatch

The runtime generates and dispatches code variants into the program asynchronously. When a new variant of a code region is requested, the dynamic compiler leverages the IR of the code region to generate the new variant. Once a new code variant has been generated, it is placed into the code cache. The EVT manager then modifies the EVT so that the target of the corresponding virtualized edge is the head of the newly minted variant in the code cache. The EVT update is a single atomic memory write operation on most modern platforms, and thus requires no synchronization between the host program and the runtime to work correctly.

Throughout these actions of the runtime process, execution of the program proceeds as normal until control flows through the virtualized edge, at which point control reaches the new code variant.

3.1.3.3 Monitoring, Phase Analysis and Decisions

The runtime supports both *introspection*, monitoring changes in the host program, and *extrospection*, monitoring changes in the execution environment. Based on this monitoring, the runtime makes decisions and adapts to changing system conditions such as application input/load fluctuation, starting or stopping of co-running appli-

cations, and phase changes among both the host programs and external programs.

Introspection. For host programs, the runtime identifies hot code regions by sampling the program counter periodically through the `ptrace` interface. The runtime then associates the program counter samples with high-level code structures such as functions, allowing the runtime to keep track of which code regions are currently hot, as well as how hot regions change over time.

To identify phase changes, the runtime also leverages hardware performance monitors to track the progress of the program. Phases are defined in terms of the hot code identified by program counter samples described above as well as by the progress rate of the running applications using metrics such as instructions per cycle (IPC) or branches retired per cycle (BPC). Since hardware performance monitors are ubiquitous on modern platforms and can be sampled with negligible overhead, this approach allows the runtime to conduct phase identification in a manner that is both lightweight and general across hardware platforms.

Extrospection. Similarly, for other external programs, the runtime can optionally track program progress and identifies phase changes via hardware performance monitors. Microarchitectural status and performance, using metrics such as cache misses or bandwidth usage, are also tracked through the performance monitor interface. Additionally, the runtime can be configured to use application-level metrics reported through application-specific reporting interfaces, such as queries per second or 99th percentile tail query latency for a web search application.

Dynamic Transformation Decisions. The decision engine determines (1) when to invoke the dynamic compiler, (2) what transformations to apply, and (3) which variant to dispatch into the running program. The policies guiding the decision engine depend on objective of the runtime system (e.g., the optimization and security applications shown later use their own decision engines). Note that these policies can be designed to carefully control how often compilation occurs to limit the overhead

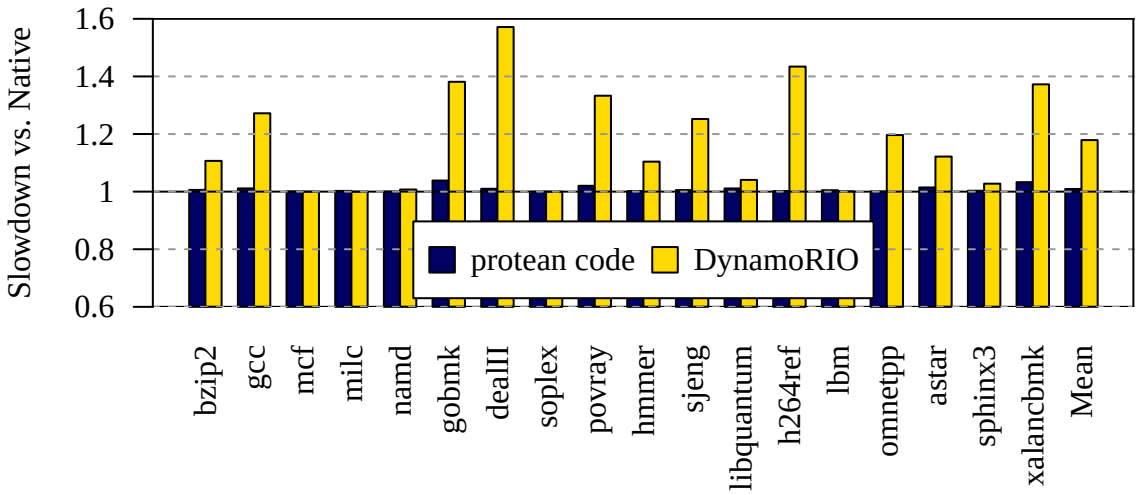


Figure 3.3: Dynamic compiler overhead when making no code modifications (normalized to native execution)

introduced by running the dynamic compiler alongside the application.

3.2 Performance Investigation

We now perform an investigation of the runtime overhead of the protean code mechanism. The protean code static compiler and runtime compiler are implemented on top of LLVM version 3.3. When compiling protean code or non-protean code benchmarks, compilation is done with `-O2`. All experiments are performed on a quad core 2.6GHz AMD Phenom II X4 server. We use the SPEC CPU2006 benchmark suite [80]; as LLVM does not natively support Fortran, our prototype implementation does not handle Fortran and so we focus our evaluation exclusively on the C and C++ benchmarks in SPEC CPU2006.

3.2.1 Virtualization Mechanism

First we investigate the baseline cost of virtualizing execution with protean code and compare this cost with that of virtualizing execution with DynamoRIO [30]. DynamoRIO is a state of the art binary translation-based dynamic compiler, chosen

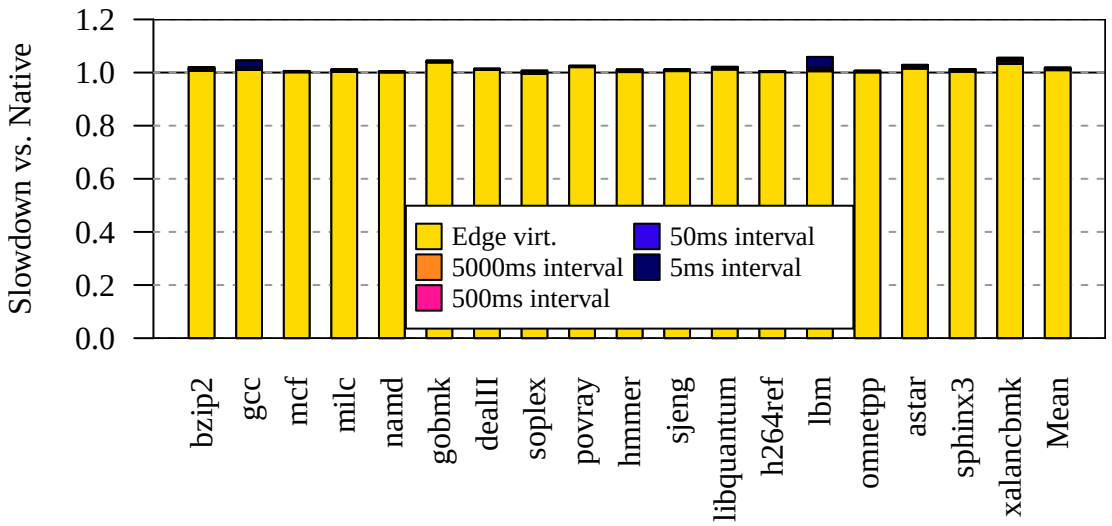


Figure 3.4: Dynamic compilation stress tests; compilation occurs on a separate core from the host application

as a baseline because it is a mature software project that is actively maintained and is well known for its low overhead relative to other dynamic compilers [65, 188].

Figure 3.3 shows the overhead for SPEC applications compiled as protean code relative to the non-protean code version of the benchmark. The base performance overhead of protean code mechanism is shown to be negligible, less than 1% on average. DynamoRIO, on the other hand, introduces an average of 18% overhead when performing no code modification. The primary distinction between binary translation and protean code is that protean code performs compilation asynchronously, out of the application's control flow path. Running protean code is low overhead because the application is allowed to continually execute, even when code is being compiled and dispatched. Binary translation incurs higher overhead because it requires all execution to occur from the code cache or interpreter, and thus control is continually diverted from the application back to the binary translation system.

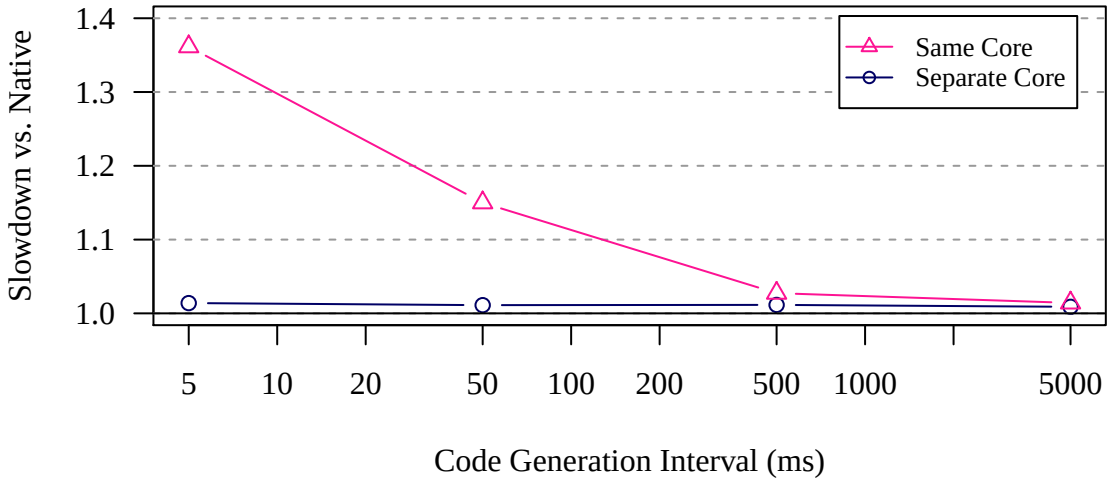


Figure 3.5: Dynamic compilation stress tests on separate vs. same core

3.2.2 Dynamic Compilation Overhead

The protean code runtime runs in its own process and performs compilation asynchronously with respect to the running host application, employing a dynamic compiler to introduce new code variants into the running host program. We next present a set of dynamic compilation stress tests to demonstrate the impact of the level of activity of the dynamic compiler. In these experiments, the host program is run with a protean runtime configured to periodically recompile randomly selected functions throughout the life of the running application.

Figure 3.4 shows the results of these experiments for the SPEC benchmarks for a range of different time intervals between recompilations, where the runtime process (including the dynamic compiler) uses a dedicated physical core. The results show that this causes the dynamic compiler to generate very little overhead to the host program, even when performing recompilation every 5ms. We note that the LLVM compiler backend uses an average of around 5ms to compile a function, so the 5ms trigger interval results in the dynamic compiler being active almost continuously. Figure 3.5 presents, for the SPEC benchmarks, the average performance overhead of

performing the same dynamic compilation stress tests, with the runtime on the same core as the host or on a separate core from the host.

While executing the runtime on a separate core introduces minimal overhead no matter how frequently code generation is performed, the overheads of performing the compilation on the same core as the host program can be significant in extreme cases where compilation is nearly continuous. In an era of multicore and manycore processors, and particularly in production environments, the common case is for cores to be heavily underutilized. For example, Google reports typical server utilization levels of 10-50% [19]. Nevertheless, in such instances where no separate core is available for the runtime, this overhead can be controlled by limiting the frequency of recompilation. As shown in Figure 3.5, the overhead of recompilation on the same core becomes negligible at 800ms.

3.3 Summary

This chapter presents protean code, a novel approach to dynamic compilation designed to be deployable for performance optimization in production environments. Protean code is nearly free of performance overhead (<1% on average), operates without any special hardware or programmer support, and has the flexibility of a robust static compiler. This combination of features gives protean code the ability to remain in place continuously to establish an online code transformation capability. Upcoming chapters will show how this capability can be leveraged to build runtime systems to improve the efficiency and security of modern systems.

CHAPTER IV

Online Code Transformations to Improve Utilization in Datacenters

Rampant dynamism due to load fluctuations, co-runner changes, and varying levels of interference poses a threat to application quality of service (QoS) and has limited our ability to allow co-locations in modern warehouse scale computers (WSCs). Instruction set features such as the non-temporal memory access hints found in modern ISAs (both ARM and x86) may be useful in mitigating these effects. However, despite the challenge of this dynamism and the availability of an instruction set mechanism that might help address the problem, a key capability missing in the system software stack in modern WSCs is the ability to dynamically transform (and re-transform) the executing application code to apply these instruction set features when necessary.

Leveraging protean code, this chapter describes *Protean Code for Cache Contention in Datacenters* (PC3D). PC3D dynamically transforms running applications to strategically insert non-temporal access hints, allowing low-priority batch applications to execute at high efficiency when running alongside high-priority latency-sensitive applications. Our results show that PC3D achieves utilization improvements of up to 2.8x (1.5x on average) higher than state-of-the-art contention mitigation runtime techniques at a QoS target of 98%.

4.1 Protean Code for Cache Contention in Datacenters

Protean Code for Cache Contention in Datacenters (PC3D) is a protean code runtime that dynamically applies compiler transformations to insert non-temporal memory access hints, tuning the pressure a host application exerts on shared caches when the QoS of an external application is threatened. PC3D is implemented entirely as a runtime system that operates on an application prepared by the protean code compiler, requiring no changes to the basic protean code compiler setup described in Section 3.1.

The goal of PC3D is to find and dispatch variants of the host program code that contain a mix of non-temporal cache hints that allows the host's co-runners to meet their QoS targets while maximizing the throughput of the host. To ensure co-runner QoS, PC3D searches through a spectrum of program variants that have varying levels of cache contentiousness. The effectiveness of interference reduction of each variant is empirically quantified online by the protean code runtime. The best-performing program variant is then dispatched and runs until a new program phase or external application sensitivity phase is detected. In cases where relying solely on non-temporal cache hints is unable to ensure QoS of the external applications, naps are mixed with cache pressure reduction as a fallback.

4.1.1 Code Variant Search Space

PC3D generates and dispatches program variants that contain a selection of non-temporal cache hints. We refer to each such program variant as a bit vector $\overline{M} = \langle M_1, M_2, ..., M_N \rangle$, where N is the number of loads in the host program's code and $M_i \in \{0, 1\}$ represents the absence or presence of a non-temporal cache hint associated with the ith load. The set of program variants of this form is the set of all possible bit vectors of length N, which has a cardinality of 2^N. Figure 4.1 shows the four variants

```
prefetchnta (%r14)          // m1
mov    %r13,%rsi
shl    $0x4,%rsi
mov    (%r14),%r8
prefetchnta (%r8,%rsi,1)    // m2
mov    (%r8,%rsi,1),%rax
```

(a) $<m_1, m_2> = <1, 1>$

```
prefetchnta (%r14)          // m1
mov    %r13,%rsi
shl    $0x4,%rsi
mov    (%r14),%r8
                            // m2
mov    (%r8,%rsi,1),%rax
```

(b) $<m_1, m_2> = <1, 0>$

```
                            // m1
mov    %r13,%rsi
shl    $0x4,%rsi
mov    (%r14),%r8
prefetchnta (%r8,%rsi,1)    // m2
mov    (%r8,%rsi,1),%rax
```

(c) $<m_1, m_2> = <0, 1>$

```
                            // m1
mov    %r13,%rsi
shl    $0x4,%rsi
mov    (%r14),%r8
                            // m2
mov    (%r8,%rsi,1),%rax
```

(d) $<m_1, m_2> = <0, 0>$

Figure 4.1: The set of variants for a small code region within `libquantum` on x86_64. Non-temporal hints and affected loads are shown in bold

Algorithm 1: Greedy search for a code variant $\overline{best}$ that uses the right mix of cache contention reduction and napping to maximize application performance

output: $\overline{best}$
/* enact/evaluate $\overline{0}$ to obtain the nap intensity ($nap_{\overline{0}}$) applied to the variant to meet co-runner QoS and the performance ($R_{\overline{0}}$) of the variant at that nap intensity */
$(nap_{\overline{0}}, R_{\overline{0}}) \leftarrow VariantEval(\overline{0}, 0, 1)$
$(nap_{\overline{1}}, R_{\overline{1}}) \leftarrow VariantEval(\overline{1}, 0, 1)$
$nap_{UB} \leftarrow nap_{\overline{0}}$, $nap_{LB} \leftarrow nap_{\overline{1}}$
$\overline{m} \leftarrow \overline{1}$, $\overline{best} \leftarrow \overline{1}$, $R_{\overline{best}} \leftarrow R_{\overline{1}}$
$i \leftarrow 1$
while $i \leq n$ *and* $nap_{LB} < nap_{UB}$ **do**
  $\overline{m} \leftarrow \langle m_1, ..., !m_i, ..., m_n \rangle$ // flip ith bit in $\overline{m}$
  $(nap_{\overline{m}} R_{\overline{m}}) \leftarrow VariantEval(\overline{m}, nap_{LB}, nap_{UB})$
  if $R_{\overline{best}} < R_{\overline{m}}$ **then**
    $R_{\overline{best}} \leftarrow R_{\overline{m}}$, $\overline{best} \leftarrow \overline{m}$, $nap_{UB} \leftarrow nap_{\overline{m}}$
  else
    $\overline{m} \leftarrow \langle m_1, ..., !m_i, ..., m_n \rangle$ // reject change
  end
  $i++$
end
return $\overline{best}$

for a small code region ($N = 2$) within `libquantum`, where each of the four variants contains a different mix of non-temporal cache hints. PC3D searches these variants using a greedy search algorithm whose complexity is $O(N)$, described in detail in Section 4.1.3. However, even with a search complexity that is linear in the number of load instructions, the number of variants may still be large. To navigate this space efficiently, PC3D employs several heuristics.

4.1.2 Variant Search Space Reduction

PC3D focuses on the loads most likely to have a significant impact on application behavior. The heuristics employed to this end are as follows:

- **Exclude Uncovered Code** — Leveraging the PC samples collected for host program phase analysis, we expect code regions that never appear in those samples

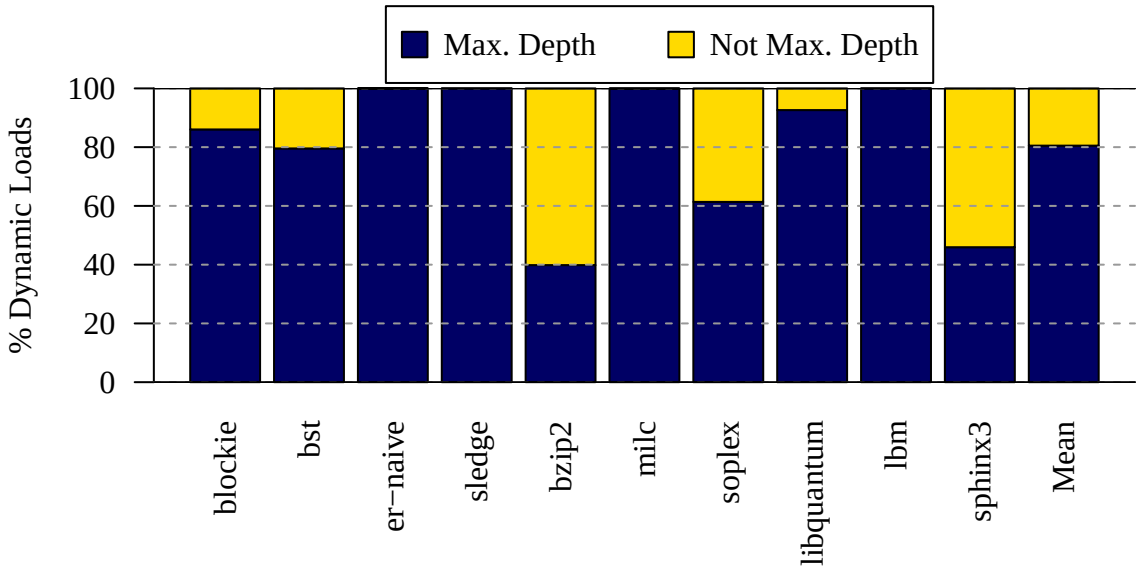


Figure 4.2: Proportion of dynamic loads in contentious applications coming from loads at maximum loop depth

to have a minimal impact on cache pressure and application performance. Therefore, the loads from regions not appearing in the PC samples are pruned from the search space prior to the search. This reduces the number of loads that must be considered by an average of 12×.

- **Prioritize Hotter Code** — Furthermore, we expect code regions appearing more frequently in the PC samples to have a higher impact. Therefore PC3D prioritizes loads from hotter code regions in the search.
- **Only Innermost Loops** — For a range of contentious applications as shown in Figure 4.2, we have observed that an average of more than 80% of the dynamic loads come from the maximum-depth loop(s) within each of the program's functions. Leveraging the program's IR, PC3D recognizes loops and their nesting depths, then prunes from the search space loads that are not at the maximum depth.

The number of static loads remaining after applying these heuristics is on average a factor of 44× smaller than the total number of static loads in the program (see Section 4.2.1).

These heuristics focus the optimization decisions made by PC3D on the most

important regions of code, a strategy we expect will also prove to be useful among other protean runtimes. After PC3D applies these heuristics, its search is limited to variants that are of the form $\overline{m} = \langle m_1, m_2, ..., m_n \rangle$, where $m_i \in \{0, 1\}$. $\overline{m}$ is a bit vector of the n loads from innermost loops among active code regions in the program phase, ordered roughly by how much impact they are expected to have on execution. For convenience, we refer to the variant where every load lacks a non-temporal hint as $\overline{m} = \overline{0}$ and its converse, the variant where every load has a non-temporal hint, as $\overline{m} = \overline{1}$.

4.1.3 Traversing the Variant Search Space

The variant search is guided by Algorithm 1. The search begins by evaluating variants $\overline{0}$ and $\overline{1}$, which are the variants that exert the most and least amount of cache pressure, respectively, out of all the variants in the search space. Because these variants are at the extremes of cache pressure, they are also at the extremes of the nap intensity required to meet co-runner QoS targets, and therefore may be viewed as lower and upper bounds, respectively, for the nap intensity that would theoretically be required to satisfy co-runner QoS for *any* program variant. As we discuss shortly, these bounds are used to limit the range of nap intensities that are evaluated for each variant, improving how quickly PC3D can converge on the right code variant.

Using $\overline{1}$ as a starting point, the algorithm steps through loads in the order of decreasing importance. For each load, the algorithm revokes the load's non-temporal hint, then calls $VariantEval$ (Algorithm 2) to enact the resulting code variant and evaluate whether that revocation improves the application's performance given the particular level of cache pressure produced by that variant along with the level of nap intensity required to allow the application's co-runners to meet their QoS targets. If the incremental change is found to have improved application performance, the change is kept and the algorithm repeats these steps on the next load. Otherwise,

Algorithm 2: $VariantEval$, evaluation of a single program variant in PC3D

input : $\overline{m}$, nap_{LB}, nap_{UB}
output: $nap_{\overline{m}}$, $BPS_{\overline{m}}$
$nap_{cur} \leftarrow (nap_{LB} + nap_{UB})/2$
$BPS \leftarrow 0$
generate and dispatch variant $\overline{m}$
while $nap_{LB} < nap_{UB}$ **do**
  set_nap_intensity (nap_{cur})
  if *QoS of co-runners is satisfied* **then**
    $nap_{UB} \leftarrow nap_{cur}$
    $BPS \leftarrow$ BranchesPerSecond $_{\overline{m}}$
  else
    $nap_{LB} \leftarrow nap_{cur}$
  end
  $nap_{cur} \leftarrow (nap_{LB} + nap_{UB})/2$
end
return (nap_{cur}, BPS)

the change is rejected and the algorithm repeats these steps on the next load.

Note that each variant accepted as the best produces more cache pressure than the previous best version. Similar to the logic that was used to establish program-wide lower and upper bounds on the nap intensity range, upon accepting a variant as the best the upper bound on nap intensity is lowered to the nap intensity of the newly accepted variant.

4.1.4 Online Evaluation of Variants

PC3D searches for program variants that improve application performance while meeting co-runner QoS. Guiding the search are empirical evaluations of a sequence of program variants, which are dispatched then evaluated against the current running set of co-runners. Each variant produces a particular level of cache contentiousness, and may need to run with a particular nap intensity to allow its co-runners to hit their QoS targets.

This concept is demonstrated in Figure 4.3, which presents the performance of

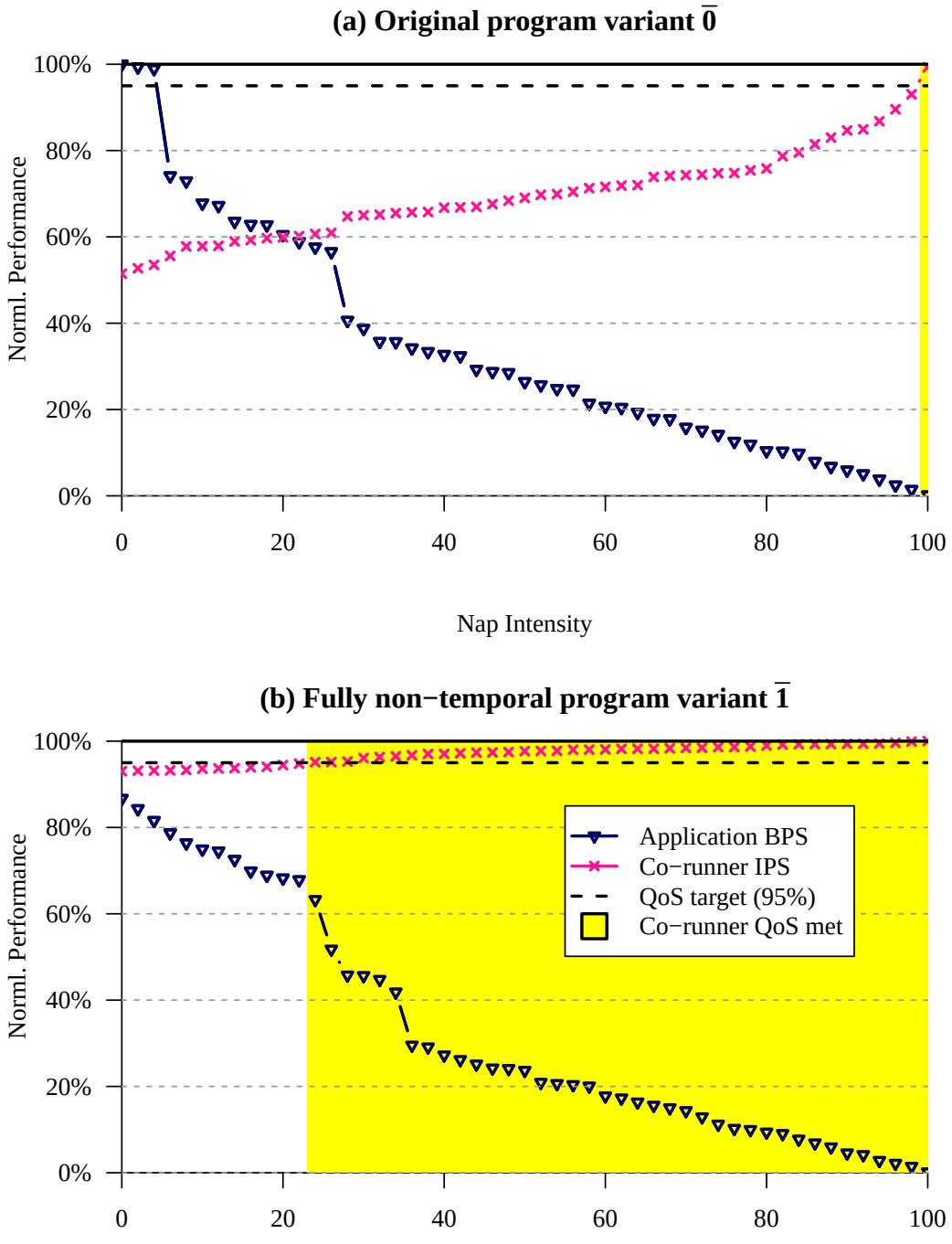


Figure 4.3: Online empirical evaluation for two variants of `libquantum` (application) running with `er-naive` (co-runner)

two variants of `libquantum` (host application) running with `er-naive` (external high-priority co-runner) as a function of the nap intensity applied to `libquantum`. Performance of `libquantum` is reported as branches per second (BPS) normalized to its BPS while running alone, while performance of `er-naive` is reported as instructions per second (IPS) normalized to its IPS running alone. We use BPS for host applications since, unlike branch counts, their static instruction counts change with the insertion/removal of non-temporal hints. As Figure 4.3 shows, each of these two variants exerts a different level of cache pressure on `er-naive`, and thus given a hypothetical QoS target of 95% for `er-naive`, a different level of nap intensity is required to allow `er-naive` to hit its QoS target. In this example, the `libquantum` variant in 4.3(a) requires a nap intensity of 99% to allow `er-naive` to meet its QoS target, while the variant in 4.3(b) requires a nap intensity of just 23%. At those respective nap intensities, the performance of variant (b) is far better than that of (a).

When evaluating a variant dynamically to discover the minimum nap intensity needed to meet co-runner QoS, PC3D need not evaluate the entire spectrum of nap intensities. The performance of both the application and its co-runners are monotonic as a function of nap intensity, so PC3D organizes the variant evaluation as a binary search over the range of nap intensities, shown in Algorithm 2. To reduce the search even further, PC3D performs the binary search only within the range of nap intensities between the lower and upper bounds established by evaluating other variants.

4.1.5 Monitoring Co-runner QoS

PC3D continuously monitors application co-runners to measure their quality of service (QoS). In this work, we use co-runner instructions per second (IPS) relative to the IPS running without the host application as a proxy for QoS. To measure co-runner IPS without the host, PC3D uses a flux approach similar to the mechanism described in [180], in which the host is put to sleep for a short period of time (40ms

in our work) and performance measurements are taken while the co-runners execute without interference from the host. We deploy one such measurement every 4 seconds, allowing the flux technique to be deployed with very little (1%) overhead.

4.2 Evaluation

Methodology The protean code static compiler and runtime compiler are implemented on top of LLVM version 3.3. When compiling protean code or non-protean code benchmarks, compilation is done with `-O2`. All experiments are performed on a quad core 2.6GHz AMD Phenom II X4 server. Applications used throughout the evaluation are drawn from CloudSuite [66], the SPEC CPU2006 benchmark suite [80], the PARSEC benchmark suite [23] and SmashBench [115].

4.2.1 PC3D Variant Search Heuristics

PC3D searches a set of program variants to arrive at a variant that improves the host application performance in the presence of some set of external applications. One of the keys to making this approach effective is to locate good code variants quickly. To accomplish this, PC3D employs several heuristics, described in detail in Section 4.1.2, to reduce the number of load instructions considered in the search. Figure 4.4 evaluates how effective these heuristics are across a set of contentious applications. Each cluster shows the number of loads that must be considered by the search as each successive heuristic is applied, normalized to the total number of loads in the application. Where there are multiple phases in a program, Figure 4.4 presents the average number of loads across all phases. Absolute counts of the number of loads that appear in each program are also included as numbers at the top of the plot.

As described in Section 4.1.2, PC3D first discards loads from uncovered code – code regions that appear to the runtime system to have never executed during the current phase. On average, discarding loads from uncovered code results in a reduction of the search space by a factor of 12×. Second, PC3D extracts loop structure from the IR and discards each load that is not at the maximum loop depth within each function.

Overall, these heuristics are effective, reducing the number of static loads exam-

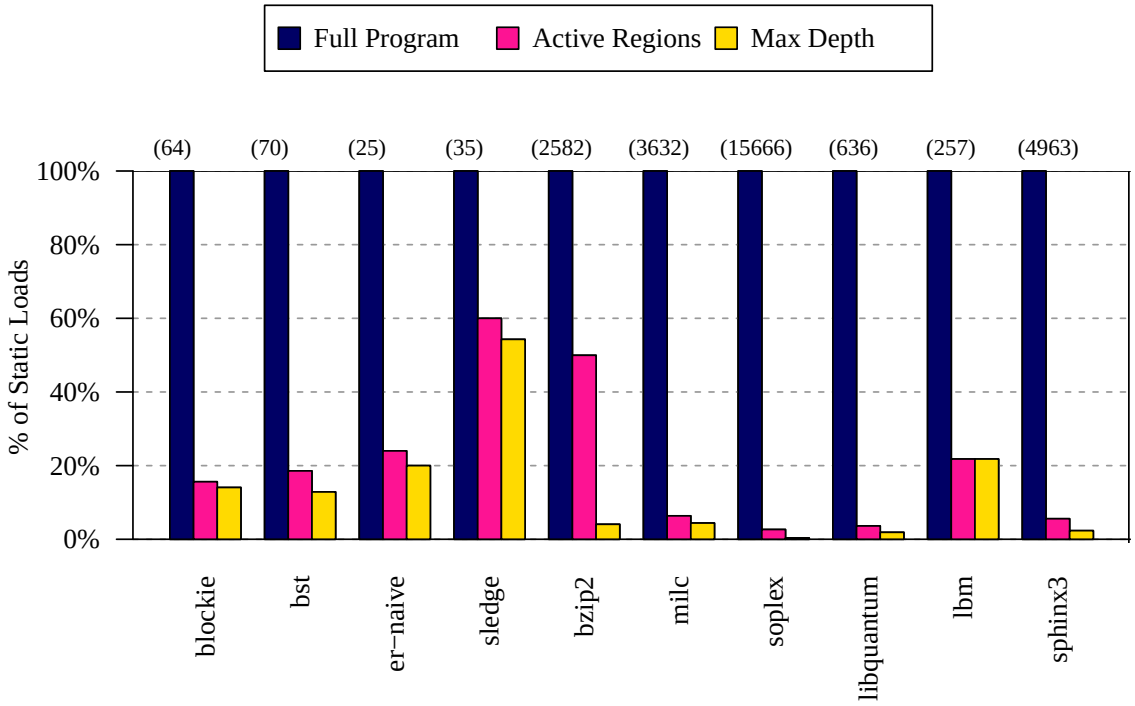


Figure 4.4: Heuristics significantly reduce the search space for PC3D. Static load counts of the full programs are presented in parentheses above the bars

ined in the search by an average factor of 44× while covering more than 80% of the dynamic loads. It is notable that the reduction in number of loads is largest for programs with high load counts, such as `soplex` (15666 loads reduced to 57) and `sphinx3` (4963 loads reduced to 116), showing that the heuristics help keep the variant search manageable even for programs that have large code bases.

4.2.2 Utilization Improvements from PC3D

In this section we evaluate PC3D, showing its impact on server utilization and application QoS when running batch applications with latency-sensitive webservice applications, including `web-search`, `media-streaming` and `graph-analytics` from

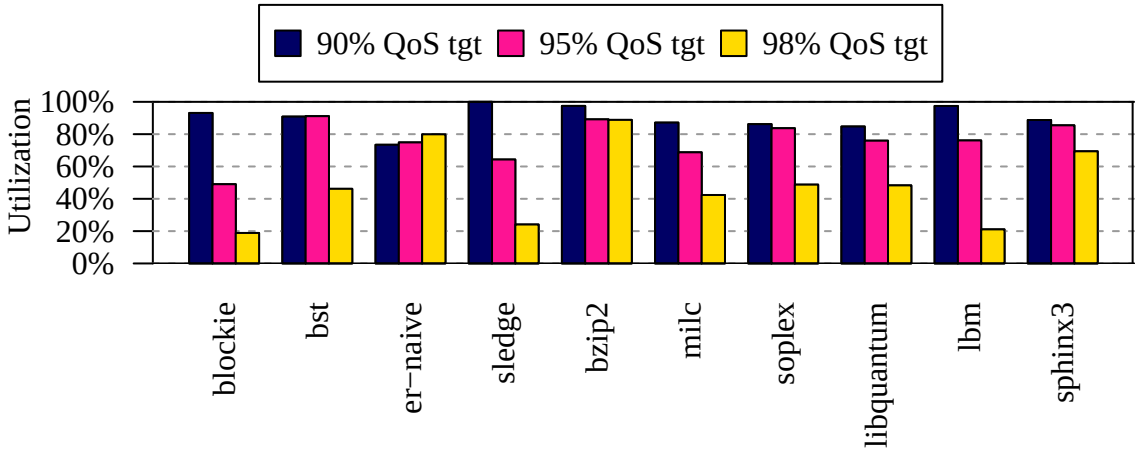


Figure 4.5: Utilization improvement of applications running with `web-search`

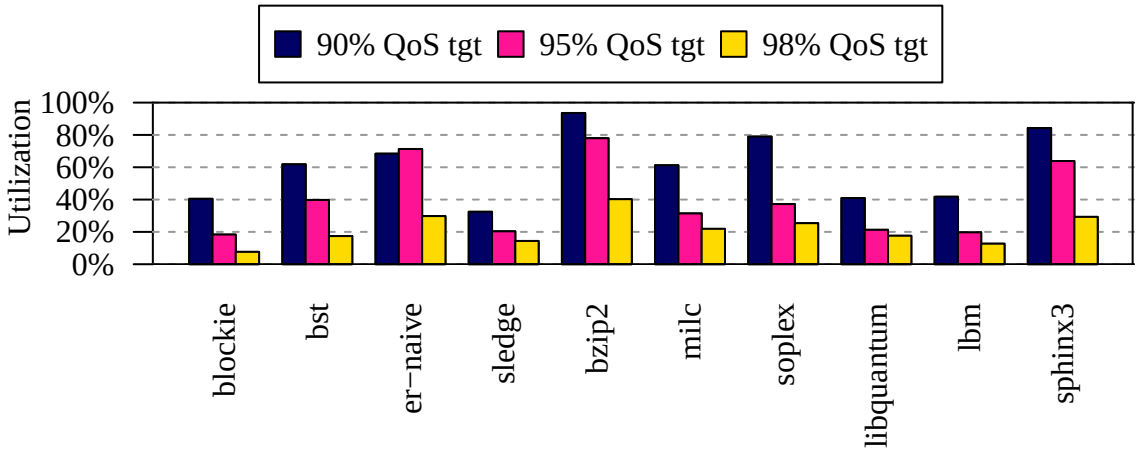


Figure 4.6: Utilization improvement of applications running with `media-streaming`

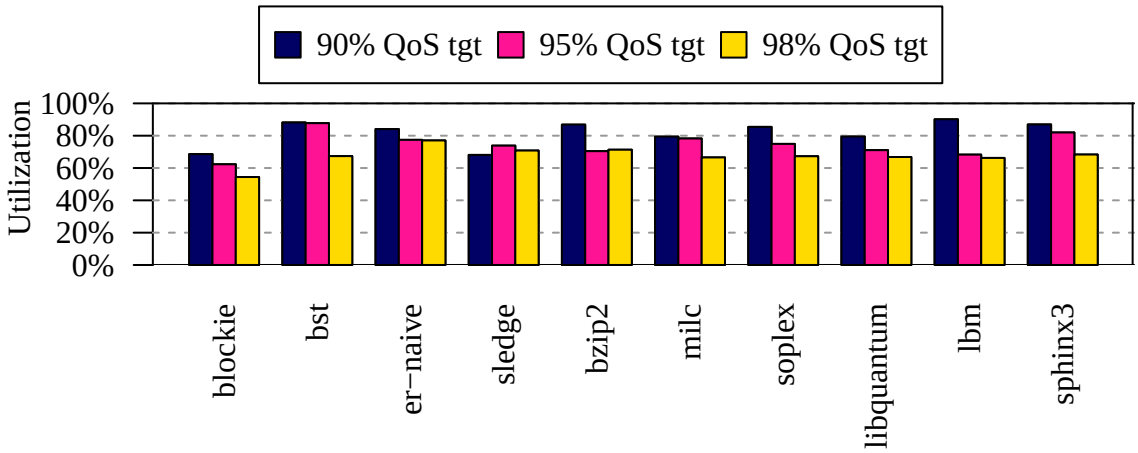


Figure 4.7: Utilization improvement of applications running with `graph-analytics`

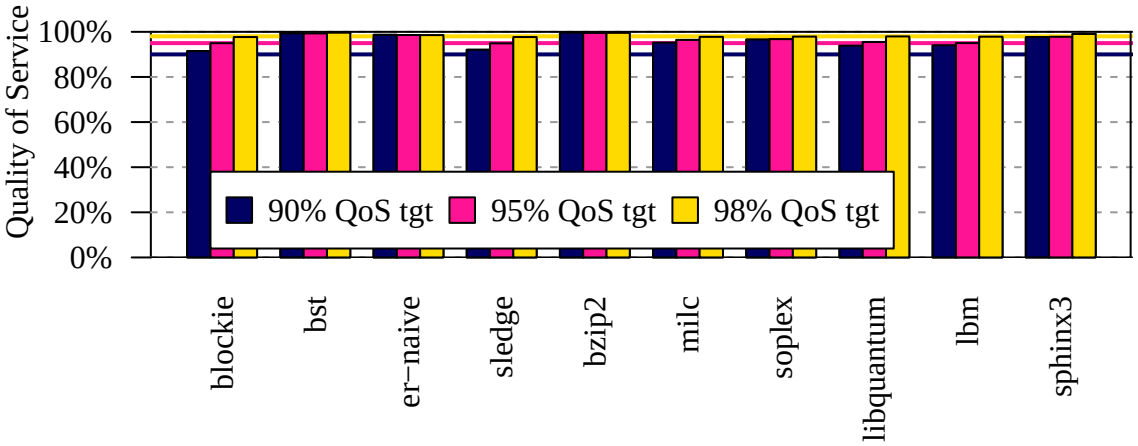


Figure 4.8: QoS of `web-search` running with various applications

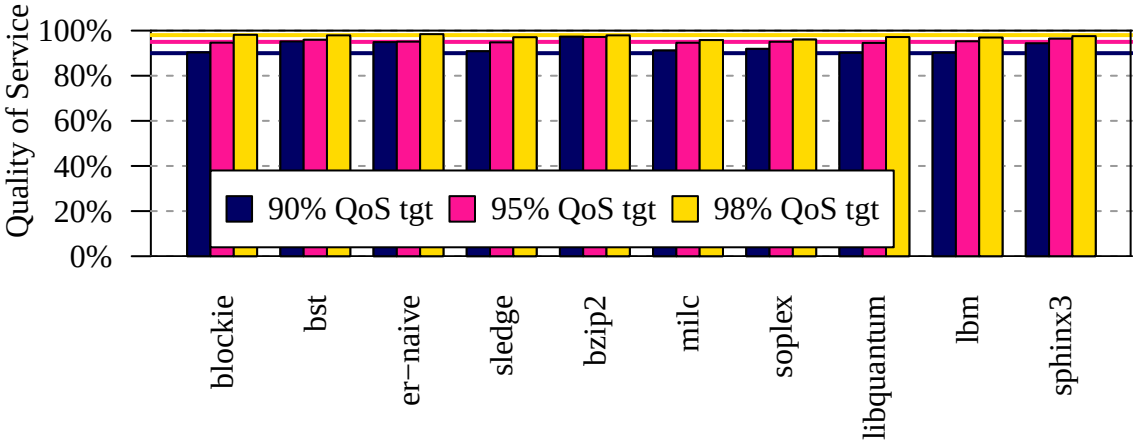


Figure 4.9: QoS of `media-streaming` running with various applications

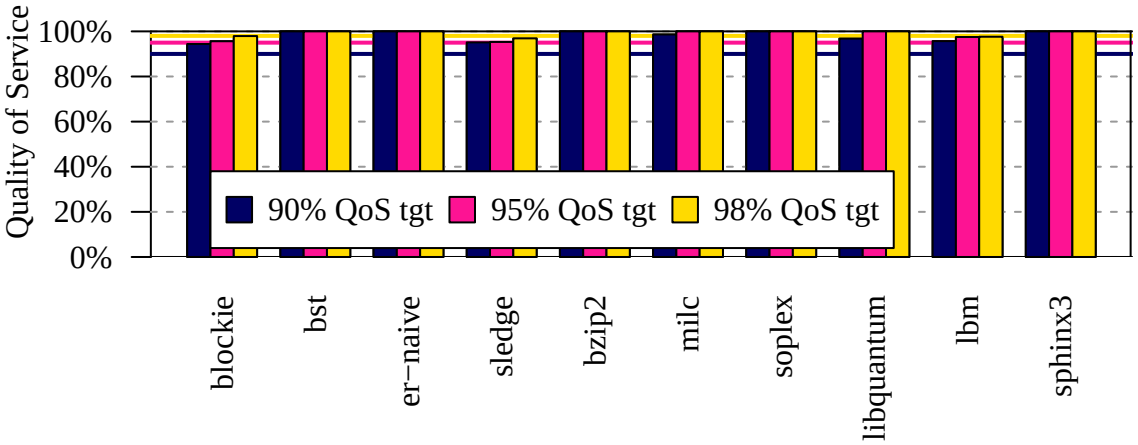


Figure 4.10: QoS of `graph-analytics` running with various applications

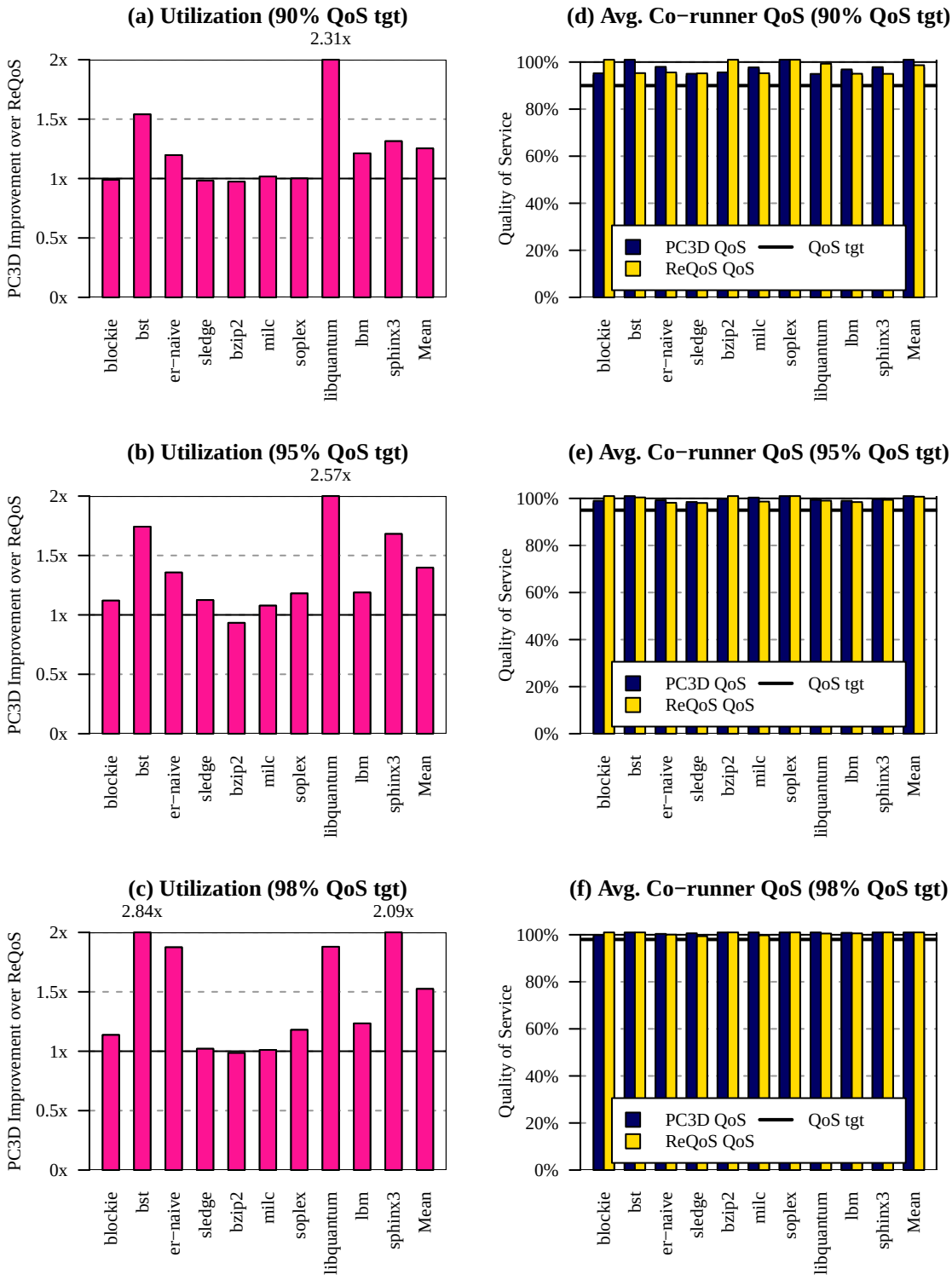


Figure 4.11: Utilization (top) and QoS (bottom) of PC3D vs. ReQoS, presented as the average across all CloudSuite, SPEC and SmashBench applications

Table 4.1: Applications used in datacenter experiments

	Host (batch) applications	**External (latency-sensitive) apps**
CloudSuite	-	`web-search`, `media-streaming`, `graph-analytics`
SPEC CPU2006	`bzip2`, `milc`, `soplex`, `libquantum`, `lbm`, `sphinx3`	`libquantum`, `mcf`, `milc`, `omnetpp`, `xalancbmk`
SmashBench	`bst`, `blockie`, `er-naive`, `sledge`	`bst`, `er-naive`
PARSEC	-	`streamcluster`

CloudSuite. The set of latency-sensitive and batch applications we evaluate are presented in Table 4.1. For these experiments, QoS is presented as the instructions per second (IPS) an application achieves normalized to its IPS running alone on the server. Using IPS in this fashion as a proxy for QoS is consistent with practices in industry [186], where simple performance monitors are collected regularly and ubiquitously via mechanisms such as the Google Wide Profiler (GWP) [137] and used for making QoS estimates. Likewise, we present application utilization as the branches per second (BPS) measured by PC3D as a fraction of the BPS the non-protean version of the application achieves while running alone on the server. BPS is a useful metric in this case because PC3D introduces control-invariant code transformations that may include executing extra non-temporal access hint instructions in key code regions.

In these experiments, the latency-sensitive application runs on a single core of the server, while the contentious batch application runs on another single core. The contentious batch application is compiled with the protean code compiler, and may be modified dynamically to be less cache contentious if PC3D detects that the latency-sensitive application fails to meet its QoS target. The PC3D runtime consumes only a small fraction of the cycles on the server (Figure 4.13), monitoring all running applications to detect co-phase changes, checking that the latency-sensitive application

meets its QoS target, and potentially introducing transformations that improve the cache contentiousness of the batch application.

Live Webservices. Figures 4.5, 4.6 and 4.7 show the utilization gains achieved by PC3D over a policy of disallowing co-locations on a series of benchmarks as they run with `web-search`, `graph-analytics`, and `media-streaming`. Each cluster of bars shows the results of a particular batch application running against one of the webservices. The three bars in each cluster show the utilization gained with QoS targets of 90%, 95% and 98%. As applications co-run with `web-search`, they show an average utilization gain of 49% when a 98% QoS target is used. When less stringent QoS targets are in place, PC3D must mitigate contention to a lesser degree, which allows them to achieve higher utilization rates. With a 95% QoS target, the average utilization is 67% and with a 90% QoS target the utilization gain is 81%. Similarly, utilization improvements for `graph-analytics` are 67%, 75%, 82% for the three QoS targets. `media-streaming` is more sensitive to contention than `web-search` and `graph-analytics`, where we observe utilization improvements of 22%, 40% and 60%. Overall, these results show that PC3D consistently delivers substantial utilization gains, even in the presence of heavily contentious applications such as `libquantum` and `lbm`.

Figures 4.8, 4.9 and 4.10 present the QoS of the co-running webservice applications during the same set of experiments. These results show that PC3D reliably meets its QoS targets.

Comparison to State-of-the-Art. Figure 4.11 presents the utilization achieved by PC3D compared to ReQoS [170], an approach for reducing application contentiousness that employs a hybrid static/dynamic approach to introduce naps into the running application. The results shown are the average utilization improvement of PC3D over ReQoS for a number of batch applications averaged over the entire spectrum of CloudSuite, SPEC and SmashBench co-runners. PC3D employs a napping mecha-

nism similar to the mechanism used by ReQoS to throttle applications when reducing cache contention by dynamically inserting non-temporal hints is insufficient to allow the latency-sensitive co-runner to meet its QoS target, so in several cases ReQoS and PC3D show similar utilization levels. In a number of cases, however, PC3D gains far more utilization than ReQoS. For example, at a 98% QoS target, PC3D delivers over 2x the utilization of ReQoS on `sphinx3` by finding an improved code variant, leading to far lower cache contentiousness at relatively small performance overhead to `sphinx3`. On average, PC3D improves utilization by a factor of 1.25, 1.45 and 1.52x at QoS targets of 90%, 95% and 98%, respectively. Figure 4.11 also includes the co-runner QoS, again presented as the average over the entire spectrum of co-runners. Both PC3D and ReQoS consistently meet the co-runner QoS targets.

4.2.3 Webservice with Fluctuating Load

To further evaluates how PC3D adapts to the dynamism in the application and its execution environment, Figure 4.12 presents the dynamic behavior of PC3D and ReQoS as `libquantum` runs with `web-search`. The load on `web-search` shifts over the course of the run, with the load pattern shown in 4.12(a). 4.12(b) shows a trace of the performance (branches per second) of `libquantum` over the same time frame. 4.12(c) shows the QoS of `web-search`, and 4.12(d) shows the cycles spent running the PC3D runtime.

PC3D Dynamic Behavior. `libquantum` initially (t=0) begins to execute alongside `web-search`. PC3D continuously monitors `web-search` as an external application, and detects that `libquantum` jeopardizes `web-search` QoS, so PC3D begins to search for alternate code variants for `libquantum` that allow `web-search` to meet its QoS while allowing `libquantum` to make better progress. The performance of `libquantum` during the variant search is shown in greater resolution in 4.12(e). By t=20, PC3D has arrived at an improved variant of `libquantum`, and PC3D allows it

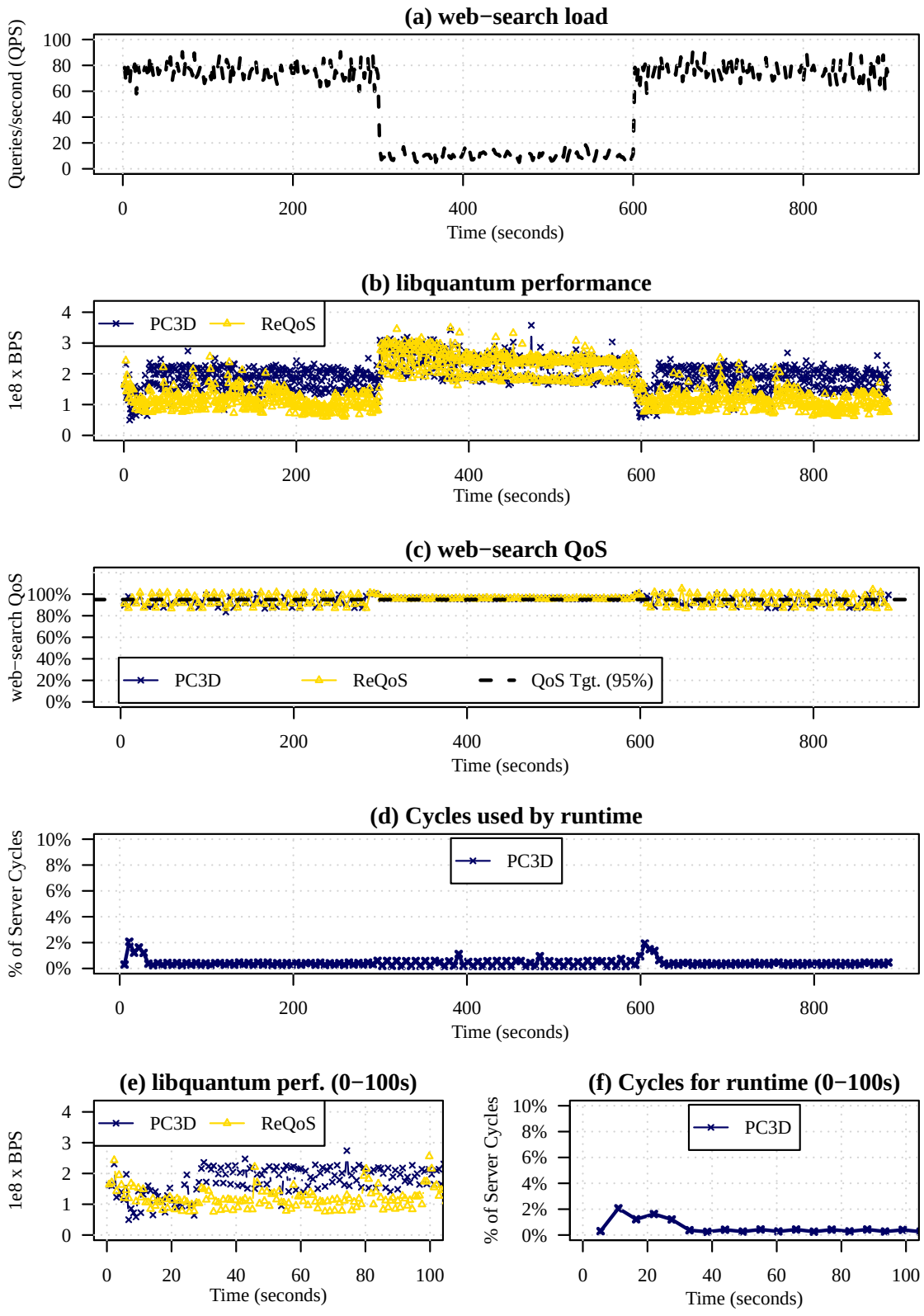


Figure 4.12: Dynamic behavior of `libquantum` running with `web-search` using the PC3D runtime

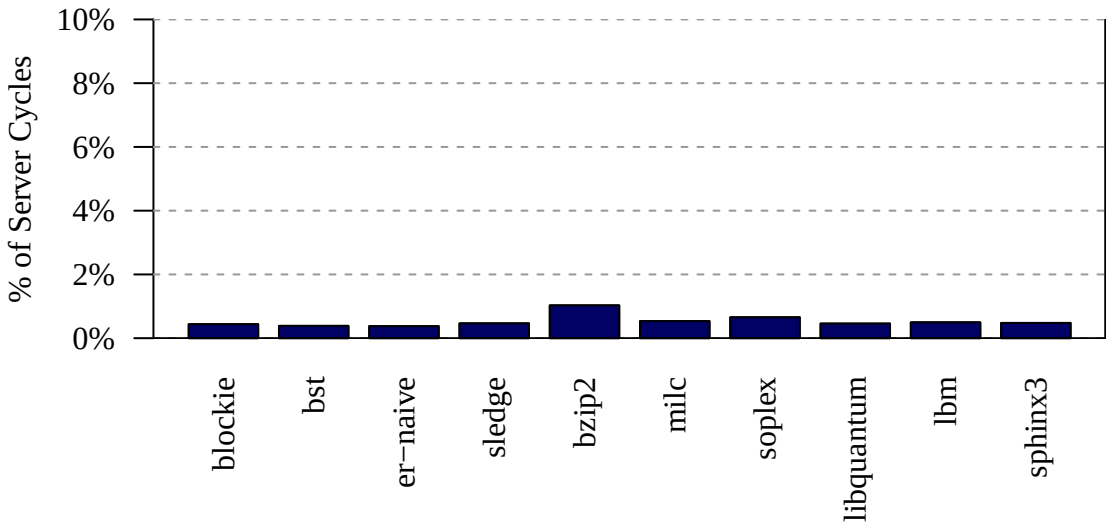


Figure 4.13: Average fraction of server cycles consumed by the PC3D runtime

to run with this variant until a co-phase change is detected at t=300.

At t=300, the demand placed on `web-search` shifts, at which point PC3D detects a change in the behavior of `web-search`, causing it to revert `libquantum` back to its original (no non-temporal hints) variant. Until t=600, the original variant of `libquantum` runs at full speed because `web-search` is not sensitive to contention at low load.

At t=600, the load to `web-search` picks up and PC3D again searches for an improved variant that reduces cache contention. At t=620, the variant search ends and the improved variant of `libquantum` runs until the end of the experiment (t=900).

Cycles Consumed by PC3D. A unique feature of protean code is that the work of dynamic compilation of a host program may be offloaded to use otherwise spare cycles on the host server, putting those cycles to work for the benefit of the running applications. Figure 4.12(d) shows the fraction of server cycles used by the PC3D runtime. Activity is minimal, kept to well below 1% of the server's cycles for the majority of the run. Two brief mini-spikes of up to 2% appear at t=0 (a higher-

Table 4.2: Workload mixes for scale-out analysis

LS	`web-search`, `graph-analytics`, `media-streaming`
WL1	`libquantum`, `bzip2`, `sphinx3`, `milc`
WL2	`soplex`, `bst`, `milc`, `lbm`
WL3	`sledge`, `soplex`, `sphinx3`, `libquantum`

resolution view of this spike is presented in 4.12(f)) and t=600 as PC3D generates code to search for variants that improve the performance of `libquantum`.

4.2.3.1 Cycles Used by the Runtime

We highlight that this low level of overhead is not specific to this pair of applications. While the demand on the runtime to generate new variants is inevitably a function of the optimization objective, in PC3D the CPU utilization levels of the dynamic compiler and the entire runtime are quite low. Figure 4.13 presents the percentage of the server's cycles used by the PC3D runtime to manage a variety of batch applications, which is less than 1% in all cases.

ReQoS Dynamic Behavior. Figures 4.12(b) and (c) also show the impact of ReQoS on the same run of `libquantum` and `web-search`. ReQoS adjusts the nap intensity, reacting to load changes at t=300 and t=600. During periods of high load it allows `web-search` to meet its QoS target strictly by applying naps to `libquantum`, causing `libquantum` to make significantly slower progress than it makes when running with PC3D.

4.2.4 Impact of PC3D at Scale

This section discusses the impact of deploying PC3D in a large-scale datacenter cluster that houses a mix of webservice and batch applications, showing that, by substantially improving server-level utilization, PC3D can have a large impact on the

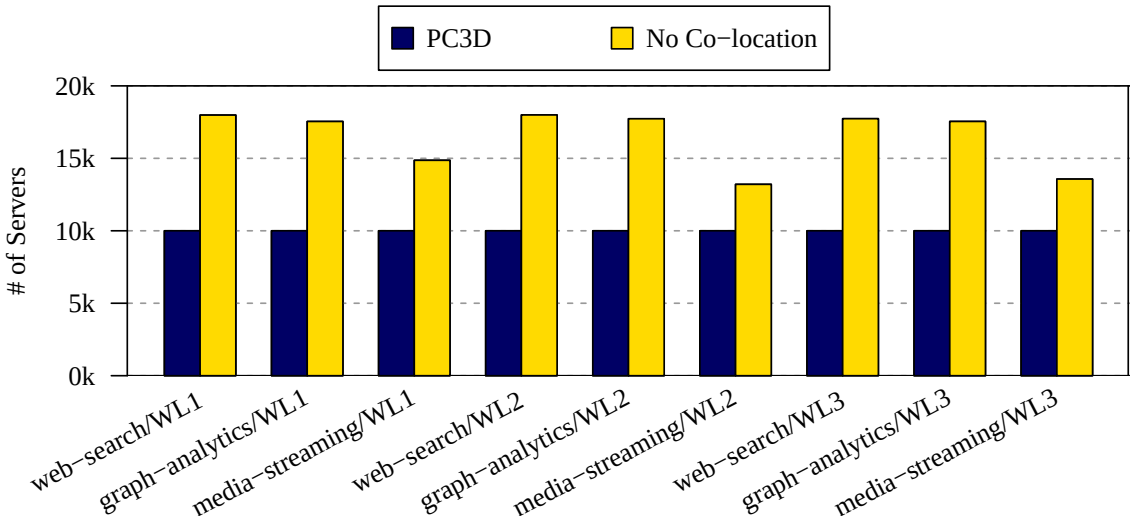


Figure 4.14: Server count required to run workload mixes for PC3D vs. no co-location

number of servers needed to house a particular workload and on the energy efficiency of the datacenter.

Server Requirements. Figure 4.14 presents an analysis of the number of servers required to house a variety of webservice and batch application mixes. This analysis assumes a datacenter with 10k machines and the workload mixes described in Table 4.2, with 10k instances of a latency-sensitive webservice (LS) with 95% QoS target along with 10k batch application instances comprised equally of one of the mixes shown in the table (WL). Running with PC3D, the 10k machines are able to achieve a particular level of throughput on each application. Using a policy of disallowing co-locations, extra servers are needed to run the batch applications to achieve an equivalent level of throughput as the PC3D-enabled datacenter. Figure 4.14 shows that between 3.5k and 8k extra servers are needed on top of the original 10k servers to achieve a level of batch throughput that matches a PC3D-enabled datacenter.

Energy Efficiency. Using a large number of extra servers also has a significant impact on the overall energy efficiency of the datacenter. Using a similar setup to the

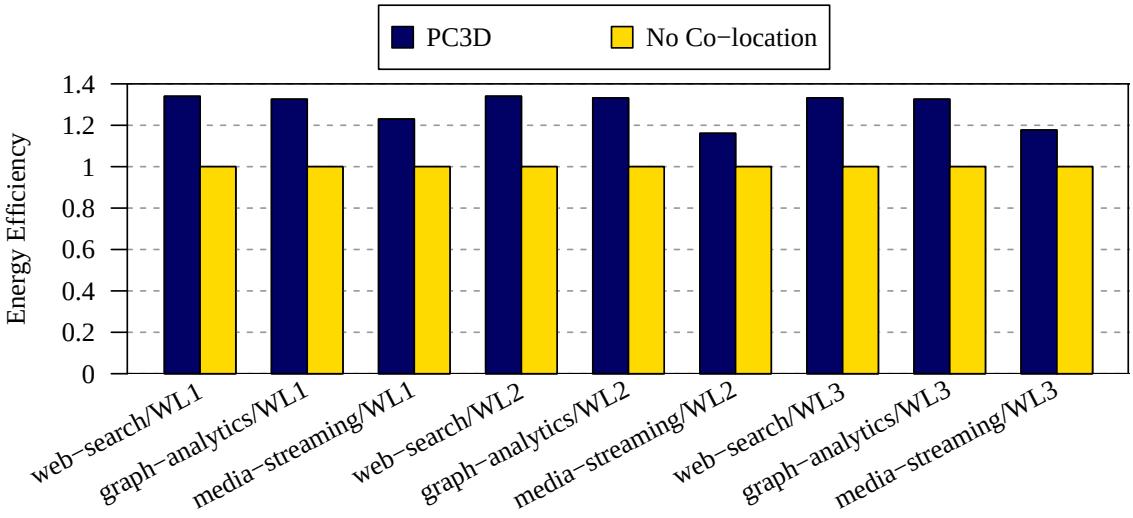


Figure 4.15: Normalized energy efficiency of workload mixes for PC3D vs. no co-location

previous experiment, we employ a linear CPU utilization model to derive the power consumption of the servers within the datacenters, from which we compute the overall performance per Watt of each datacenter and derive energy efficiency comparisons of the datacenters. Figure 4.15 presents a comparison of the energy efficiency of the PC3D-enabled datacenter normalized to the No Co-location datacenter running the same workload at the same throughput, from which we observe that PC3D improves energy efficiency at the datacenter level by 18-34% across a spectrum of webservice and batch workloads.

4.3 Summary

This chapter presents the design and evaluation of Protean Code for Cache Contention in Datacenters (PC3D), a runtime approach to mitigating cache contention for live datacenter applications. PC3D uses the online code transformation capability of protean code to dynamically inserting and removing software non-temporal cache hints, allowing batch applications to achieve high throughput while meeting latency-

sensitive application QoS. On a spectrum of webservice and benchmark applications, PC3D achieves utilization improvements of up to 2.8x (average of 1.5x) higher than a recently published state-of-the-art contention mitigation runtime at a QoS target of 98%.

CHAPTER V

Input Responsive Approximate Computing

This chapter introduces Input Responsive Approximation (IRA), an approach that uses a *canary input* — a small input carefully constructed to capture the intrinsic properties of the original input — to automatically control how approximation is applied on an input-by-input basis for approximate programs. The key insight of this approach is the observation that prior work on choosing *how* to approximate arrives at conservative decisions by discounting substantial differences between inputs when applying software approximation techniques. The main challenges to overcoming this limitation lie in making the choice of how to approximate both effectively (e.g., the fastest approximation that meets a particular accuracy target) and rapidly for every input. With IRA, each time the approximate program is run, a canary input is constructed and used dynamically to quickly test a spectrum of approximation alternatives. Based on these tests, the approximation that best fits the desired accuracy constraints is selected and applied to the full input to produce an approximate result.

We use IRA to select and parameterize mixes of four approximation techniques from the literature for a range of 13 image processing, machine learning, and data mining applications. Our results demonstrate that IRA significantly outperforms prior software-only techniques, delivering an average of 10.2× speedup over exact execution while minimizing accuracy losses.

5.1 The Case for Input Driven Dynamism

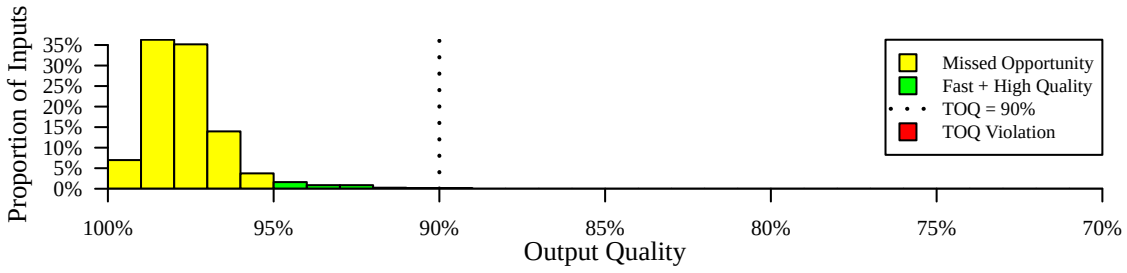
The ability of approximate computation to produce acceptably high-quality results is vital to ensure that users have a positive experience and thus is one of the keys to making approximation broadly deployable in real systems. Current techniques for preserving result accuracy focus on the worst case, resulting in overly conservative approximation for other cases. This section discusses the opportunity available in the presence of a technique that dynamically monitors and controls the quality of results for individual inputs.

5.1.1 Input Matters for Output Quality

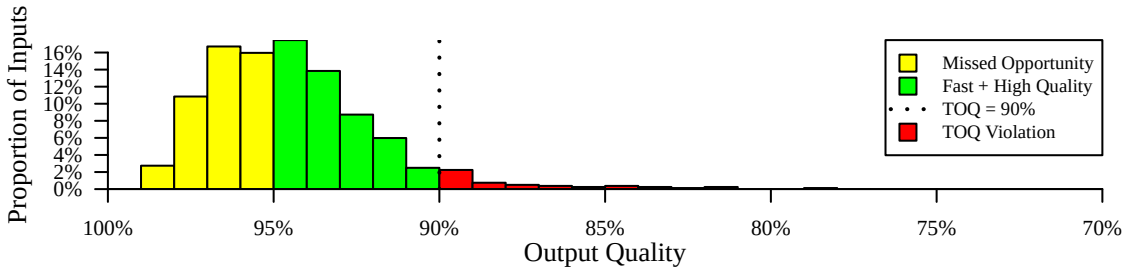
Input is an important part of the accuracy of an approximate computation. To illustrate this, we detail the output quality produced by three different tiling approximations [146] of an image processing application called *gamma correction* [133] applied to 800 input images. Tiling is based on the assumption that, in many application domains such as image and video processing, elements nearby one another (e.g., pixels in an image) are likely to have similar values. Instead of computing each element of the output, a tiling approximation computes a single output element and projects that output onto the surrounding elements to form a tile. Tiling can be tuned to trade off lower accuracy for better performance by increasing the size of the tile.

Figure 5.1 presents histograms of the output quality for 800 different images across three tile sizes. For the purposes of illustration, we assume that the *target output quality*[1] (TOQ) of the approximation is 90%. As shown in the figure, for all three tile sizes, different inputs can result in very different output qualities. For example, across these inputs 8x8 tiling (Figure 5.1(b)) results in output qualities ranging from

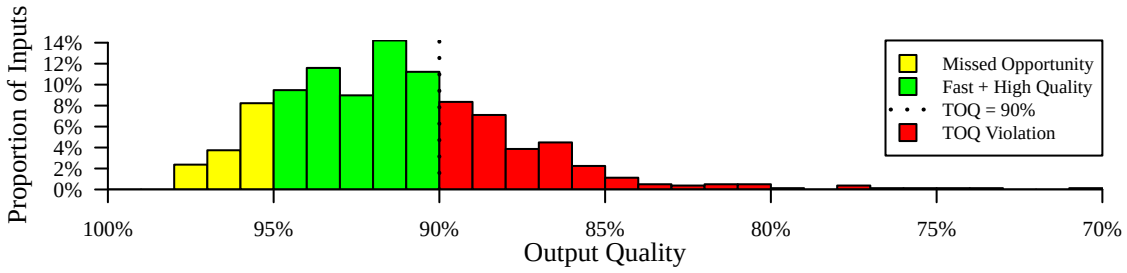
[1]Target output quality (TOQ) is the minimum acceptable result accuracy [147], supplied by the user of the application.



(a) 4x2 tiling approximation (5.9x speedup)



(b) 8x8 tiling (22x speedup)



(c) 16x16 tiling (83x speedup)

Figure 5.1: Histograms of the accuracy of three tiling approximations applied to the same 800 images; some mix of missed opportunities and unacceptably low accuracy are present in each approximation

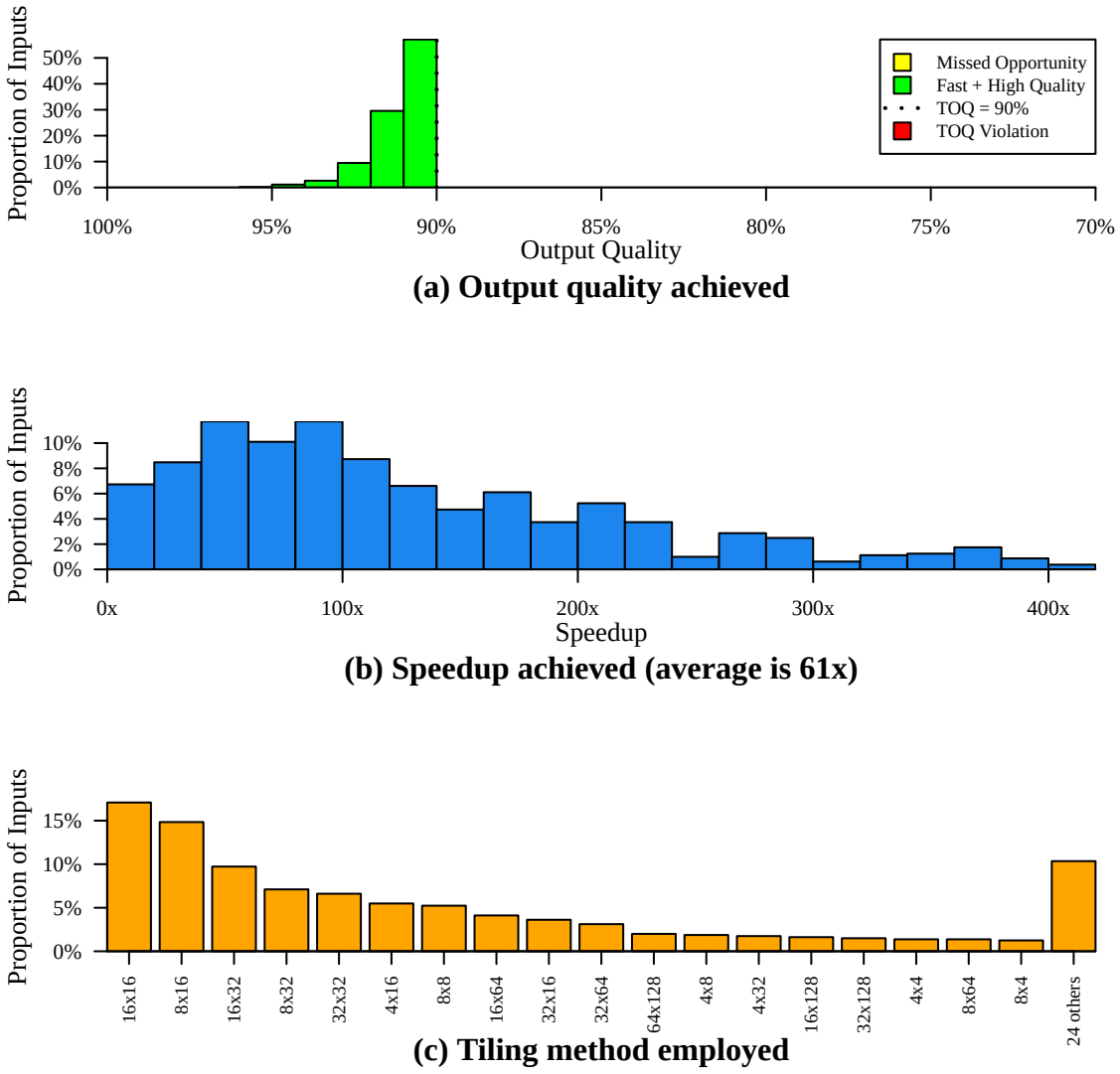


Figure 5.2: A dynamic oracle approximation system using the most effective tiling approximation method (fastest without violating TOQ) achieves an average speedup of 61× and uses 42 different approximation options

79%-99% because the assumption made by the approximation technique (that nearby pixels are similar to one another) holds true to a different extent depending on the composition of the input. Furthermore, we have observed that a wide range in output quality across inputs is not unique to tiling approximation and gamma correction, persisting across many computational problems and approximation techniques.

5.1.2 Limitations of Existing Approaches

The ubiquitous approach used in approximate computing is to choose a single approximation option for some problem and apply that approximation to multiple inputs. Existing approaches to approximation therefore suffer from a form of the *problem of aggregation*, in which aggregate behavior (average or worst) is not necessarily representative of individual behavior. In the presence of multiple differing inputs, an approximation system that uses a single approximation across inputs either leaves performance opportunities on the table, violates output quality restrictions, or both.

To illustrate this, we refer again to Figure 5.1, where (a), (b) and (c), are histograms of the result accuracy for three increasingly aggressive approximate gamma corrections applied to 800 input images. We assume a TOQ of 90%, and characterize the outputs as falling into 3 classes: TOQ violating approximations ($<$ 90% output quality), fast + high quality approximations (90-95% output quality) and missed opportunities (95-100% output quality). The 4x2 tiling approximation, shown in 5.1(a), produces minimal TOQ violations, but the speedup is limited to 5.9×. The bulk of these output qualities can be classified as missed opportunities. A more moderate approach, 8x8 tiling, is shown in 5.1(b). In this case, 5% of the results violate the TOQ with a speedup of 22×. Finally, the results of an aggressive approximation are shown in 5.1(c). This approach uses 16x16 tiling and yields 83× speedup with 30% of the outputs violating the TOQ.

5.1.3 The Opportunity for Dynamism

Ideally, approximation would have the best of both worlds – no missed opportunities and no TOQ violations. This could be achieved by dynamically choosing the most *effective* approximation method for each input – the fastest approximation method that does not violate the TOQ.

To illustrate this opportunity, Figure 5.2(a) presents a histogram of output quality over the 800 inputs using the most effective of the available approximation methods, chosen by a dynamic oracle. Unlike the previous example, the most effective approximation is always fast and high quality, never leaving performance on the table and never violating the target output quality. Moreover, Figure 5.2(b) shows a histogram of the speedups achieved on the set of 800 inputs. The speedups vary significantly, ranging from 3.5× to 410× (average 61×) due to the fact that a wide range of approximations are chosen. As shown in Figure 5.2(c), across 800 inputs, 42 unique approximation methods are chosen, with no single approximation being used on more than 17% of the inputs. That is, a wide range of approximation methods are used to obtain the maximally effective approximation across the set of inputs and no single approximation is dominant. *The key to taking advantage of this opportunity is to customize the approximation for each input on an individual basis, and to develop that customized approximation quickly.*

5.2 Overview of IRA

Given a computational problem, a menu of approximation options, and an input to the problem, the goal of Input Responsive Approximation (IRA) is to rapidly choose an effective approximation for that input. Our approach to achieving this goal is shown in Figure 5.3 and does the following:

1. Canary Input – first, IRA dynamically produces a *canary input*, a smaller

representation of the input (Section 5.3.1). The creation of the canary is guided by hypothesis testing, a statistical framework used to ensure that the resulting canary is large enough to be representative of the full input, sharing key properties with the full input, while being no larger than necessary.

2. Customize the Approximation – next, exact and approximate solutions are computed using the canary to select the most effective from among the available approximation options, including selecting the code regions to approximate and how to approximate within those regions (Section 5.3.2). Because the canary input is much smaller than the full input, IRA is able to rapidly forecast how numerous approximations fare on a particular input by running the canary input with each of those approximations. Unlike prior work, IRA predicts the accuracy and performance of approximations on each input on demand and *ex ante*, allowing it to find and use a customized, effective approximation for every input.

3. Compute Approximate Solution – last, the customized approximation deemed effective for the canary is applied to the full input to produce an approximate solution that is of acceptable accuracy (Section 5.3.3). As we show later, this approach is extremely effective, leading to large performance improvements with minimal accuracy losses and outperforming oracle versions of prior techniques.

5.3 IRA Design and Implementation

This section provides a detailed description of how IRA develops a customized approximation for each problem input.

5.3.1 Reasoning About Canary Inputs

Creating a canary input that exhibits the properties of the full input has three main challenges. First, in determining the similarity of the canary and the full input, we

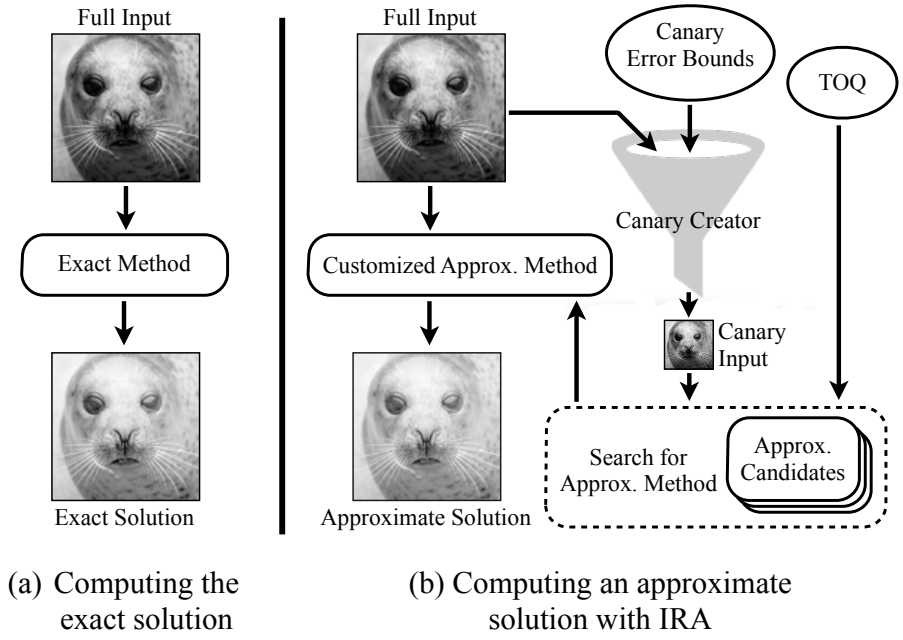


Figure 5.3: Exact computation and approximation with IRA

must use a definition of similarity that reflects meaningful properties of the inputs. Second, we must be able to choose the canary in a way that is both computationally inexpensive and ensures that the definition of similarity is satisfied. Third, we want to choose a canary that is much smaller than the full input, as this will be a large determinant of the time spent employing the canary to test various approximations.

A plausible approach for creating canaries could be to sample down all inputs at the same rate. Unfortunately, this approach produces canaries that are either (1) larger than necessary for "well-behaved" inputs, introducing extra overhead in the approximation search process or (2) too small to adequately represent the full input, resulting in a search that provides a misleading model of approximation accuracy characteristics. Empirically, we have found that the dynamic canary creation mechanism in IRA significantly outperforms a fixed scale-down strategy for creating canaries. This issue is explored in greater detail in Section 5.4.2.

This work explores four different metrics of canary similarity, designed to span a range of definitions of what it means for inputs to be similar. These metrics range from

a very simple metric of ensuring that the values in the canary are close, on average, with the values in the full input to complex metrics that ensure the similarity of local properties within small regions of the input. We discuss these metrics in detail in Section 5.3.1.3.

To address the second challenge, we ensure low overhead in the canary creation process by employing statistical sampling in the analysis of each potential canary input, thus allowing us to compute metrics on just a small subset of the canary input when analyzing its similarity to the full input. To ensure that the definition of similarity is satisfied in a chosen canary, we use a carefully designed algorithm based on robust, automated hypothesis tests that minimize the likelihood of making an incorrect decision about each canary. In particular, we take special care to design our approach to avoid both *false negatives* – incorrectly finding dissimilarity – and *false positives* – incorrectly finding similarity. These are also known as Type I and Type II errors, respectively. The avoidance of false positives ensures that the canary we select is highly likely to be similar to the full input.

Likewise, avoiding false negatives is key to ensuring that the chosen canary is no larger than needed. If we mistakenly rejected a small canary that was actually similar to the full input in favor of a larger canary, the canary-driven search can have unnecessarily high overhead.

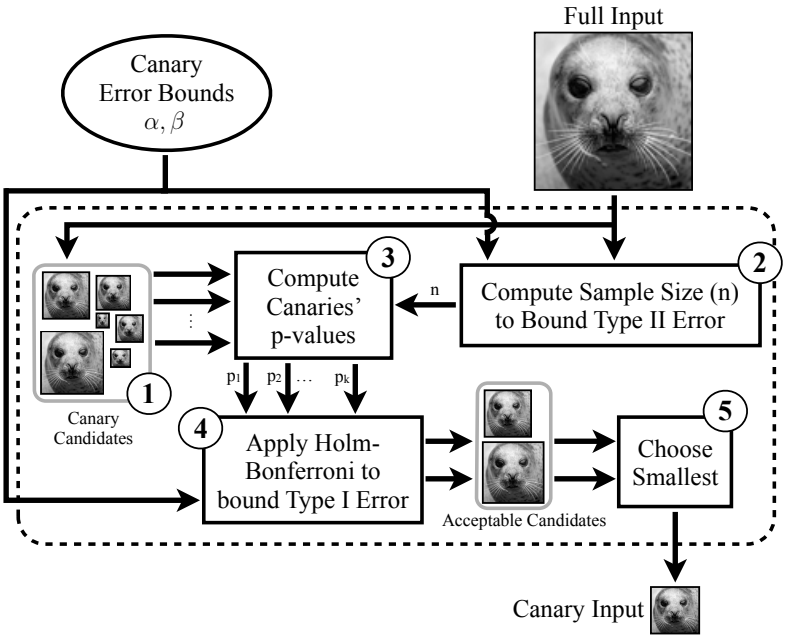


Figure 5.4: Canary input creation

	Mean	**Variance**	**Local Homogeneity**	**Autocorrelation**
Description	Mean μ of input elements	Variance σ^2 of input elements	Proportion Λ of elements represented by λ in canary $\notin [\lambda - \sigma z_{1-\alpha/2}, \lambda + \sigma z_{1-\alpha/2}]$	Correlation ρ among pairs of input elements (y_j, y_{j+1})
Null Hypothesis (H_0) **Alt. Hypothesis** (H_A)	$H_0 : \overline{\mu_i} = \mu_0$ $H_A : \overline{\mu_i} \neq \mu_0$	$H_0 : \overline{\sigma_i}^2 = \sigma_0^2$ $H_A : \overline{\sigma_i}^2 \neq \sigma_0^2$	$H_0 : \overline{\Lambda_i} \leq 0.1$ $H_A : \overline{\Lambda_i} > 0.1$	$H_0 : \overline{\rho_i} = \rho_0$ $H_A : \overline{\rho_i} \neq \rho_0$
Test Statistic (t_i)	$t_i = \frac{\mu_0 - \overline{\mu_i}}{\overline{\sigma_i}\sqrt{n}}$	$t_i = \frac{\overline{\sigma_i}^2}{\sigma_0^2}$	$t_i = \frac{\sqrt{n}\ \lvert\overline{\Lambda_i} - 0.1\rvert}{\sqrt{0.1(1-0.1)}}$	$t_i = \frac{ln\left(\frac{(1+\rho_0)(1-\overline{\rho_i})}{(1-\rho_0)(1+\overline{\rho_i})}\right)}{2\sqrt{n-3}}$
p-value (p_i)	$p_i = 2P(Z > t_i)$	$p_i = 2P(F_{n-1,n-1} > t_i)$	$p_i = 2P(Z > t_i)$	$p_i = 2P(Z > t_i)$
Sample Size (n)	$n = 2(z_{1-\alpha/2k} + z_{1-\beta/k})^2$	Formula yields no simple form; see Cohen [46] for details.	$g(x) = \sqrt{x(1-x)}$ $n = 0.1^{-2}(g(0.1)z_{1-\alpha/2k} + g(\overline{\Lambda_i})z_{1-\beta/k})^2$	$n = \frac{4(z_{1-\alpha/2k} + z_{1-\beta/k})^2}{ln((1+\rho_0)/(1-\rho_0))^2}$
Acceptability Test	*Holm-Bonferroni method:* sort p-values $p_1, p_2, \ldots p_k$ to obtain sorted p-values $p_{(1)}, p_{(2)}, \ldots p_{(k)}$. Find the minimum index m such that $p_{(m)} > \frac{\alpha}{k+1-m}$, then reject all canaries $C_{(i)}$ where $i \geq m$.			
Definitions	α: the desired bound on the probability of committing any Type I errors (false negative), β: the desired bound on Type II errors (false positive) k: the number of canary candidates, C_i: the ith canary candidate, $\overline{x_i}$: the sample statistic x for canary C_i, x_0: the sample statistic x for the full input Z: the standard normal distribution, z_y: the quantile function at y of Z, $F_{b,c}$: the F-distribution with degrees of freedom b and c			

Table 5.1: Similarity metrics used to assess canary similarity to full input, along with the relevant statistical formulas

5.3.1.1 Canary Construction

The algorithm for creating a canary is depicted in Figure 5.4. The inputs to the algorithm are the desired bound on the likelihood of getting a Type II error α, the desired bound on getting a Type I error β, and the full input to the problem. The output of the algorithm is a small canary input deemed similar to the full input.

① **Generating Canary Candidates.** First, a set of candidate canaries $C_1, C_2, \ldots, C_k$ are generated. One of the key determinants of the quality of the canary is its size; a larger canary is likely to be a better reflection of the full input than a smaller canary. However, as the purpose of the canary is to use it in a dynamic search, a larger canary will also tend to result in a more expensive search. Our approach to generating candidates is to expose this inherent tradeoff, using candidates of many different sizes then choosing the smallest canary from among the candidates that is similar enough to the full input according to one of the metrics described in Section 5.3.1.3.

We generate $C_1, C_2, \ldots, C_k$ such that they are regular, strided subsets of the full input. If N is the size of the full input, we currently select canary candidates that are size $^N/_{16}$, $^N/_{32}$, $^N/_{64}$, $^N/_{128}$ and $^N/_{256}$. The reason we explicitly avoid selecting canaries larger than $^N/_{16}$ is that canaries that are larger may take an unacceptably long time in the dynamic search, likely counteracting the performance gains IRA aims to achieve by approximating the problem. For one-dimensional inputs such as arrays of scalars or arrays of structs, an input of size $^1/_t$ is produced by taking every tth element from the input. For two-dimensional inputs such as matrices and images, an input of size $^1/_t$ is produced by taking every $^1/_{\sqrt{t}}$th element along both dimensions. This approach can easily be extended to higher-dimension inputs, however this was not necessary for any of the test applications in this work.

5.3.1.2 Canary Selection

The remainder of the steps in this algorithm are focused on choosing the smallest canary from among these candidates that is similar to the full input.

② **Sample Size.** We next calculate the number of samples to take from each canary when evaluating their similarity. This calculation is designed to bound the likelihood of getting a Type II error when comparing those properties to the full input, discussed in further detail in Section 5.3.1.4. This sample size is denoted n.

③ **Canary Statistics.** We calculate the statistics needed to perform hypothesis tests on the canaries, taking a random sample of size n from each canary C_i, then use this to compute a test statistic for the canary t_i and a p-value p_i associated with that test statistic. We discuss tests statistics and p-values in more detail in Section 5.3.1.4, however t_i is simply a statistical measurement of the similarity between the canary and full input, while p_i is the statistical significance of that measurement.

④ **Canary Acceptability.** Using the resulting p-values $p_1, p_2, \ldots, p_k$, we employ the Holm-Bonferroni method, a technique designed specifically to bound the likelihood of getting a Type I error when performing multiple hypothesis tests [86], to partition the candidate canaries into two groups – those that are suitable representations of the full input because they are statistically similar enough to it, and those that are not.

⑤ **Select Canary.** Finally, the smallest of the acceptable canaries is returned and used by IRA to perform a dynamic search for the most effective approximation. If no such canary is available, IRA immediately ceases approximation and begins to execute the exact version of the program.

5.3.1.3 Input Similarity Metrics

The purpose of the canary is to drive a dynamic search to determine how the full input to the problem should be approximated. As such, it is of critical importance

that the canary be similar to the full input. However, similarity can be measured in many ways. In this work, we consider four distinct definitions of similarity.

Mean. IRA supports using the arithmetic mean of the values in the canary and full inputs as the similarity metric. We define the mean of an input Y composed of values $y_1, y_2, \ldots, y_N$ as μ_Y. For convenience, the formal definition of μ_Y is supplied in Equation 5.1. A canary found to be acceptable according to this metric has an average value close to the average value of the full input.

$$\mu_Y = \frac{1}{N} \sum_{j=1}^{N} y_j \tag{5.1}$$

Variance. IRA also supports using the variance of values in the input as the similarity metric. The variance of Y is defined as σ_Y^2, the definition of which is supplied in Equation 5.2. A canary that meets this standard of closeness will contain values that are dispersed to a degree similar to the dispersion found in the full input.

$$\sigma_Y^2 = \frac{1}{N} \sum_{j=1}^{N} (y_j - \mu_Y)^2 \tag{5.2}$$

Local Homogeneity. The canary is produced using a subset of the values in the full input. Thus, in essence, a single value in the canary embodies a (potentially large) number of values from the full input. To ensure the values in the canary are not highly dissimilar to the values in the full input they are supposed to embody, IRA leverages a measure of this dissimilarity. We denote this metric Λ_Y, defined by comparing each value y_j in the full input to λ_j, its representative value in the canary, and calculating the proportion of those values that are at least $z_{1-\alpha/2}$ standard deviations (see Table 5.1 for the definition of z) away from λ. The formal definition of Λ_Y is shown in Equation 5.3.

$$\Lambda_Y = \frac{1}{N}\sum_{j=1}^{N}\begin{cases} 0 \text{ if } |y_j - \lambda_j| \leq \sigma_Y z_{1-\alpha/2} \\ 1 \text{ otherwise} \end{cases} \tag{5.3}$$

Autocorrelation. Last, IRA support measuring similarity between a canary and the full input by testing that their autocorrelations are similar. Autocorrelation is a special case of correlation, and is a measure of how similar each value in the input is to its neighbor. High coefficients of autocorrelation (those close to 1) indicate that neighboring values share a linear relationship across the input, while low coefficients (those close to zero) indicate no such relationship. Thus, autocorrelation detects small-scale patterns in the input. For an input Y, the coefficient of autocorrelation is ρ_Y. We provide a formal definition of autocorrelation in Equations 5.4 and 5.5.

$$Y' = \{y_1, y_2, \ldots, y_{N-1}\}, Y'' = \{y_2, y_3, \ldots, y_N\} \tag{5.4}$$

$$\rho_Y = \frac{1}{\sigma_{Y'}\sigma_{Y''}}\sum_{j=1}^{N-1}(y_j - \mu_{Y'})(y_{j+1} - \mu_{Y''}) \tag{5.5}$$

5.3.1.4 Statistical Underpinnings

At its core, canary selection in IRA is built on the statistical foundations of hypothesis testing, in which evidential basis for hypotheses can be weighed statistically, allowing rejection of hypotheses that are not supported by the available evidence. For our purposes, the hypotheses considered are statements such as *canary C_i has the same autocorrelation as the full input.* Such a hypothesis can be rejected if a comparison between the full and canary inputs does not provide sufficient evidence to support the hypothesis. Thus, by rejecting the hypothesis equating the canary to the full input we reject the canary. Alternatively, when the hypothesis test fails to reject the null hypothesis, the canary is deemed to be acceptably similar to the full input.

It is important to note that IRA may need to consider many such hypotheses when constructing a canary, and thus is subject to the *multiple testing problem* [120]. The multiple testing problem describes the situation where evaluating the validity of multiple hypotheses increases the likelihood of incorrectly evaluating at least one of the hypotheses. For example, consider a set of hypotheses concerning the fairness of 100 coins, the validity of which are assessed by flipping each coin 10 times and calling those coins with at least 9 heads or tails biased. Applying this test to unbiased coins, it is unlikely that any particular coin will appear unfair, a probability of 2.1%. However, there is a very strong likelihood (88.6%) that at least one coin will be judged to be unfair, an incorrect determination. Similarly, the multiple testing problem applies to our canary hypotheses. Therefore, when evaluating canaries we adjust our statistical methods by incorporating the Bonferroni correction for Type I errors and the Holm-Bonferroni method for Type II errors to ensure that we avoid the multiple testing problem. These adjustments are discussed in detail shortly.

Hypothesis Testing. In a hypothesis test, we propose two hypotheses relating to the similarity of a canary to the full input. These hypotheses are called the null hypothesis H_0 and the alternative hypothesis H_A. For each canary C_i, we construct null and alternative hypotheses and determine whether to accept or reject C_i based on the evidence found in favor of the null hypothesis. Our discussion will focus on hypothesis testing for the arithmetic mean of the input, however IRA supports several other metrics that have been discussed previously and are summarized in Table 5.1. These other metrics can be used by substituting their equations in place of the equations described for the mean.

A hypothesis test for the mean takes the form shown in Equation 5.6, where $\overline{\mu_i}$ is the sample mean of canary C_i and μ_0 is the sample mean of the full input.

$$H_0 : \overline{\mu}_i = \mu_0$$
$$H_A : \overline{\mu}_i \neq \mu_0 \tag{5.6}$$

Next, the truth of H_0 is evaluated by calculating and evaluating a test statistic. The test statistic is used to produce a p-value for the test, the probability of attaining a test statistic at least as extreme as the observed test statistic given that the null hypothesis is true. Thus, the smaller the p-value, the lower the probability of the observed test statistic appearing if the null hypothesis is true. Some significance level α is chosen as a cutoff point for the hypothesis test, where $p \leq \alpha$ causes the null hypothesis to be rejected and the alternative hypothesis to be accepted. In particular, to compute the t-statistic and p-value for the mean, we use the standard formulas shown in Equations 5.7 and 5.8.

$$t_i = \frac{\mu_0 - \overline{\mu_i}}{\overline{\sigma_i}\sqrt{n}} \tag{5.7}$$

$$p_i = 2P[Z > t_i] \tag{5.8}$$

Standard single comparison hypothesis tests stop here, rejecting the null hypothesis if $p_i \leq \alpha$. However, we must take further steps to avoid the multiple comparisons problem.

Controlling Type I Errors. The Holm-Bonferroni method considers multiple hypotheses simultaneously [86]. It outputs a set of hypotheses that are rejected, and a set that are not, where the probability of obtaining *any* Type I errors is bound by α. The method begins by sorting the p-values $p_1, p_2, \ldots, p_k$ from lowest to highest, resulting in a new indexing of p-values $p_{(1)}, p_{(2)}, \ldots p_{(k)}$ corresponding to null hypotheses $H_{(1)}, H_{(2)}, \ldots H_{(k)}$. It then rejects hypotheses $H_{(1)}, H_{(2)}, \ldots H_{(m-1)}$, where m is the minimal index that satisfies Equation 5.9.

$$p_{(m)} > \frac{\alpha}{k+1-m} \tag{5.9}$$

The result of this method is a set of null hypotheses $H_{(m)}$, $H_{(m+1)}$, ..., $H_{(k)}$ that are not rejected, corresponding to a set of canaries $C_{(m)}, C_{(m+1)}, \ldots C_{(k)}$ that are acceptably similar to the full input.

Controlling Type II Errors. Given desired bounds α and β on the likelihood of getting any Type I or Type II errors, respectively, the standard formula for computing the number of samples needed to ensure the likelihood of getting a Type II error of no more than β in a single comparison hypothesis test is shown in Equation 5.10.

$$n = 2(z_{1-\alpha/2} + z_{1-\beta})^2 \tag{5.10}$$

To account for the multiple testing problem when using k canaries, we use the Bonferroni correction [57], substituting α/k and β/k in place of α and β in Equation 5.10.

$$n = 2(z_{1-\alpha/2k} + z_{1-\beta/k})^2 \tag{5.11}$$

This adjusted formula requires an increased sample size over the non-adjusted formula. However, sampling overhead remains reasonable even for large numbers of canary candidates (large k) because the sample size due to this adjustment grows sub-linearly as k increases [176].

Smallest Acceptable Canary. All of the canaries that remain from the preceding set of steps are acceptably similar to the full input. However, it is important that we choose the acceptable canary that results in the shortest search time in IRA's next step. Thus, this algorithm terminates by choosing the smallest from among the acceptable canaries.

5.3.2 Choosing an Effective Approximation

IRA uses the canary input to rapidly and dynamically decide how to approximate the problem on the full input. This section describes that process.

5.3.2.1 Definition of Result Accuracy

Controlling and maintaining sufficiently accurate computation is important in approximate computing [146, 147, 159]. Prior work has pointed out that result accuracy is domain, application, and context dependent [124] and includes such varied metrics as the scaled difference between output, the peak signal to noise ratio (PSNR) or the average absolute output accuracy. Therefore, we design IRA to be agnostic to the specific method used to calculate result accuracy. That is, we assume only that the application developer provides a well-defined accuracy calculation function. Our formulation of this function $F_{accuracy}$, given two solutions S_{exact} and S_{approx}, computes a single accuracy metric $\delta \in [0, 1]$ describing the accuracy of S_{approx} relative to S_{exact}. $F_{accuracy}$ is leveraged by IRA to compute the accuracy of a number of approximations on the canary input, comparing them to the solution produced by the exact method on the canary input. We assume also that the user of the application supplies a minimum acceptable result accuracy, called the target output quality (TOQ).

5.3.2.2 Where and How to Approximate

There may be a number of code regions amenable to approximation in an application. Consider an application with two disjoint loops that can be approximated with loop perforation and tiling, respectively. Each such code region is an approximation opportunity, and IRA treats each approximation opportunity as one dimension in a multi-dimensional search space by encoding the parameters for each approximation opportunity as a range of numerical values $\{1, \ldots, v\}$. In the encoding, the value 1 has special meaning, and is used to represent the exact computation in lieu of

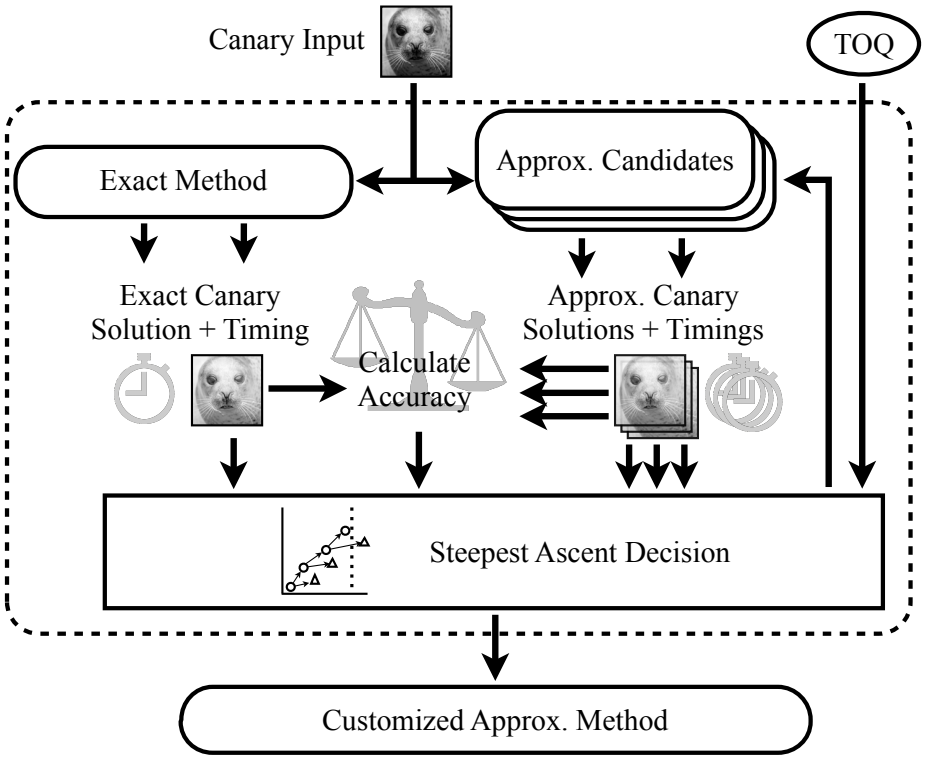


Figure 5.5: Search for approximation using canary

approximation.

Furthermore, many approximation techniques may be parameterized, such as the rate at which iterations are skipped in loop perforation or the size of one side of a tile in tiling approximation. In such cases, numbers larger than 1 encode each value that can be taken by a parameter. Our search algorithm makes only the assumption that larger values correspond to more aggressive approximation (i.e., that it runs faster but has lower accuracy). By encoding the search space in this fashion, IRA has the option to select the exact computation at each approximation opportunity, allowing it to choose where to approximate. By selecting between the values larger than 1, IRA determines how aggressively to take advantage of each approximation opportunity.

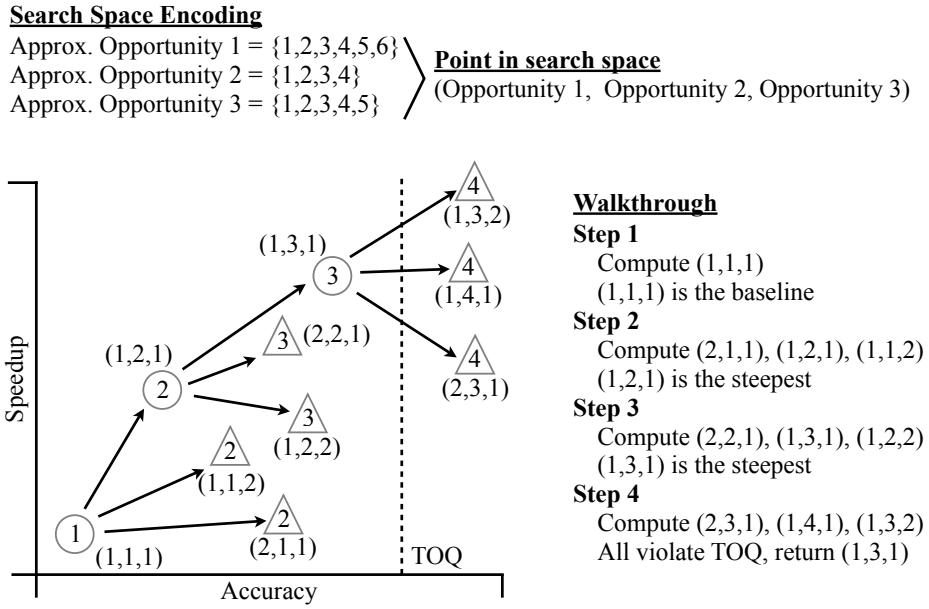


Figure 5.6: Example search for an effective approximation

5.3.2.3 Search for an Effective Approximation

IRA uses a greedy approach based on steepest ascent hill climbing [144] to tune the parameters for the available approximations, using the approach presented in Figure 5.5. An approximation option is defined as a point in an m-dimensional space $(d_1, d_2, \ \ldots \ , d_m)$, where each dimension is an encoded range of numerical values as described in the previous section. IRA first evaluates the point $(1, 1, \ \ldots \ 1)$ on the canary, which is the exact solution to the problem on the canary. This solution is used as a timing and accuracy baseline, against which approximate solutions are evaluated. Beginning at $(1, 1, \ \ldots \ 1)$, IRA then iteratively evaluates the incrementally more aggressive value for each of the tuning parameters, computing the accuracy and speedup relative to the exact version, then selects the increment that both satisfies the TOQ set forth by the user and yields the steepest slope in terms of accuracy vs. speedup. If no such increment exists, the search terminates. If such an increment exists, it is used as the starting point for the next iteration. Upon termination, the last valid point is returned by the search and is used to approximate the full problem

input.

The search process runs the exact computation in addition to a number of approximation alternatives on the canary. The execution of this search typically very fast relative to exact execution on the full input because the amount of computation needed in regularly structured computation depends substantially on the size of the canary, which is much smaller than the full input. We quantitatively evaluate the time spent in the search in Section 6.4, showing that it equates to an average of 3.2% of exact execution time on a suite of test applications.

Example. Consider Figure 5.6, which shows an example search over 3 approximation opportunities. In step 1 we evaluate $(1, 1, 1)$ as a baseline. In step 2, the next available increment along each dimension is tested – $(2, 1, 1)$, $(1, 2, 1)$ and $(1, 1, 2)$ in this case. $(1, 2, 1)$ is found to have the steepest ascent in the speedup/accuracy space, and is used as the baseline for the next step. In step 3, $(2, 2, 1)$, $(1, 3, 1)$ and $(1, 2, 2)$ are tested, and $(1, 3, 1)$ is found to have the steepest ascent. Finally, in step 4 $(2, 3, 1)$ $(1, 4, 1)$ and $(1, 3, 2)$ are tested. Each is found to violate the TOQ bound and the search halts, returning $(1, 3, 1)$ as the most effective approximation.

Application	Description	Domains	Input Suite	Approximation(s) Used
CrossCorr	Measure signal/image similarity over sliding window	Pattern recognition, cryptanalysis, neurophysiology	800 IMAGE	4× extrapolate, 1× 2D-tile
FuzzyKmeans	Cluster with fuzzy cluster membership	Machine learning, data mining	4 SVM	5× perforate
Gamma	Apply gamma correction to an image	Image processing	800 IMAGE	1× 2D-tile
GaussianFilter	Apply a Gaussian filter to an image	Image processing	800 IMAGE	1× 2D-tile
Integration	Numeric integration of transcendentals	Scientific computing, engineering	19 EQN	1× numerical approx.
Inversek2j	Kinematics for 2-joint arm	Robotics	90 ANGLE	4× numerical approx.
Jmeint	Triangle intersection detection	3D gaming	40 TRI	1× algorithm choice
LucasKanade	Optical flow estimation	Computer vision	2 PERFECT	2× perforate
Kernel	Estimate a probability density function	Machine learning, signal processing, econometrics	2 KDDCUP	4× perforate
Kmeans	Cluster points for classification	Mach. learning, data mining	4 SVM	4× perforate
MatMult	Matrix-matrix multiply	Machine learning, scientific computing, game theory	40 PDF	2× perforate
MeanShift	Apply mean shift to an image	Computer vision, image processing	4 SVM	3× perforate
ScalarProd	Dot product of two vectors	Mechanics, machine learning, graphics	40 PDF	2× perforate

Input Suite	Description
90 ANGLE	Sets of angles drawn from 90 different probability distributions
19 EQN	Sets of equations containing polynomials with different max degree
800 IMAGE	A database of 800 images
2 KDDCUP	1999 KDD Cup data set from the UCI Machine Learning Repository [13]
40 PDF	Probability dists used: beta, binomial, cauchy, chi-squared, exponential, f, gamma, geometric, hyper, log-normal, normal, poisson, t, uniform, weibull
2 PERFECT	Medium and large inputs from the PERFECT benchmarks [18]
4 SVM	Support vector machines from the UCI Machine Learning Repository [13]
40 TRI	Sets of triangles in the unit cube, varying distributions of triangle sizes

Table 5.2: Applications and input sets used in the evaluation

5.3.3 Putting it all Together

Final Approximation. The previous section described how IRA derives an approximation method that is effective for the canary input. This approximation method, which has been customized to be effective on the canary input, is then deployed on the full input, producing an approximate result that is of acceptable quality.

Runtime Safety. In approximate computing, altering computation to trade performance for accuracy, particularly when discarding computation, can have the effect of changing control flow, producing unsafe intermediate results (e.g., a 0 that will be used as the denominator in a division operation), or memory accesses that corrupt state or result in access violations, resulting in runtime faults that were not anticipated by the application programmer. Prior work has shown that it is often possible to recover from memory errors using checkpointing [158] or heap replication [20], and from floating point errors using reevaluation or rollback [78], resuming computation to successfully produce a result. We have experienced no such faults in our experiments, however IRA can be augmented in the future to include mechanisms to guard against these faults.

5.4 Evaluation

We evaluate IRA to examine its impact on performance and result accuracy for applications spanning a number of computational domains.

5.4.1 Methodology

Applications and Inputs. We evaluate 13 applications that use between 2 and 800 inputs. These applications cover a number of the important problem domains that include image processing, data mining, machine learning and computer vision. Applications and inputs are summarized in Table 5.2.

Approximation Techniques. Our experiments bring input responsiveness to four classes of approximation techniques. The approximation techniques themselves have limited in terms of how they can be applied to software. For example, tiling approximation requires iterative computation on image pixels thus is applied only to CrossCorr, Gamma and GaussianFilter, while numerical approximation requires a library call to a math function (e.g., trigonometric functions in Inversek2j). In many cases, multiple approximations are used side by side among an application's code regions. A summary of which approximations are applied to which benchmarks is summarized in Table 5.2. The approximations techniques include:

- **Loop Perforation [84, 140]** – loop perforation discards iterations in a loop. We use either unadjusted perforation, in which every nth iteration in a loop is executed, or extrapolated perforation, which is similar to unadjusted perforation but extrapolates computed results to make up for the skipped iterations. Loop perforation can be made more aggressive by using larger values of n. We use the loop perforation in CrossCorr, FuzzyKmeans, LucasKanade, Kernel, Kmeans, MatMult, MeanShift and ScalarProd.

- **Tiling [146]** – instead of computing each element of an output, a tiling approximation computes a single output element and projects it onto the surrounding elements to form a tile. Tiling approximation is made more aggressive by using larger tile sizes. We use the tiling approximation in CrossCorr, Gamma and GaussianFilter.

- **Algorithmic Choice [8, 55]** – we use IRA to choose between five different algorithmic implementations of Jmeint that offer different accuracy-performance tradeoffs in computing whether pairs of 3D triangles intersect. The most complex algorithm is the exact algorithm, while the simplest algorithm uses computationally cheap heuristics that work well only when triangles are far apart.

- **Numerical Approximation [81]** – we employ numerical approximation techniques within Integration and Inversek2j. Integration numerically integrates a non-integrable set of equations using the trapezoid method, which can be made faster and less accurate by using fewer trapezoids. Inversek2j involves a motion calculation that relies on the trigonometric functions $sin(x)$, $cos(x)$, $sin^{-1}(x)$, $cos^{-1}(x)$. We approximate these trigonometric functions by using the first 1 (sin and sin^{-1}) or 2 (cos and cos^{-1}) terms of the function's Taylor series in lieu of the precise library implementation. These approximations are accurate when x is near zero, and become less accurate farther away from zero. Thus we can trade speed for accuracy by choosing a k such that approximation is used only when $|x| < k$, making the approximation more aggressive by using larger values of k.

Platform. All results are collected on a stock 2.4GHz Intel Xeon E5-2407v2 (Ivy Bridge) server running Linux kernel 3.11.0. Applications are executed on the server in serial, and task pinning is used to prevent process migration.

Error Bounds, TOQ and Accuracy. All experiments in this evaluation use canary error bounds $\alpha = \beta = 0.05$, thus obtaining Type I and Type II error bounds of 0.05. TOQ values ranging from 90% to 97.5% are used in the evaluation, specified for each experiment. IRA is agnostic to the accuracy metric, simply using the supplied definition of accuracy (see Section 5.3.2.1) and configuring the approximation so as to not violate the accuracy target set forth by the user. In our evaluation, we use miss rate as the accuracy metric for Jmeint, absolute relative error for ScalarProd and average centroid distance from the origin in Kmeans and FuzzyKmeans. For all other applications, accuracy is defined as the average of element-wise absolute relative error.

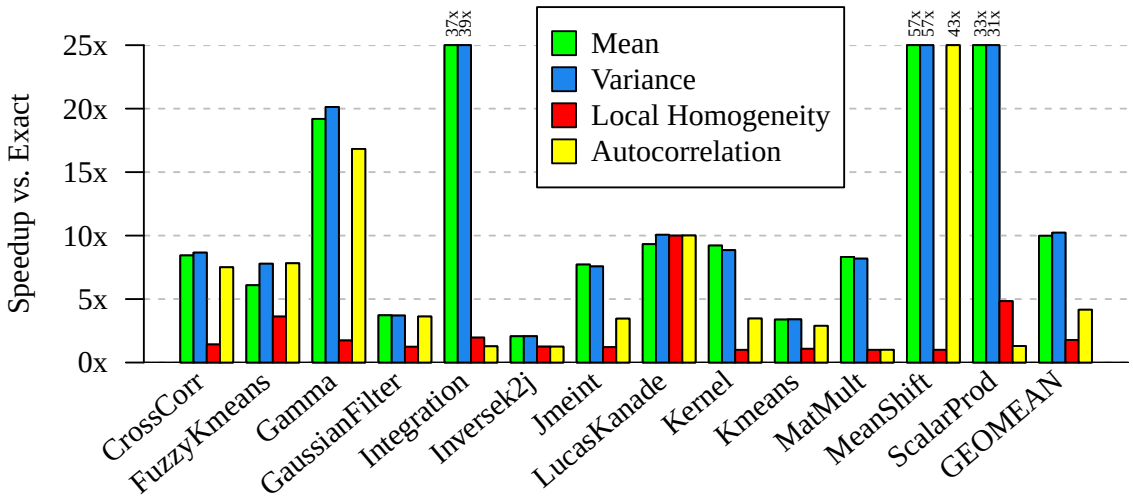


Figure 5.7: Comparison of canary similarity metrics

5.4.2 Canary Construction

Similarity Metrics. Figure 5.7 presents the speedup over exact computation obtained by IRA when employing each of the four canary similarity metrics described in Section 5.3.1 at a TOQ of 90%. As the figure illustrates, the largest speedups obtained are for Variance and Mean, which average 10.2× and 9.9×, respectively. The speedups obtained when using autocorrelation are modest, averaging 4.3×, while using local homogeneity causes a speedup of 2.1×.

This is a somewhat surprising result, as the simpler metrics – Mean and Variance – perform better while achieving similarly low counts of TOQ violations (TOQ is violated on less than 1% of inputs on average for all metrics). Closer inspection reveals that autocorrelation and local homogeneity are more difficult similarity metrics to satisfy, thus they often result in either (1) choosing larger canaries, leading to longer search times, which diminishes the overall speedup or (2) finding no acceptable canaries, and thus no approximation being used. This is particularly true for local homogeneity, which achieves speedups near 1 for 8 of the 13 applications. This leads us to the insight that more complicated is not always better; autocorrelation

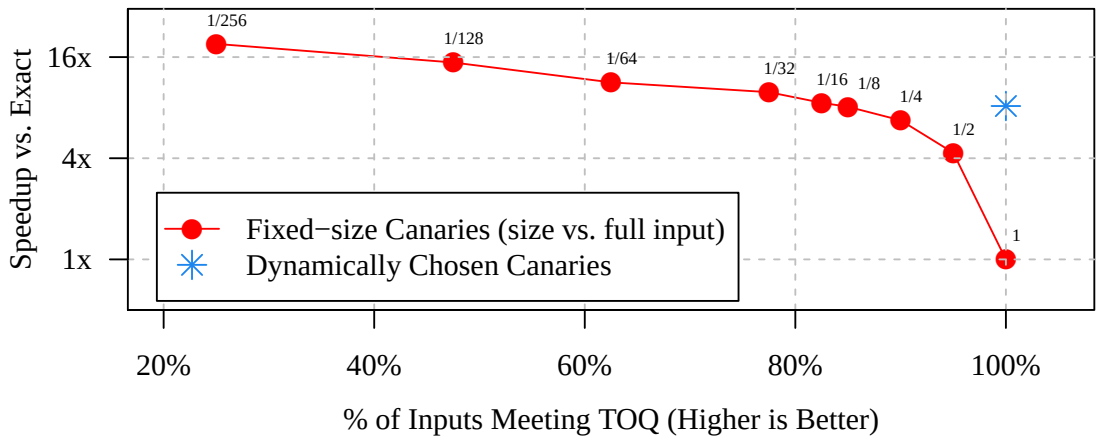


Figure 5.8: Speedup and number of TOQ violations for dynamically chosen canaries (blue star) vs. fixed-size canaries (red circles) on MatMult; all fixed size canaries achieve lower speedup, more TOQ violations, or both

and local homogeneity reject many canaries that are deemed acceptable according to their mean and variance, and which turned out to be perfectly adequate in searching for an effective approximation.

A second insight revealed by this data is that mean and variance do not significantly differ from one another in terms of the canaries selected, a fact which that holds true on average and among the individual applications. This suggests that both metrics produce reasonable canaries and function well across a range of problems and domains. Because variance is a slight improvement over mean in terms of the overall speedup of IRA, the remainder of the experiments use variance as the similarity metric when constructing canaries.

Dynamically-Sized Canaries. Dynamically-sized canaries are valuable because they avoid two pitfalls that could occur when creating fixed-size canaries across all inputs. They are no larger than necessary for "well-behaved" inputs, thus keeping the overhead low during the approximation search process, and they are large enough to adequately represent the full input, resulting in a search that yields approximations that are just aggressive enough for each input.

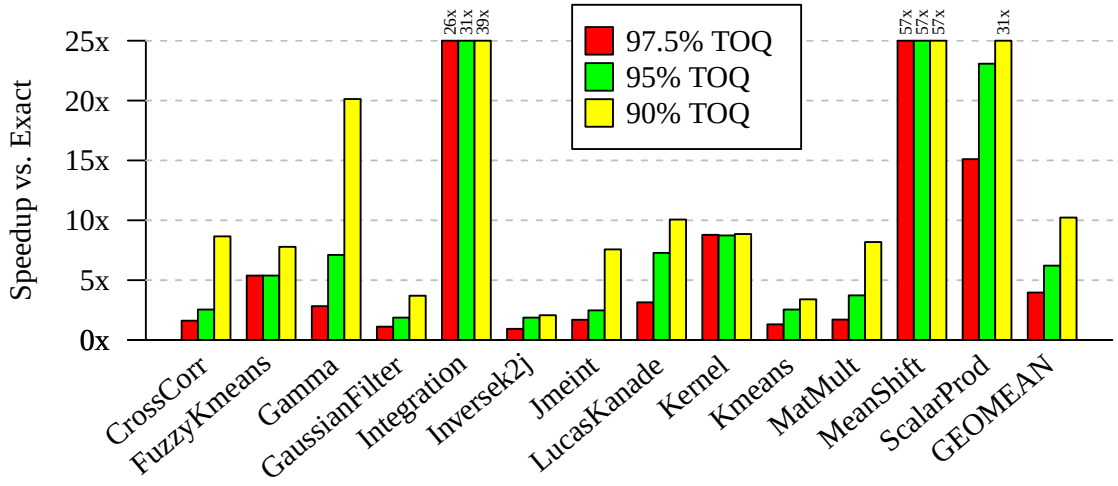


Figure 5.9: Speedup of IRA across three TOQs

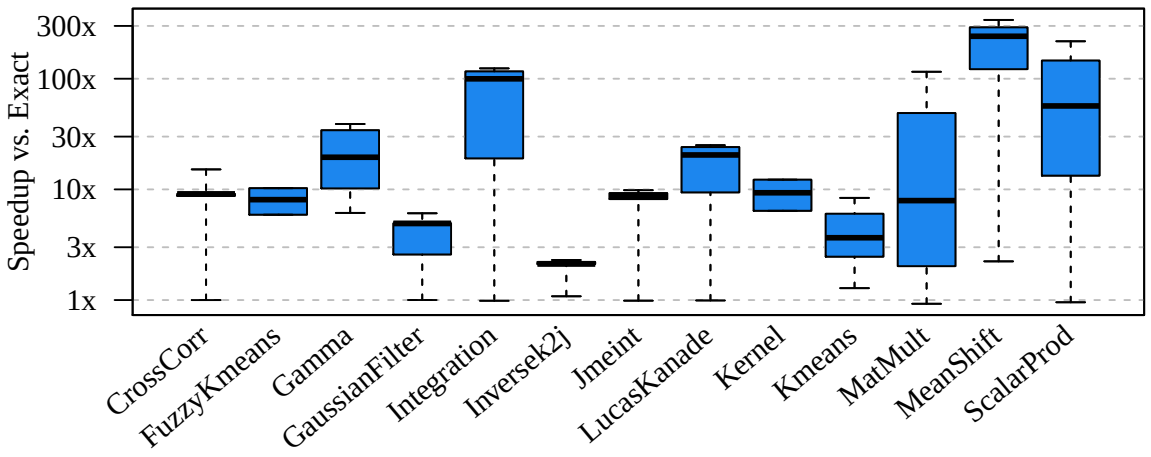


Figure 5.10: Distribution of speedups across inputs for IRA at 90% TOQ, illustrating the wide range of approximations dynamically chosen across different inputs; larger speedups occur when more aggressive approximation is applied

To illustrate this, we compare the results of using different fixed-rate canaries within IRA for approximating MatMult to using dynamically-chosen canaries at a TOQ of 90%. The inputs to MatMult are the set of 40 inputs described in Table 5.2, spanning a range of probability distributions that include long tail and high variance distributions. The results are illustrated in Figure 5.8, which shows the speedup (y-axis) and number of inputs meeting TOQ (x-axis) achieved by IRA when using a range of fixed-size canaries (red circles and line). As the figure shows, there is a tradeoff between speedup and input violations to be made when using fixed-size canaries: smaller canaries produce larger speedups but large numbers of TOQ violations, while larger canaries produce fewer TOQ violations but smaller speedups. Improving in both dimensions is the point illustrating the speedup (8.2×) and TOQ violations (0%) achieved when using dynamically-chosen canaries (blue star). This demonstrates the advantage of using dynamically-chosen canaries. A small canary is used when a small canary can serve as a suitable representation of the full input, while a large canary is used when a small canary cannot.

5.4.3 IRA Speedup and Accuracy

Speedup. We refer next to Figure 5.9, which presents the average speedup achieved by IRA relative to the runtime of the exact computation across three TOQ values: 97.5%, 95% and 90%. Each application is run on all inputs, and the speedups presented are the geometric mean of speedup across the inputs. Performance measurements of IRA are the end-to-end runtime, including the time to produce the canary input, search for the customized approximation and run that approximation on the full input. As one would expect, IRA achieves speedups that scale up as the TOQ is relaxed, ranging from an average of 3.9× at 97.5% TOQ up to 10.2× at 90% TOQ.

Dynamism. Figure 5.10 presents boxplots of the speedups achieved across inputs for each application at TOQ=90%. The boxplots highlight the maximum (upper

Application	Meets TOQ (% of Inputs)	Application	Meets TOQ (% of Inputs)
CrossCorr	790 / 800 (98.8%)	FuzzyKmeans	4 / 4 (100%)
Gamma	752 / 800 (94.0%)	GaussianFilter	797 / 800 (99.7%)
Integration	19 / 19 (100%)	Inversek2j	90 / 90 (100%)
Jmeint	40 / 40 (100%)	LucasKanade	4 / 4 (100%)
Kernel	2 / 2 (100%)	Kmeans	4 / 4 (100%)
MatMult	40 / 40 (100%)	MeanShift	4 / 4 (100%)
ScalarProd	40 / 40 (100%)	**MEAN**	**99.4%**

Table 5.3: The proportion of inputs for which IRA hits the target output quality (TOQ) at TOQ=90%

whisker), 75th percentile (box upper edge), median (line within box), 25th percentile (box lower edge) and minimum (lower whisker) speedups. The large range of speedups shown in Figure 5.10 highlights the key feature of IRA: different inputs to the same application can be more or less difficult to approximate. IRA takes advantage of these differences to choose the right approximation for each input and maximize the performance that can be gained when applying approximation.

This dynamism allows IRA to realize significantly higher performance for many cases that cannot be taken advantage of by approaches that apply one approximation across inputs. Consider an oracle approach to choosing the single best approximation approach across all inputs. Even with full knowledge of all inputs and how each of the approximation options available to IRA fares on those inputs, we find that such an oracle can achieve an average speedup of only 4.9×, as opposed to 10.2× speedup with IRA, while delivering the same level of accuracy as IRA

Accuracy. The accuracy of the results produced by IRA is presented in Table 5.3, showing the number of TOQ violations across inputs at TOQ=90%. On average, IRA meets TOQ for over 99% of inputs. Furthermore, for 10 of 13 applications there are no output quality violations, and the maximum proportion of TOQ violations is 6% for Gamma. Moreover, those cases that violate TOQ are typically not far from TOQ. For instance, 78% of violating cases have an output quality of 88% or better (within 2% of TOQ). From this we conclude that IRA works very well at producing a minimal number of TOQ violations in practice, however we take care to note that

IRA makes no guarantees about output accuracy.

5.4.4 Where is the Time Spent?

Figure 5.11 presents a breakdown of the time spent by IRA (TOQ=90%) as a fraction of the total runtime of the exact application run. These percentages are the average across all inputs for each application. The bulk of the time shown in the figure is execution time saved by approximating the application with IRA. We divide the time spent by IRA into three parts: the time spent creating a canary input (green; barely visible), the time spent using the canary to search for an approximation (blue), and the time spent running the chosen approximation on the full input (red).

The time spent choosing the canary is small, which is to say that the remaining bottlenecks in IRA are elsewhere. Many applications – Gamma, Integration, Inversek2j, Jmeint, Kernel, MatMult, MeanShift and ScalarProd – spend a small proportion of the time searching for the the approximation, while many others spend a sizable fraction of time doing so. When a large amount of time is spent in the search, this is caused by a combination of large canaries and high-dimensional search spaces (that is, those that have a larger number of approximation opportunities to explore).

The size of the approximation search spaces varies significantly across applications, ranging from 4 in the case of Jmeint (4 versions of the algorithm constitute the search space) to 22,500 in the case of CrossCorr where 5 approximation options are parameterized. Our hill climbing algorithm takes O(m*n) steps, where m is the number of approximations to parameterize and n is the number of ways to parameterize a single approximation. In our experimentation, we have found that searches often end in fewer than 10 steps and typically take no more than a few dozen steps, ultimately resulting in searches that average 3.2% of exact execution time.

The search time for choosing the approximation might be reduced by paring down

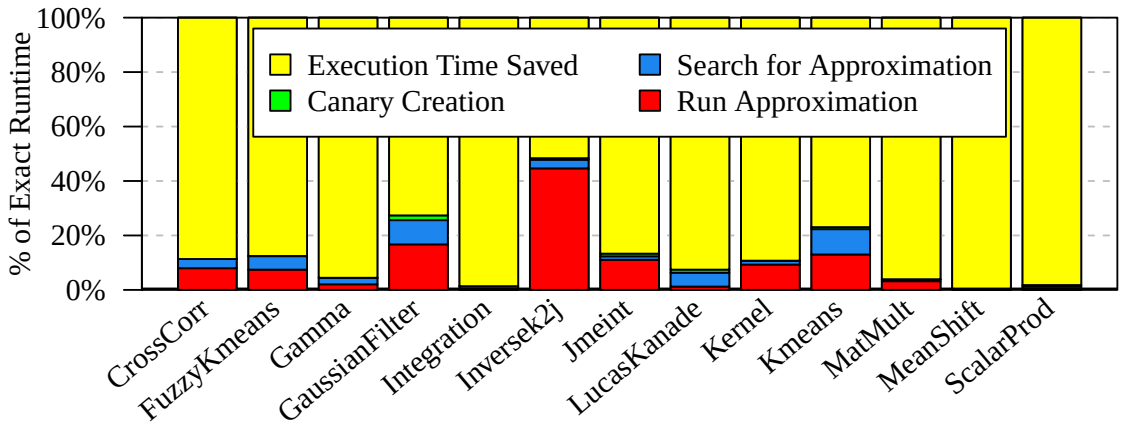


Figure 5.11: Breakdown of time spent by IRA, showing time to create the canary (barely visible), search for the approximation, and run the chosen approximation on the full input

the number of approximation opportunities and parameter ranges to reduce the size of the search space. For example, if certain approximation opportunities were revealed through static analysis, offline profiling, or feedback from earlier runs of IRA to result in ineffective approximations for a substantial fraction of inputs, those opportunities could be discarded. However, because the goal in this work is to automate the process of choosing the approximation without the aid of offline profiling or analysis, we implement no such feedback loop.

5.4.5 Comparison to Prior Work

Green [15] and SAGE [147] are two state-of-the-art calibration systems that dynamically tune approximation to control TOQ violations. Green uses profiling in concert with calibration at fixed periods to tune how aggressively to apply approximation. SAGE is also calibration-based, however it is entirely dynamic in nature and it continually changes the calibration period as more inputs are seen, lengthening the period when calibration shows that the current tuning of the approximation does not violate TOQ.

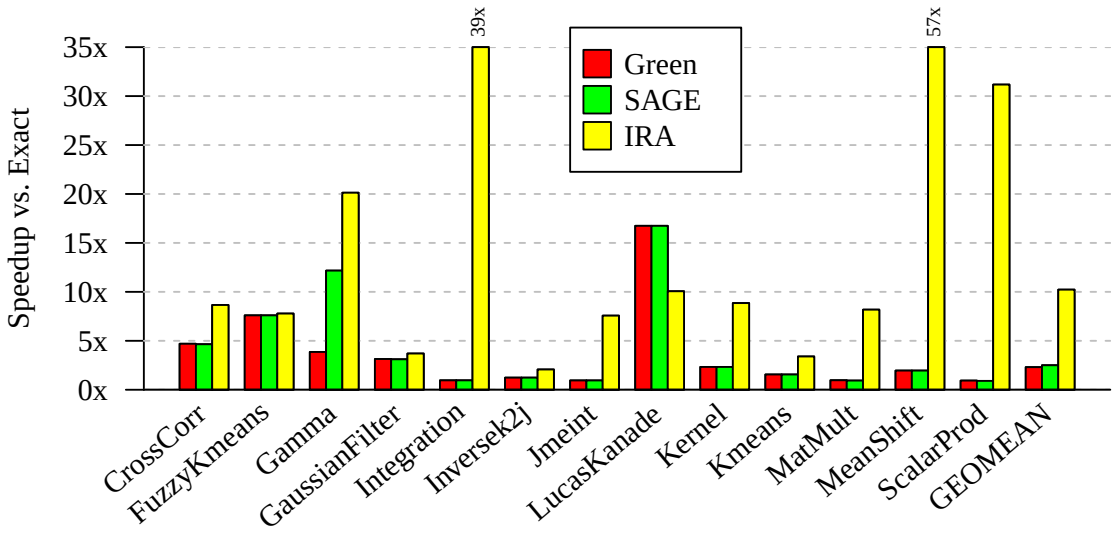


Figure 5.12: Comparison of IRA to calibration-based approximation with Green [15], SAGE [147], showing that IRA achieves more than 4× speedup of each

We compare IRA to using oracle versions of the SAGE and Green runtime systems to choose approximations. Our implementations of the SAGE and Green runtimes do two things perfectly that cannot be done in practice. First, calibration on an input yields the precise speedup and accuracy for that input on all approximations, allowing the approximation to be tuned to exactly the most effective approximation at each calibration point. Second, calibration intervals for each approach are set by an oracle for each application. For Green, the best calibration interval is used out of all possible calibration intervals, and for SAGE both the best calibration interval and calibration adjustments are used. Thus, our experiments are upper bounds for the speedups achievable on these applications, inputs and approximation techniques with Green and SAGE.

We compare IRA against the oracle Green and SAGE by holding the number of TOQ violations achieved by each approach constant and examining the speedups achieved. The results of this comparison are shown in Figure 5.12, which shows the speedup of the three techniques where the TOQ violations are held constant at the TOQ violations IRA achieves at 90% TOQ. IRA improves the performance by an

average of 10.2× by customizing the approximation to each individual input, while oracle Green speeds up by an average of 2.2× and oracle SAGE speeds up by a factor of 2.3×.

There are a number of applications for which the oracle Green and SAGE provide no speedup, such as MatMult and ScalarProd. This occurs because some of the application inputs have high variance or long tails, making them very difficult to approximate. For these applications, Green and SAGE get locked into unnecessarily conservative approximation approaches for a series of inputs once calibration has been done on a difficult input. IRA, on the other hand, employs conservative approximations on these difficult inputs while applying appropriately aggressive approximation on others. On LucasKanade, Green and SAGE achieve more speedup than IRA. This occurs because LucasKanade can be aggressively approximated on all inputs, thus allowing Green and SAGE to calibrate once and run those aggressive approximations for all input, whereas IRA spends valuable time searching for an approximation on all inputs.

5.5 Summary

This chapter described Input Responsive Approximation, an approach for automatically configuring approximation of regularly structured computations for each problem input. IRA accomplishes this by producing a *canary input* at the problem outset, a reduced version of the full input rigorously chosen so as to retain the properties of the full input. This canary is used to rapidly test and choose from among the available approximations. We use IRA to approximate 13 image processing, machine learning, data mining, and computer vision applications. Using these applications, we showed that IRA achieves an average speedup of 10.2× at a target output quality of 90%, far higher than idealized versions of state-of-the-art prior work.

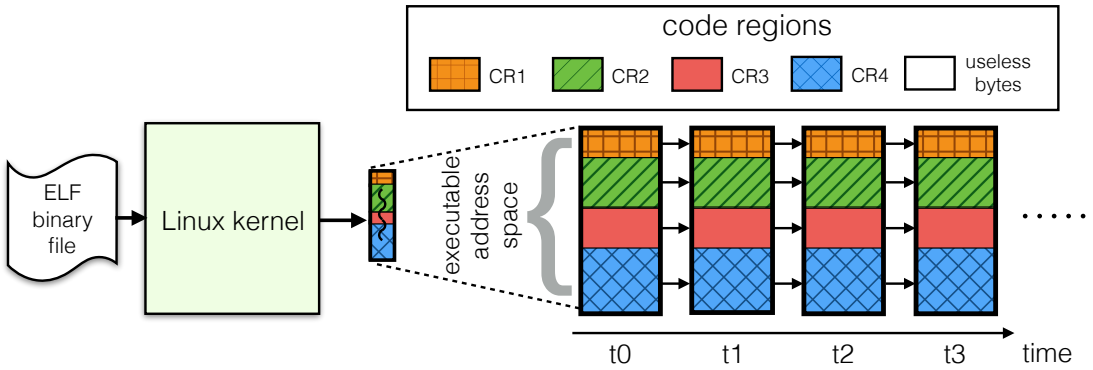
CHAPTER VI

Online Code Transformations in the Operating System for Increased Security

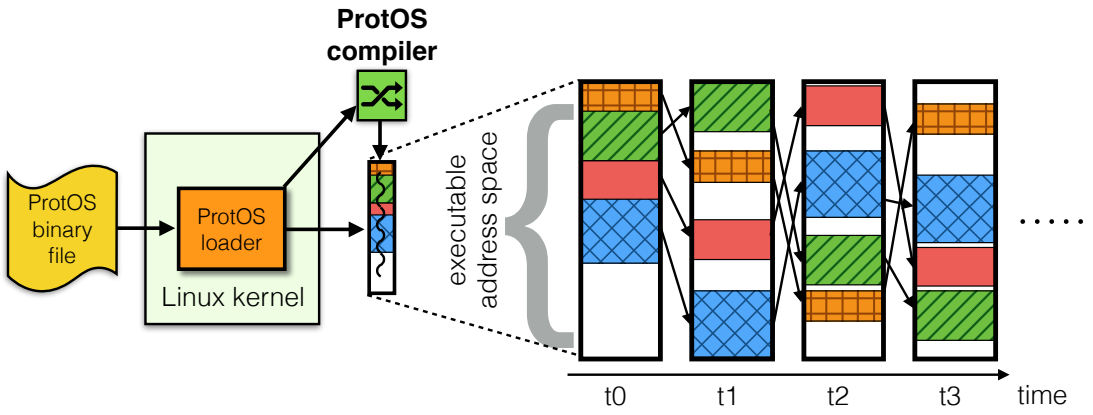
This chapter describes the motivation and design of a novel operating system architecture with an online code transformation capability for increased security. We show that this code transforming operating system, *ProtOS*, can be used to design a new class of protections that undermine code reuse attacks by employing continuous code re-randomization. Continuous code re-randomization constantly re-positions and reorganizes a program's executable bytes while the program runs, thwarting an important class of execution hijacking attacks by leaving attackers unable to exploit assumptions as to the location and structure of code in memory.

ProtOS introduces the robust code transformation power of a dynamic compiler into the OS, enabling a new class of security techniques. This novel design allows the OS to continuously transform executing code with low overhead on arbitrarily complex native programs. Hosting the code re-randomization technique in the operating system has the security advantage of protecting critical structures used for runtime code transformations.

The operation of the code re-randomization service is illustrated in Figure 6.1. Figure 6.1(a) shows execution of a process over time using a conventional system



(a) Conventional OS that uses fixed, randomized code locations



(b) ProtOS uses continuously re-randomized code locations

Figure 6.1: ProtOS thwarts code reuse attacks by using its online code transformation capability to continuously re-randomize code as the program runs

architecture that leaves code locations fixed throughout execution. Figure 6.1(b) depicts the code re-randomization service in ProtOS, which continuously iterates over the code in the program, leveraging a robust dynamic compiler to generate re-randomized variants of program code. Our evaluation of ProtOS shows that it can re-randomize program code frequently enough to resist state-of-the-art code reuse attacks based on memory disclosures and side channels with modest performance overheads that average 9% across a wide range of applications.

6.1 Why a Code Transforming OS?

Address Space Layout Randomization (ASLR) is a technique that randomizes the position and contents of code and data once as the application begins running. Coarse-grain forms of ASLR are implemented in most modern operating systems, randomizing the position of certain segments at load-time. The main limitation of ASLR in the presence of sophisticated code reuse attacks is its static nature, only randomizing code once at the beginning of execution, which results in a lack of protection throughout an application's lifetime. A defensive capability that offers continuing protection during an application's execution is needed to defend against such attacks.

The primary insight underlying the need for a code transforming OS is that constructing code reuse attacks takes time, ranging from hundreds of milliseconds to minutes and even weeks [154, 156, 162]. Thus, if the locations of executable code can be changed between the point in time when memory becomes visible to the attacker and the point in time when the attacker has successfully constructed the attack based on that knowledge of memory and begins deploying it, the attack will not succeed. In other words, these attacks can be thwarted by a practical mechanism for dynamically and continuously re-randomizing code. There are challenges in designing such a mechanism for online code transformation – that mechanism must itself be secure, transparent, robust and efficient. To overcome these challenges, this work proposes a

compiler-empowered operating system architecture than can continuously transform running application code.

6.1.1 Decoupled Application and Compiler

Classical approaches to dynamic compilation and optimization are based on virtualization, whereby the application and dynamic compiler are tightly coupled at runtime and control passes back and forth between them to allow the compiler to generate code and control execution. Even without performing any code transformation, the most of efficient of this class of systems introduces 20-30% overhead just to possess the dynamic compilation capability [30]. Instead, our approach is based on protean code. The key element in the design of protean code is that it decouples the dynamic compiler from the application to allow the application to run at near-native speeds ($< 1\%$ overhead) while the compiler works in parallel to generate and stitch in code without requiring access to the original source code. We thus design a system that spawns a dedicated lightweight dynamic compilation process as it spawns the application process, which runs in parallel to the application and transforms its code as they both execute.

6.1.2 OS-Hosted Online Code Transformation

The dynamic compilation process thus acts as a transparent service (that is, the application needs no knowledge of it) that transforms application code as it runs. This design of dynamic compilation as a transparent service is essential for keeping the dynamic compiler from being subverted or disabled, and by building the dynamic compilation capability into the operating system we are able carefully construct it to protect critical structures in the dynamic compiler.

Unlike userspace dynamic compilation, a dynamic compilation process in our system architecture is designed specifically to avoid interaction with other processes

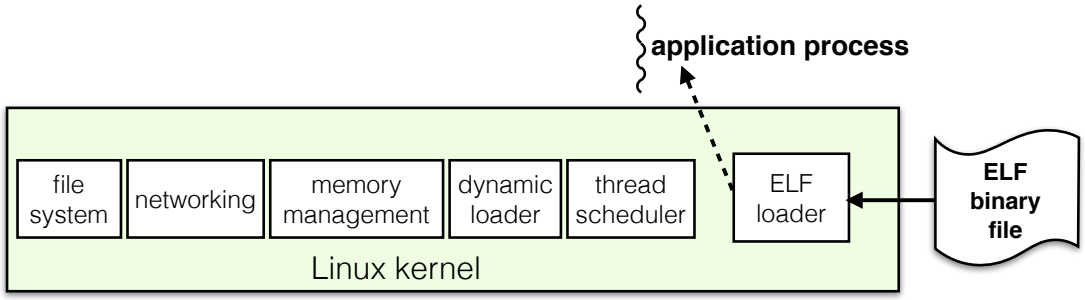
and subsystems. For example, it has no input beyond the process ID and metadata of the host program, supplied by the kernel when invoking the dynamic compiler. It has no environment variables, ignores signals generated by other userspace processes, performs no device I/O, and only interacts with the file system when loading dynamically-linked libraries (see Section 6.2.4). Thus, file I/O can be avoided entirely by static-linking programs. Avoiding these interactions as much as possible is particularly important for designing security mechanisms such as code re-randomization, as it minimizes the attack surface of the dynamic compilation process and makes it difficult for attackers to leverage the dynamic compilation capability in attacks.

6.1.3 Beyond Security

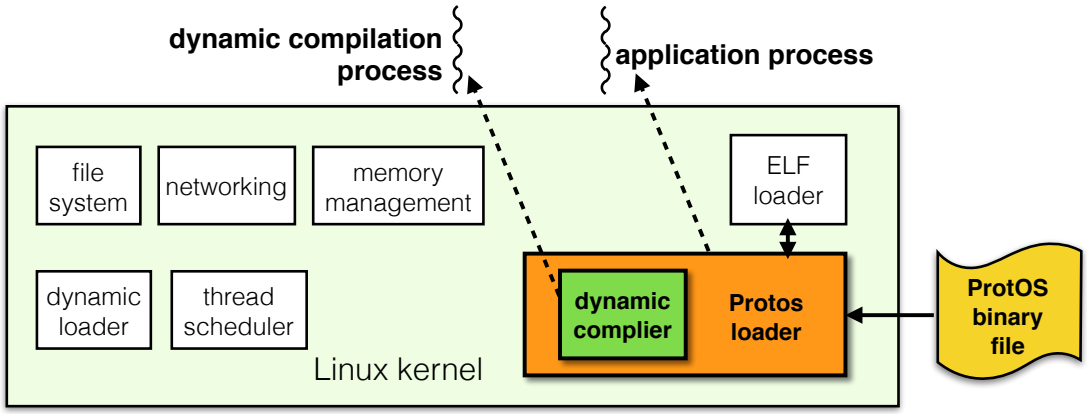
Beyond the security focus of this chapter, an enhanced system architecture with an online code transformation capability opens up a new design space for other aspects of the OS. The OS hosts a number of services that can benefit from a dynamic compilation capability to improve performance. For example, a task scheduler could transform application code online to share resources more effectively [103, 169, 170] or to dynamically invoke core-specific optimizations in heterogeneous multi-core systems as it moves tasks among cores with different capabilities [45, 74]. Similarly, a memory manager could use page management policies in concert with online code transformation to improve the performance or predictability of the TLB [14, 93, 134].

6.2 ProtOS System Architecture

This section describes the design and implementation of ProtOS, our realization of a code transforming operating system. We describe the main elements of ProtOS and how it extends a traditional system architecture to include a code transformation capability. These extensions are described in the context of our prototype implementation of ProtOS on top of Linux/x86_64, though the concepts involved are generic



(a) Conventional system architecture



(b) ProtOS system architecture

Figure 6.2: System architecture of ProtOS

and make minimal assumptions about the underlying system.

6.2.1 Overview

Figure 6.2 contrasts the design of ProtOS with a conventional system design. We highlight the modifications to the existing system architecture in the figure, which primarily include a light-weight dynamic compiler and an extended program loader. The ProtOS loader is an extension to the existing ELF loader in Linux that sets up the binary, configures a dynamic compilation process, then launches that dynamic compilation process alongside the application. Throughout application execution,

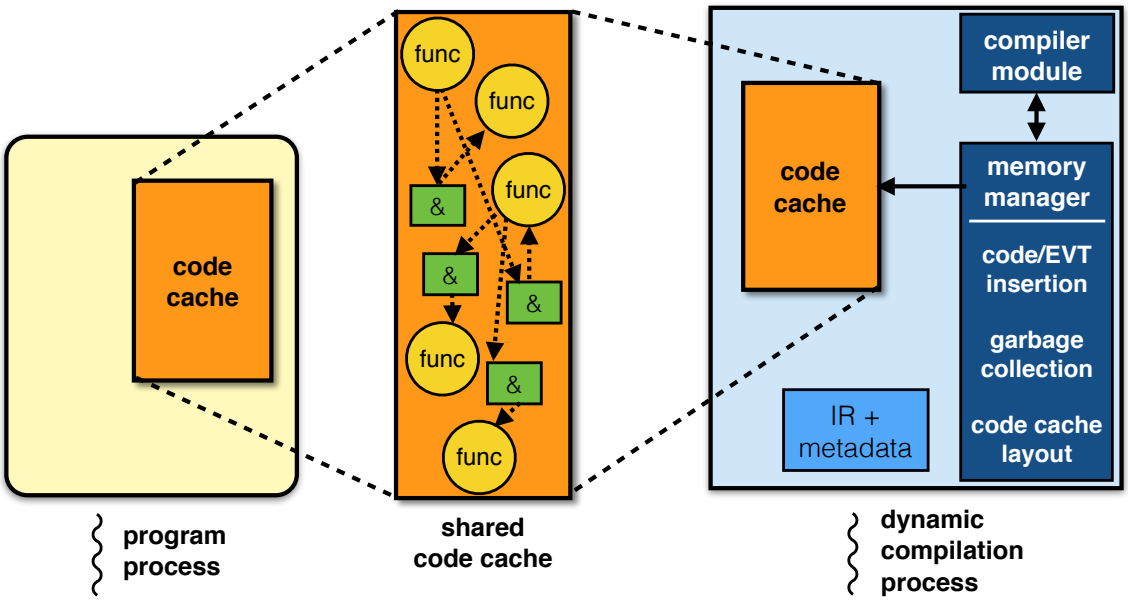


Figure 6.3: Overview of ProtOS runtime system. All program execution occurs from the code cache, a shared memory region between the program and the compiler. The dynamic compiler runs asynchronously to update the code cache

the dynamic compiler spawned by ProtOS provides online code transformation as a transparent service to the application, acting on native application binary in memory with little overhead. We describe the compiler component and the loader in further detail in Sections 6.2.2 and 6.2.3, respectively.

6.2.2 Online Code Transformation

The online code transformation facilities in ProtOS are based on the techniques described in Chapter III. First, at compile time, the direct calls in the program are virtualized (refer to Figure 3.1), a step that provides points at which the application's execution can be dynamically controlled. As we show later, this virtualization step introduces negligible runtime overhead. When an application compiled this way runs, a dynamic compilation process executes alongside to generate and stitch in newly produced code, shown in Figure 6.3. The dynamic compilation process places

newly compiled code into a code cache, a region of memory shared between the host application and the dynamic compiler. When the new code is ready for execution, the dynamic compiler simply overwrites the target of the relevant virtualized call instruction. The decoupling of the application and compiler has significant performance advantages over conventional, fully-virtualized dynamic compilation.

Another important feature of ProtOS's dynamic compilation mechanism over conventional dynamic compilation is that a conventional dynamic compiler operating in the same process as the program is often forced either to violate the `W` $\oplus$ `X` paradigm (described in detail in Section 2.4) or sacrifice performance. Consider a code page housing newly minted code from the dynamic compiler that is also in the near-term execution path of the program. The compiler must either sacrifice performance by toggling `W` and `X` permissions on the page between code writes and execution, inducing TLB flushes, or sacrifice security by leaving the page `WX` while code is written to the page. Decoupled dynamic compilation in ProtOS, on the other hand, runs the program and compiler in separate processes uses separate permissions for code pages in each process – `RX` for the program, `RW` for the dynamic compiler – obviating this tradeoff.

6.2.2.1 Edge Virtualization Table

The targets of compiler-virtualized calls are organized into a structure in the binary called an edge virtualization table (EVT) to provide a convenient mechanism for new code to be stitched into the running program. This structure somewhat resembles the global offset table (GOT) used in stock dynamically-linked ELF programs to house the locations of shared library procedures. Unlike the GOT, however, the EVT provides the location of *every* function in the host program. For the sake of efficiency, it must be easily accessible to the host program, and thus by necessity it provides a neatly organized list of function addresses that, if compromised, could be

used by attackers to gain a tremendous amount of insight into the current memory layout of the program's code.

Our approach to mitigating this possibility is to continuously re-randomize the locations of each individual EVT entry as the application runs. Randomizing the locations of all EVT entries makes the location of any particular EVT entry extremely difficult to guess, and if one such location is divulged through an information leak it does not improve the attacker's prospects for finding any of the other entries.

Another difference between the EVT and the GOT is that the GOT is typically writable in program memory (the dynamic loader, running in userspace, modifies it), while the EVT resides in the code cache, which is `RX` in the program. As a result, the EVT is not easily amenable to control flow hijacking attempts based on overwriting its entries from the program.

6.2.2.2 Program Metadata

Another part of the compilation process in ProtOS is to enhance the binary, placing a minimal metadata section into the program binary to facilitate the dynamic compiler. The metadata section contains the compiler's intermediate representation (IR) of the program along with a map of the locations of code and data structures such as functions, call sites and variables. These are leveraged when ProtOS initializes the dynamic compiler process to allow it to find and modify these structures as it recompiles the running program. The metadata is not mapped into program memory. Furthermore, prior to allowing the program to begin the dynamic compiler performs a priming recompilation once to produce a new, randomized code layout, thus the code-related metadata is useful before the application executes but becomes stale once the application has begun execution.

6.2.2.3 Garbage Collection

ProtOS destroys earlier versions of code that reside code cache so that they cannot be leveraged by attackers. There are lazy and strict models of garbage collection implemented in ProtOS. Our lazy garbage collector destroys code and EVT entries that are no longer reachable by the application. Thus, lazy garbage collection can be done with near-zero overhead because the program never has to be stopped while the garbage collector runs. The down side of this approach is that it does not guarantee strict timing bounds as to when that code will be reclaimed.

Strict garbage collection immediately collects all but the most recent version of all code and EVT entries, thus this approach bounds the time that any particular byte remains in the same place in executable memory. The design and implementation of strict garbage collection is less trivial than lazy, as it must atomically update program state when more than one version of code is reachable simultaneously. Consider a partially executed function `foo` (i.e., it is stacked and control will return to it) that needs to be garbage collected, and another copy `foo'` that will replace `foo`. In strict garbage collection, when `foo` is reclaimed the dynamic compiler must modify the architectural state (memory and registers) referring to `foo` to reflect this change. The state referencing `foo` includes function pointers, jump tables, global offset table (GOT) entries, and goto labels, which must be updated to reflect the switch from `foo` to `foo'` when `foo` is garbage collected. The program is stopped during this portion of garbage collection, as the switch from `foo` to `foo'` must be atomic with respect to the program's architectural state to guarantee correct execution.

Particularly challenging in the design and implementation of strict collection is the handling of function pointers in programs. To address this challenge, the dynamic compiler tracks the locations of function pointers in memory using an approach similar to the one outlined by Bigelow et al. [24] and updates them as needed during garbage collection. Our current implementation of strict garbage collection in ProtOS sup-

ports the typical uses of function pointers, handling the spectrum of applications used in our evaluation. For more atypical uses involving custom-encoded function pointers that are difficult to track dynamically, ProtOS would have to revert to more costly mechanisms such as runtime address translation as described in prior work [30, 110].

All of the experiments and discussion in the remainder of this chapter assume the strict model of garbage collection because the focus of this work is on providing the most secure execution environment to undermine code reuse attacks.

6.2.3 Program Loading

ProtOS uses a custom loader, outlined in Figure 6.2, that extends the conventional ELF loader in several ways, and is designed with interoperability between conventional ELF binaries and ProtOS-enabled binaries in mind, seamlessly allowing the two to coexist in the same ecosystem. The specific steps taken at load-time are as follows:

1. **Read Binary File** — the loader first reads the program's binary file. If the file contains the proper metadata, the loader treats the file as ProtOS-enabled. Otherwise, the file is treated as a normal ELF binary by passing it along to be loaded by the conventional ELF loader.

2. **Setup Program and Compiler** — the loader next initializes the program process, setting up the process as usual, except that the program's original text segment is replaced by a larger code cache segment for dynamic compilation. This code cache is also mapped into the dynamic compiler process.

3. **Configure Permissions** — the loader then configures permissions for the program and its compilation process, setting the shared memory region to `RX` on the program side and `RW` on the compiler side. The compiler process is configured to allow it to `ptrace` only the specific application process paired with

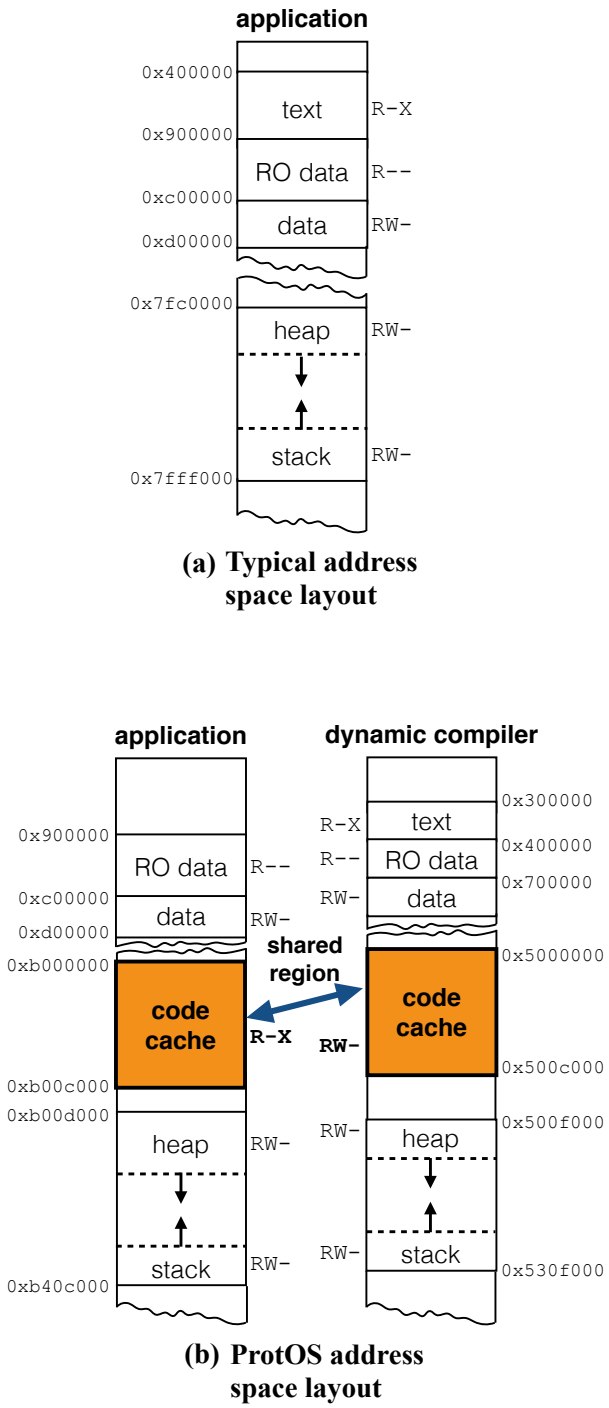


Figure 6.4: Sample address space layout of ProtOS application

it, allow it to pause, inspect and resume the program to perform management tasks such as garbage collection and failure notification.

4. **Priming Recompilation** — next, the compiler process launches as the program remains inert. The compiler performs a priming recompilation, randomizing the locations of all functions and EVT entries in the program. This step is equivalent to enacting a strong form of ASLR on the program's code before it begins execution. Note also that, like stock Linux, we perform segment-wise ASLR on the stack and heap segments for every program.

5. **Begin Program Execution** — when the priming recompilation has finished, the dynamic compiler allows the program to proceed with execution.

A typical address space layout resulting from the ProtOS loader is depicted in Figure 6.4. In 6.4(a), we illustrate the address space layout of a process resulting from the stock ELF loader and in 6.4(b) we illustrate the layouts of the program and compiler processes after being loaded with our custom loader. The key differences are (1) that a dedicated dynamic compilation process has been spawned to run alongside the program and (2) that the program contains none of its original code, instead having a code cache – a region of memory shared with the compiler process from which execution will occur – initialized with a randomized code layout.

6.2.4 Dynamically-linked Libraries

Dynamically-linked libraries can cause new code to be introduced into the application's address space at any point during execution. ProtOS is designed to gracefully handle ProtOS-enabled libraries in addition to legacy libraries.

The dynamic compilation process stubs out all procedure linkage table (PLT) entries in the program to intercept the first call to a shared library. At dynamic load time, ProtOS allows the application to load the shared library using the conventional

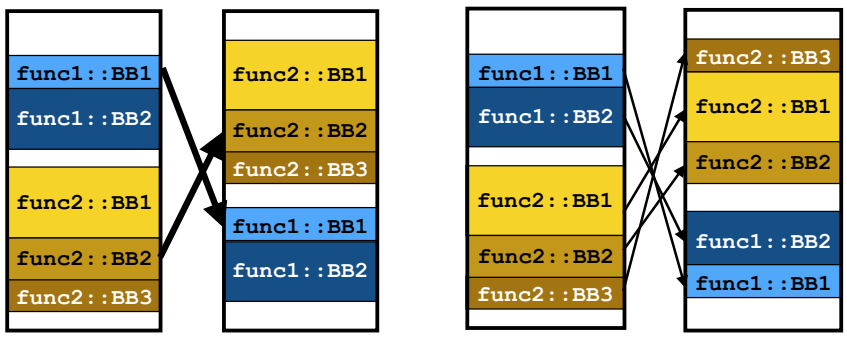
dynamic loader, then before allowing the program to continue execution locates the library file and checks whether it is a legacy library or a ProtOS-enabled library by checking for the existence of ProtOS metadata in the library. If the library is legacy, the program is allowed to continue execution without the benefit of the library being managed by the dynamic compiler. If the library is a ProtOS-enabled library, the dynamic compiler performs a priming recompilation and emits randomized library code into the program's code cache and destroys the original library code in memory. The randomized library code is subsequently managed by the dynamic compiler as though it were code from the program executable.

Conventional dynamically-linked library code pages can be physically shared among all the running processes that have loaded the library. A consequence of our approach to handling dynamically-linked libraries is that a unique copy of code from ProtOS-enabled shared libraries is present in the address space each ProtOS-managed program. As with conventional dynamically-linked libraries, data pages remain physically shared.

6.3 Continuous Code Re-randomization

Building on the online code transformation capability of ProtOS, we implement a novel, practical code re-randomization service that continuously re-positions and reorganizes program code, thwarting code reuse attacks by leaving attackers unable to make assumptions about the locations of bytes in memory.

Two forms of re-randomization are supported – a *medium-grain* approach that randomizes the location of each function, and a *fine-grain* approach that also randomizes the order of blocks within functions. Medium-grain re-randomization makes code reuse attacks difficult, placing a burden on the attacker to (1) discover the location of the function, (2) construct an attack and payload, and (3) execute their



(a) Medium-grain continuously re-randomizes function placement

(b) Fine-grain also re-randomizes block location within function

Figure 6.5: Different mixes of medium- and fine-grain re-randomization offer different resource/security tradeoffs

attack, all within a very short time window. Fine-grain re-randomization places an additional burden on the attacker because it also means they are unable to reliably assume that the function's structure remains fixed. Thus, even the discovery of the function's location may yield limited benefit to the attacker because they must also discover the function's current structure.

Two parameters define an overhead vs. security tradeoff in the re-randomization service. First is the length of time between rounds of program-wide re-randomization – shorter time between rounds provides increased protection against attacks but has higher resource overhead. Second is how often to use the fine-grain technique – fine-grain is more compute intensive but offers additional security over medium-grain. A diagram illustrating how the medium- and fine-grain approaches apply to a single function is presented in Figure 6.5.

The re-randomization service operates on the program by iteratively stepping through its functions, incrementally generating a re-randomized version of the program in the code cache. When the new program version is created, the compiler pauses the program, garbage collects the old version, finally resuming execution in the new version. These operations execute in a loop, continuously re-randomizing program code throughout execution.

6.3.1 Medium-grain Re-randomization

The steps taken to enact medium-grain re-randomization are depicted in Figure 6.6. 6.6(a) shows the call graph of a sample program with three functions. 6.6(b) shows an initial randomized memory layout of the program. The specific steps taken during a program-wide round of re-randomization are as follows:

1. **Randomize EVT** — first, the dynamic compiler chooses randomized locations for a new set of EVT entries, placing each new entry alone in a random location in the code cache. The coming steps will generate code that contains references to this new set of EVT entries, which are initialized to the addresses of the currently-executing versions of all functions and will be updated to point to new versions of each function as those new versions are generated.

2. **Re-position Functions** — the dynamic compiler chooses a function and selects a location for it randomly from among the free portions of the code cache that meet the size and alignment constraints of the compiler. The compiler then emits a new version of the function and places it into the code cache. Finally, the EVT entries for that function are updated to reflect the location of the new version of the function. Figure 6.6(c) shows the sample program after one function has been re-randomized. This function-level re-randomization proceeds until new versions of all program code reside in the code cache. This state is depicted for the sample program in Figure 6.6(d).

3. **Pause Program Execution** — it is important to note that the program runs continuously throughout the previous steps. However, at this point the dynamic compiler pauses program execution to perform garbage collection.

4. **Garbage Collection** — garbage collection is performed, clearing away the old version of code and updating architectural state to ensure that no remnants of

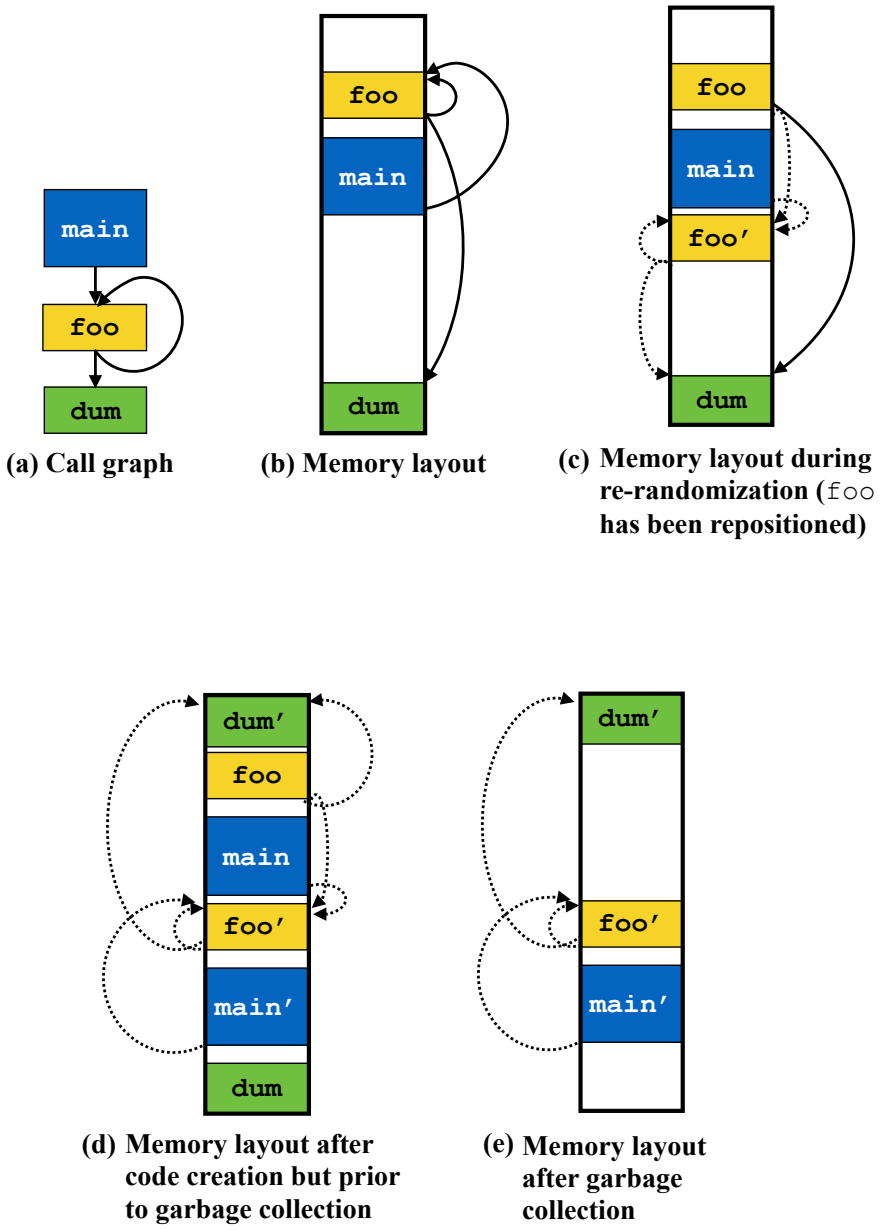


Figure 6.6: Steps taken to enact a round of re-randomization; after one round of re-randomization, all functions in the program has been re-randomized in position (medium-grain) and layout (fine-grain)

the old versions remain in the program, such as a return address to a call site in one of the old functions (see Section 6.2.2.3 for a detailed discussion of garbage collection). Figure 6.6(e) reflects the result of garbage collection in the sample program.

5. **Resume Program Execution** — the program resumes, now with fully re-randomized code and EVT entries.

6.3.2 Fine-grain Re-randomization

The steps taken to enact fine-grain re-randomization are similar to medium-grain re-randomization, however, instead of generating an identically-structured copy of each function as discussed above in Step 2, the dynamic compiler generates a new version with a re-randomized basic block order in addition to a re-randomized function placement. ProtOS allows medium- and fine-grain re-randomization to be mixed in arbitrary proportions by allowing the code re-randomization service to be configured with a parameter $p \in [\ 0, 1\ ]$. For every re-randomization, a fine-grain approach is used with probability p and medium-grain is used with probability $1 - p$.

Our implementation of fine-grain block-level reordering uses a compiler pass that randomizes block order. Because this step invokes numerous passes in the compiler to perform the block re-ordering and generate optimized code, the production of new code for fine-grain re-randomization uses significantly more CPU resources than medium-grain re-randomization. We quantify the impact of this overhead in Section 6.4.5.

6.3.3 Bytes, Bytes, Everywhere[1]

When initialized, the code cache is filled with trap instructions, `0xCC` in our x86_64 implementation. Moreover, upon freeing a region of memory previously held by an EVT entry or code, the dynamic compiler fills the region with trap instructions. Thus, all unused regions of the code cache are filled with trap instructions. This not only causes the unused areas of the code cache to be useless to attackers, but also allows ProtOS to catch would-be attackers in the act. That is, the execution of one these traps may result from a failed code reuse attack stemming from an attacker's outdated conception of what resides in program memory.

However, execution of a trap may also be the result of a buggy program. When the execution of one of these trap instructions is detected, execution of the program halts, which is detected by the dynamic compiler (the program PC is in an unused region of the code cache). The dynamic compiler then alerts the operating system to the failure of the program and the possibility of an attack, thus allowing further measures to be taken, for example, by dumping program state to disk, alerting an administrator, or writing a log message.

6.4 Evaluation

We now evaluate ProtOS, with particular focus on the resource overhead of the code re-randomization service and on using gadget detection software [145] to demonstrate the transient nature of gadgets in memory resulting from re-randomization.

6.4.1 Methodology

ProtOS is built into Linux kernel 3.13.0. The underlying hardware used in our experiments is a 16-core Intel Xeon E5-2630v3 (Haswell) with 2-way SMT, 64GB

[1]Here we refer to the line from Samuel Taylor Coleridge's *Rime of the Ancient Mariner* – "water, water, everywhere, nor any a drop to drink"

RAM and a 2.40GHz clock rate, which uses the x86_64 ISA.

We implement the ProtOS compilation infrastructure on top of LLVM version 3.3, a widely-used production strength open source compiler [101]. All programs are compiled to support ProtOS and are statically linked against a ProtOS-compiled version of the `musl` implementation of `libc` [1]. We choose `musl` because it builds easily with LLVM, whereas GNU `libc` contains a significant amount of code supported only by the GNU compiler. Our return oriented programming (ROP) gadget detection experiments use ROPGadget version 5.3 [145] with a maximum gadget size of 20 bytes.

The evaluation uses programs representing a spectrum of domains and computational characteristics that stress the performance of ProtOS and its code rerandomization service, including programs that are compute- and memory-intensive, have large code bases, and large instruction working sets. In particular, we use the following 22 programs — `bwaves`, `bzip2`, `lbm`, `mcf` and `namd` from SPEC CPU2006 [80]; `gmm` and `stemmer` from SiriusSuite [79]; `barnes`, `fft`, `lu_cb`, `lu_ncb`, `ocean_cp`, `ocean_ncp`, `water_nsquared` and `water_spatial` from Splash2x [22]; `blockie`, `bst`, `er-naive`, `naive`, `sledge` from SmashBench [115]; `blackscholes` and `freqmine` from PARSEC [23].

6.4.2 ProtOS System Overhead

We begin by evaluating what it costs to have the code transformation capability in ProtOS. We compare the runtime of ProtOS programs to their non-ProtOS counterparts compiled to ELF binaries and running on a stock Linux. The ProtOS-based experiments involve no dynamic code transformation; the program is launched alongside an inert dynamic compilation thread that immediately goes idle and makes no modifications to the program. Thus, the main source of overhead is to execute indirect calls through the EVT in the ProtOS-compiled programs instead of direct calls in

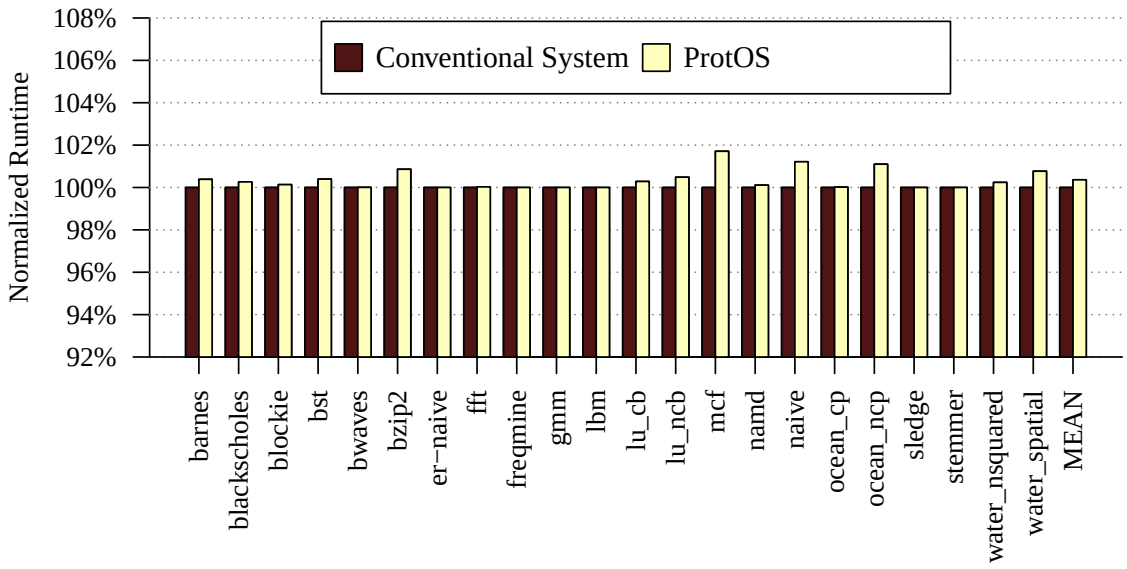


Figure 6.7: ProtOS programs show negligible slowdowns compared to programs on a stock Linux system

the conventional Linux/ELF programs. As the figure shows, the overhead of running ProtOS-compiled programs is negligible, having a maximum performance overhead of 1.7% and an average performance overhead of 0.3%. This shows that there is very little performance overhead just to use the basic ProtOS system architecture with a code transformation capability.

6.4.3 Code Re-randomization Performance

Offering continuous runtime support in thwarting code reuse attacks without introducing significant runtime overheads is a challenging problem. The most closely related techniques that attempt to provide such protection are in the area of control flow integrity, where the most comprehensive techniques have overheads of 2-5× [39, 52]. These levels of overhead are too high for users to bear and serve as a barrier to their adoption. A primary feature of our code re-randomization approach is that it transparently provides a more secure execution environment *while introducing*

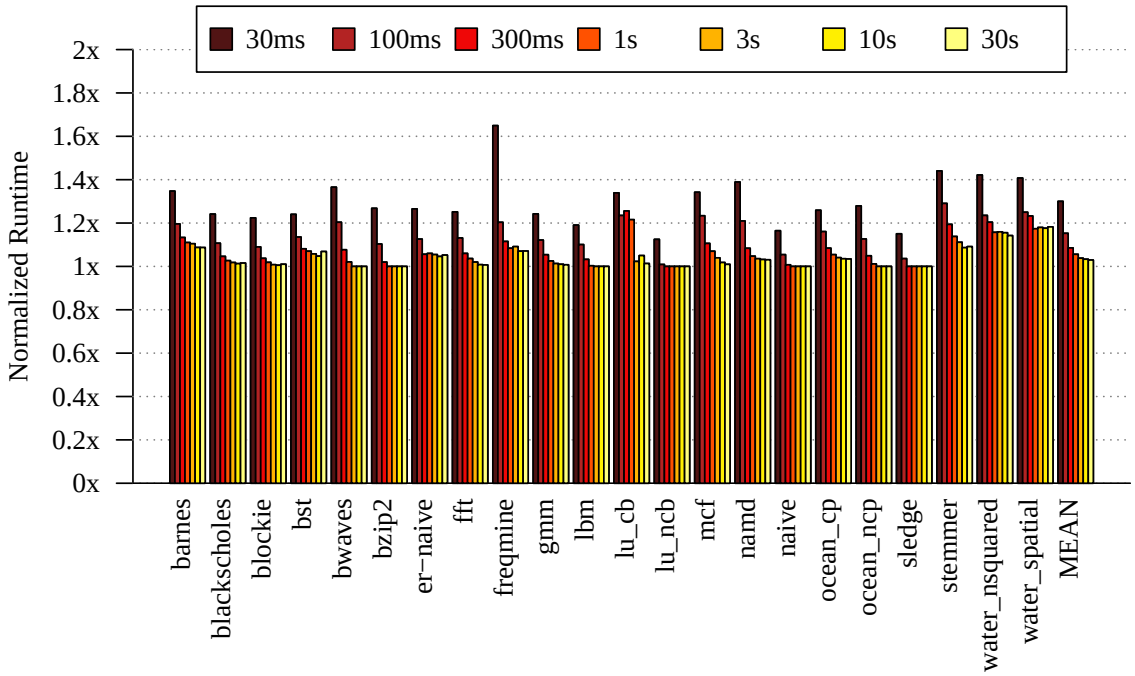


Figure 6.8: Performance overhead of the medium-grain re-randomization service in ProtOS; 300ms offers an attractive design point, in that it re-randomizes fast enough to thwart state-of-the-art code reuse attacks [156,162] with only 9% runtime overhead

minimal overhead.

Re-randomization Frequency. We conduct experiments to examine the performance impact of the re-randomization service as a function of the re-randomization frequency. Figure 6.8 presents the results, showing the runtime of each program at a range of different re-randomization frequencies ranging from 30ms to 30s when exclusively using medium-grain re-randomization. The runtime results in this experiment are normalized to the execution time of the stock ELF/Linux program, and are inclusive of all sources of overhead.

The overhead of code re-randomization increases as the re-randomization becomes more frequent. We examine the sources of overhead in more detail shortly. The performance overhead is below 5% at all frequencies larger than 1 second. Moreover, the performance overhead of re-randomizing once every 300ms averages 9%. As we discuss shortly, a frequency of 300ms thwarts current state-of-the-art ROP attacks

Workload A	`bst`, `gmm`, `lu_ncb`, `water_spatial`
Workload B	`bzip2`, `er-naive`, `stemmer`, `lu_cb`
Workload C	`mcf`, `naive`, `blackscholes`, `barnes`
Workload D	`blockie`, `sledge`, `fft`, `water_nsquared`

Table 6.1: Multiprogram workloads

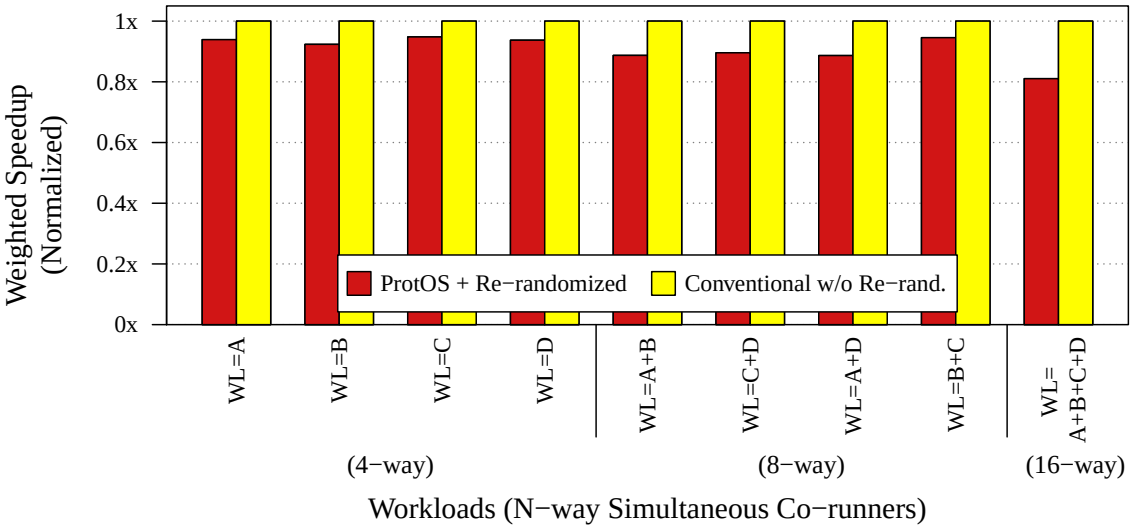


Figure 6.9: Throughput of multiprogram workloads; throughput suffers small degradations even when re-randomizing all 16 co-runners in a fully subscribed system every 300ms

by reducing them to blind guessing about where gadgets reside in memory. Finally, we point out that ProtOS has a significant resilience to advancements in the speed of code reuse attacks, in that it can re-randomize code once every 30ms to (over 10× faster than necessary to thwart modern attacks) while incurring an overhead of only 31%.

Multiprogram Workloads. To investigate the impact of using re-randomization every 300ms for all applications on a highly subscribed system, we conduct experiments on re-randomizing multiprogram workloads. We use the workloads described in Table 6.1, which are run either in isolation (1 workload implies 4 co-running applications), in pairs (8 co-running applications) or all at once (16 co-running applications).

Our experimental setup is to launch a set of co-running applications, running every application in a loop for 30 minutes while collecting the average execution time of each application. From these execution times, we compute the weighted speedup relative to the solo execution time of the applications on the stock ELF/Linux system. We then compare the weighted speedup of ProtOS to the stock Linux system.

The results are shown in Figure 6.9. The left side of the figure shows the throughput achieved for 4-way co-runner experiments, the middle shows 8-way, and the right shows 16-way. The degradation due to co-running applications on the conventional system increases as the number of co-runners increases due to the additional dynamic compilation processes and the associated interference they cause for shared resources such as caches and memory bandwidth. For 4-way co-running the throughput degradation is as low as 5%. For the 16-way co-running case, recall that the hardware platform has 16 CPUs, and thus for 16 co-running applications each CPU with 2-way SMT is subscribed running an application process and a dynamic compilation process. Nevertheless, the 16-way co-running case shows an overhead of only 19%, demonstrating that our technique remains practical even when deployed on multiprogram workloads in heavily subscribed systems.

6.4.4 Sources of Application Overhead

We quantify the sources of overhead in re-randomized applications by first measuring the direct overhead imposed by the dynamic compiler in pausing the application to perform garbage collection. Figure 6.10 presents the average overhead across applications along with the average of overhead from all other sources. As the figure shows, garbage collection overhead is negligible at longer frequencies, becoming more significant at shorter frequencies and rising to 9% when re-randomization occurs every 30ms.

Dynamically Generated Code Quality. Visible in Figure 6.10 is an overhead of

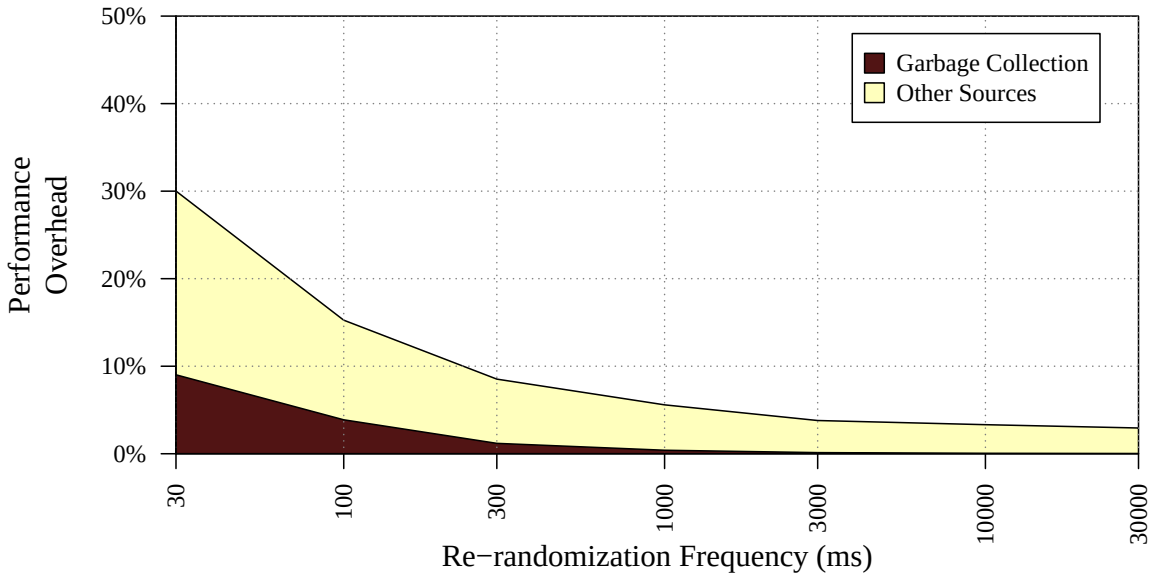


Figure 6.10: Overhead of garbage collection

3% when re-randomizing code every 30 seconds. Referring back to Figure 6.8, we observe that `barnes`, `stemmer`, `water_nsquared` and `water_spatial` exhibit slowdowns of 9-20% when re-randomization occurs every 30 seconds.

Figure 6.11 shows the results of an experiment where we re-randomize applications only once at the beginning of execution and collect the dynamic instruction counts during the run using hardware performance monitors. The figure plots a point for each application whose position along the x and y axes are its execution time overhead and dynamic instruction count overheads, respectively. This figure shows that there is an very strong correlation (the co-efficient of correlation is $\rho = 0.89$) between these two factors. Further investigation reveals that in certain instances, our dynamic compilation infrastructure generates code that is bloated relative to the code produced by the static compiler in some key hot code locations. Thus, these programs end up executing more instructions than their statically-compiled counterparts. While our dynamic compilation infrastructure based on LLVM is very mature and generates highly optimized code, this highlights the importance of continued work in developing

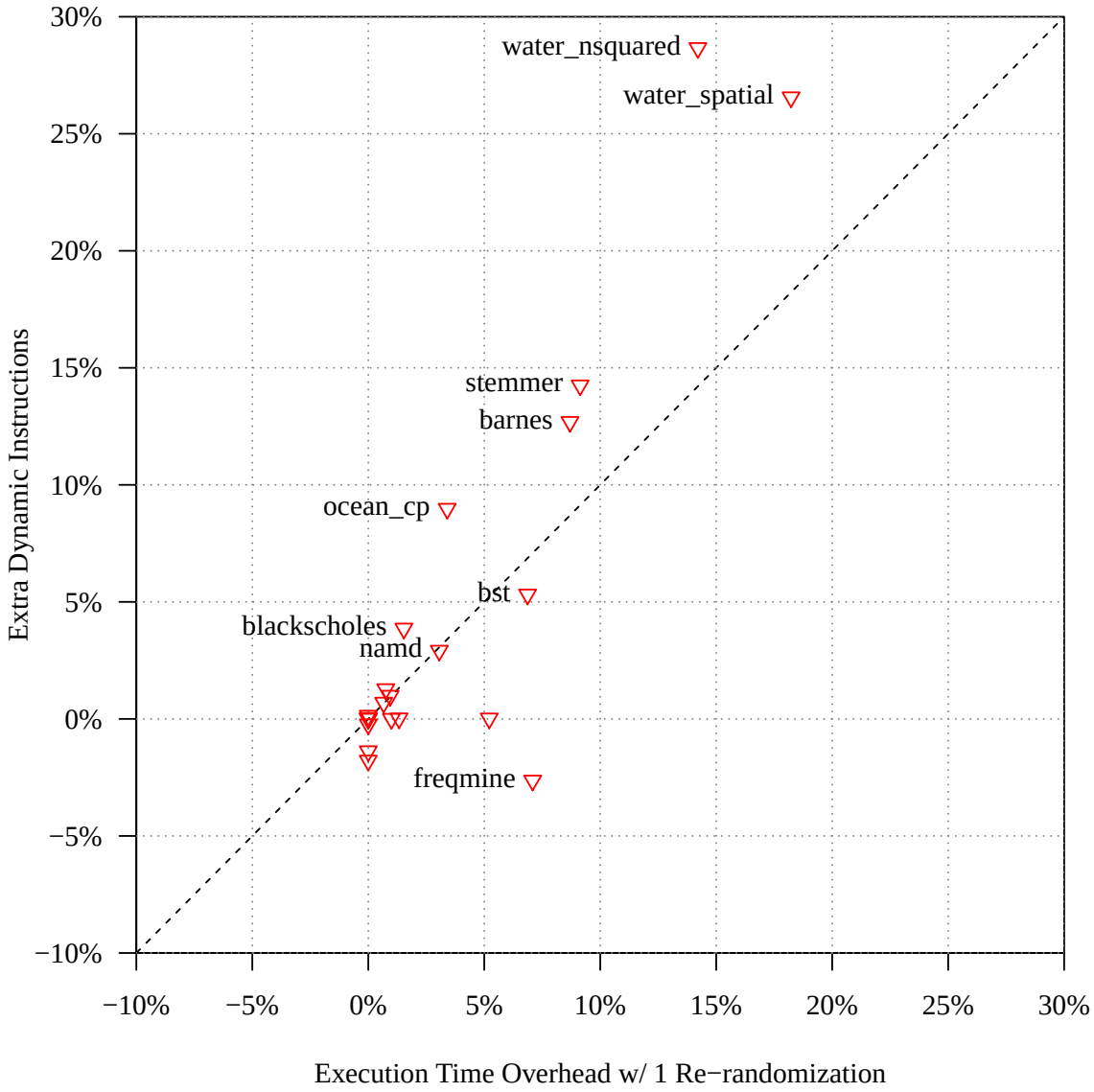


Figure 6.11: Dynamically-generated code instruction count vs. application runtime overhead; correlation between the two is p=0.89

the LLVM dynamic compiler.

Side Effects. The remaining performance overheads in re-randomized applications result from architectural and microarchitectural side effects. These side effects include (1) the impact on instruction cache and TLB from executing re-randomized versions of code that reside in different locations at different points in time and (2) contention for resources like last level cache and memory that the dynamic compiler process shares with the application.

To assess the impact of these side effects, we profile the dynamic activity of a number of hardware performance monitors during application runs and find that TLB misses show significant differences between 300ms re-randomized applications and stock applications. Figure 6.12 presents dynamic traces of four hardware performance monitors measuring important aspects of the memory subsystem collected during native runs of `mcf` on stock Linux, as well as during re-randomized runs on ProtOS. `mcf` is a memory-intensive application, and thus we expect it to be sensitive to data cache interference. However, data cache effects are barely noticeable while the TLB misses per cycle in Figure 6.12(d) are significantly higher during re-randomized runs. Nevertheless, the performance impact of these side effects amounts to just an 8% overhead on `mcf` and a 7% overhead on average across all applications.

6.4.5 Medium vs. Fine-grain Re-randomization

We now compare the resource overheads of medium-grain and fine-grain re-randomization. Fine-grain re-randomization provides additional security benefits over medium-grain re-randomization. It places an additional burden on the attacker because it means they cannot make assumptions about the structure of functions, and thus the discovery of part of the function may have limited benefit in locating additional ROP gadgets because the function's structure is not known. However, fine-grain re-randomization requires more compute resources to generate code because it invokes a series of com-

piler passes that randomizes the block order and optimizes the resulting code before emitting machine code.

CPU Utilization. We examine the CPU resource requirements of fine-grain re-randomization through a series of experiments shown in Figure 6.13 that use 8 mixes of fine-grain and medium-grain re-randomization. A particular mix of fine- and medium-grain re-randomization is characterized by p, the probability that fine-grain re-randomization is applied (medium-grain is used with probability $1 - p$). For each mix we re-randomize applications as frequently as possible, a frequency limited by how quickly the CPU can perform re-randomization rounds. As the figure shows, the more fine-grain re-randomization is used, the less frequently re-randomization can be applied to the entire program. Our experiments show that 0% fine-grain (100% medium-grain) re-randomization can be applied every 0.03s, while mixes that use 1%, 10% and 100% fine-grain can be applied every 0.15s, 1.45s, and 14s, respectively. A useful mix of the two is to use fine-grain re-randomization sparingly (e.g., with probability 1-2%), imposing minimal extra resource overhead and allowing re-randomization to occur with high frequency, while also burdening attackers by breaking the assumption that function layouts remain fixed throughout an applications lifetime.

Dynamic Behavior. Figure 6.14 illustrates the activity of the re-randomizer by showing the locations of gadgets within `er-naive` throughout its run. The experiment used to generate this illustration dumps the contents of the code cache to disk after each round of re-randomization, which is then scanned for ROP gadgets using the ROPGadget tool [145]. This example is scoped down to include only four of the functions in `er-naive` (there are well over 2000 functions in the entire program, including those from `libc`). The figure shows the locations of gadgets in the address space of `er-naive`. Each function contains a number of ROP gadgets, so to simplify

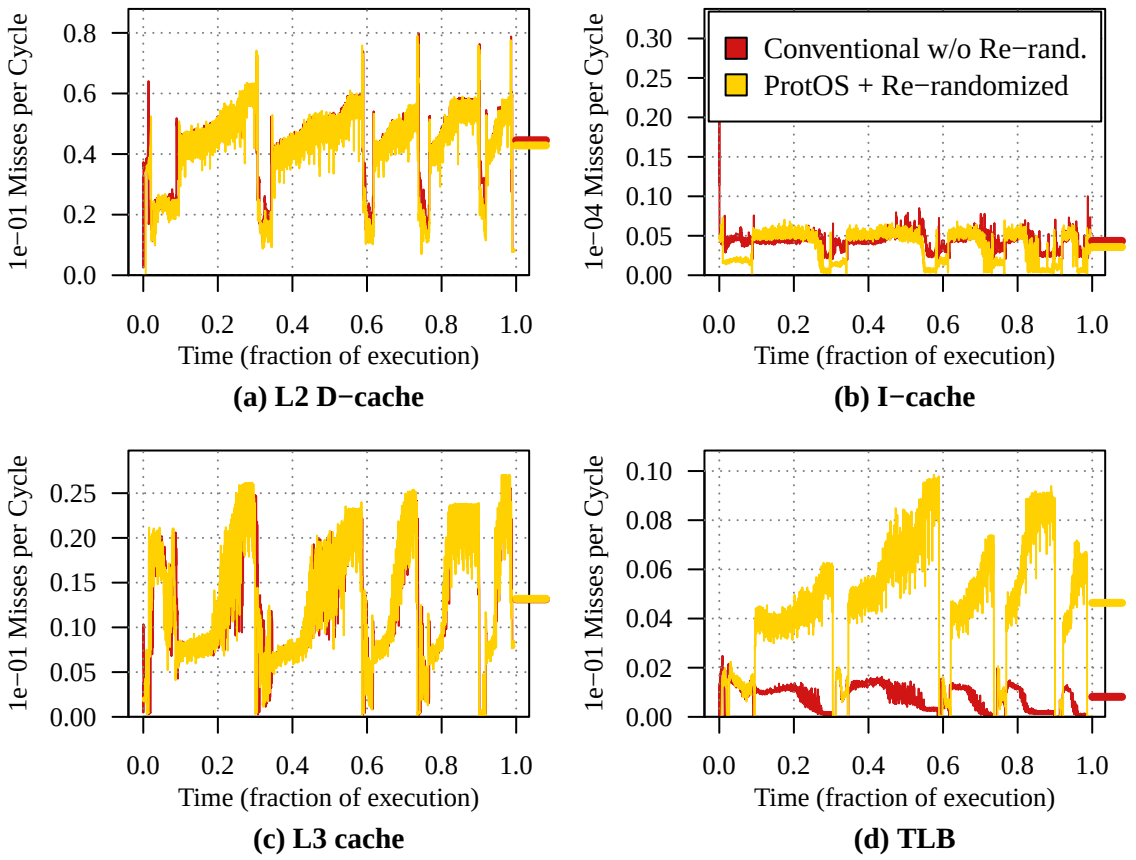


Figure 6.12: Dynamic memory behavior of `mcf` with and without re-randomization; the key factor impacting performance when re-randomizing code is frequent TLB invalidations

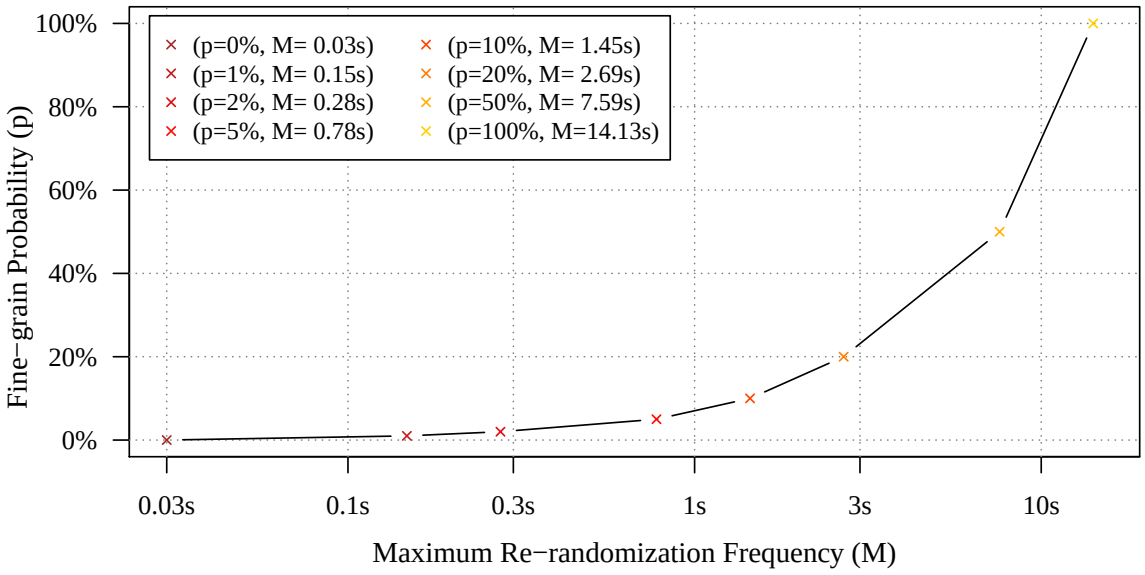


Figure 6.13: Tradeoff between frequency and granularity of re-randomization

the presentation we plot a single point rather than a cluster of points to represent the gadgets in each function.

We annotate the figure to show the positions of a subset of gadgets within `main` during several stages of the run. Between the first and second annotations, medium-grain re-randomization is performed, changing the location of the function and gadgets but preserving the structure of the function and relative positions of many gadgets. Between the second and third annotations fine-grain re-randomization changes the function layout, thereby also changing the relative positions of many gadgets within the function.

6.4.6 Security Implications

Re-randomization makes the locations of ROP gadgets unreliable to attackers. This unreliability is central in undermining the ability of ROP attacks to rely on the locations of gadgets in the construction of a code reuse attack. To measure this unpredictability, we again dump the contents of the program's executable memory

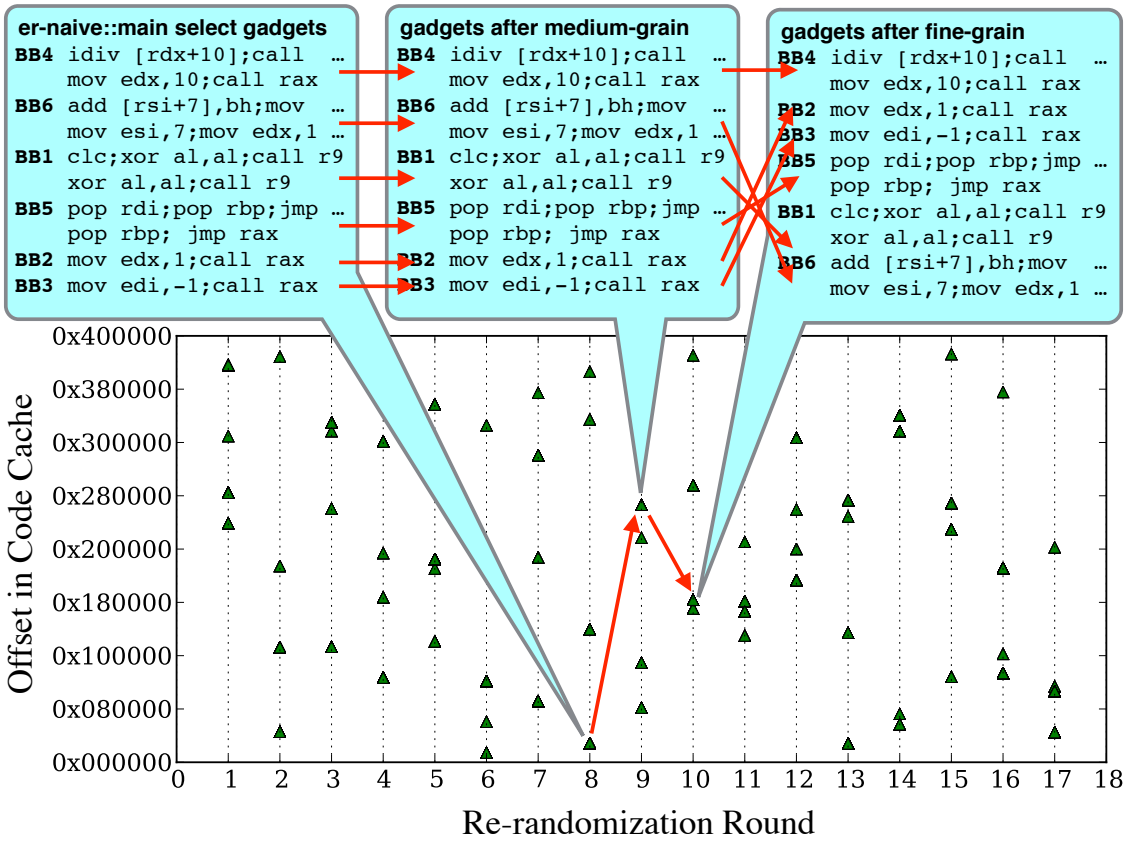


Figure 6.14: Gadgets detected within 4 functions of `er-naive`; memory is dumped after each round of re-randomization and gadgets are detected offline using ROPGadget [145]

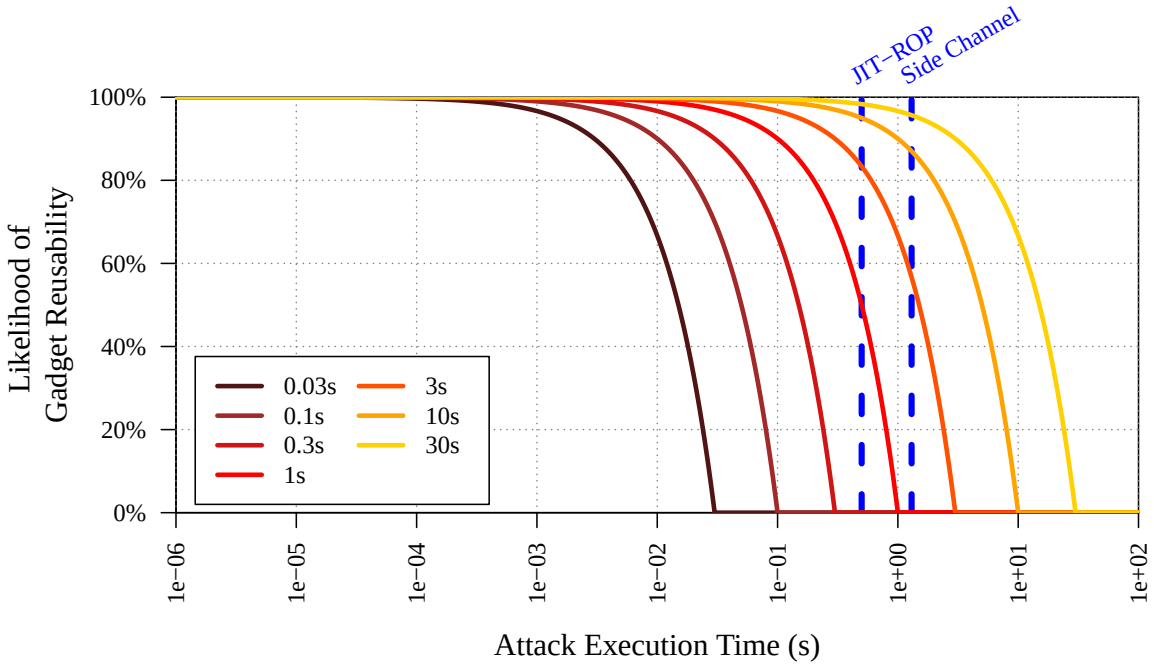


Figure 6.15: Likelihood of individual ROP gadgets remaining in place long enough to orchestrate an attack; at 300ms, re-randomization occurs rapidly enough to prevent even a single ROP gadget from remaining in place long enough to be usable in state-of-the-art ROP techniques

region and use ROPGadget to extract the available ROP gadgets from the executable memory region at different points in time and compute the average likelihood across applications and gadgets that any particular gadget remains in place after different amounts of elapsed time.

Our findings from these experiments are presented in Figure 6.15. The likelihood of finding a ROP gadget at its current position at any particular point in time in the future approaches zero as time elapses, regardless how (in)frequently the code is re-randomized. This matches the intuition of how the re-randomization works; when selecting new locations for re-randomized code, the locations are chosen at random from within the code cache. Thus, re-randomizing code at any frequency implies that *all ROP gadgets will eventually be moved or destroyed.*

Moreover, re-randomizing code every 300ms offers a very attractive design point. It introduces a modest amount of performance overhead (9% on average), and re-

randomizes code frequently enough to undermine state-of-the-art ROP attacks. Consider the just-in-time (JIT) code reuse attack, which requires at least 500ms to execute and end-to-end ROP attack by repeatedly exploiting memory disclosures to dynamically build a map of the executable code in the application [162]. When a frequency of 300ms is used, our code re-randomization implementation will have re-positioned every gadget while the attack is being orchestrated, making such an attack extremely unlikely to succeed. Similarly, recently published work has demonstrated that side channels may be used to infer the contents of executable memory when certain programming constructs are present in the program [156]. In the fastest form of their technique, leaking a single byte can take as little as 1.3 seconds, which our re-randomization service can undermine easily by re-randomizing code once per second.

6.5 Summary

This chapter motivates and describes ProtOS, a novel system architecture that hosts an online code transformation capability as a system service. Using this online code transformation service, we design and evaluate a mechanism for performing continuous code re-randomization to undermine code reuse attacks. Our experiments with ProtOS demonstrate the feasibility of this unique conception of code transformation as a system service. We show also our re-randomization technique can re-randomize code frequently enough to mitigate state-of-the-art code reuse attacks with modest performance overheads that average 9% across a broad range of applications.

CHAPTER VII

Conclusions and Future Directions

This dissertation motivates and proposes a new approach to enabling online code transformations, allow executing native programs to be transformed as they run. The approach works by allowing a dynamic compiler to run asynchronously and in parallel to the running program, thus keeping the runtime overhead of possessing an online code transformation capability to near-zero and allowing programs to execute with this capability at near-native speed. The capability to continuously perform online code transformations opens up a new design space for native programs, as such programs no longer need to have a fixed implementation throughout their execution.

I demonstrate these opportunities by describing three novel applications of online code transformations, resulting in best-in-class solutions to several challenging problems facing computer scientists today. First, I use online code transformations to improve the utilization of multicore datacenter servers, significantly reducing the number of servers needed to host latency-critical web services to mitigate the cost and environmental impacts of operating datacenters. This technique strategically injects software cache hint instructions into a running application to allow the applications housed on the server to cooperatively use shared server resources far more effectively than prior techniques that lack the online code transformation capability. Second, I build a technique to automatically configure and parameterize approximate com-

puting techniques for each program input. This technique results in the ability to configure approximate computing to achieve an average performance improvement of 10.2× while maintaining 90% result accuracy, which significantly improves over oracle versions of prior techniques. Third, I leverage online code transformation to thwart code reuse attacks, a class of attacks that are widely used today by malicious attackers to subvert and hijack the execution of software systems. This technique builds an online code transformation capability into an operating system (OS), thus allowing the OS to efficiently, transparently and continuously re-randomize code inside running applications, invalidating attackers' assumptions as to the location of code in memory that are needed to successfully execute a code reuse attack.

Beyond these opportunities, the introduction of a low-overhead online code transformation technique can have wide-reaching implications, impacting the future of hardware and software design.

7.1 Software Adaptation

The introduction of a low-overhead online code transformation technique that allows software to be dynamically manipulated can be used to employ many classes of optimizations whose efficacy depends on the application's runtime environment, broadly defined as the set of internal and external states in which the application runs (e.g., other running applications, performance/power/resilience constraints, or the application's input). Examples of this include cache tiling, thread-level parallelism, instruction scheduling, hardware and software prefetching [95], duplication, and the application of approximate computing techniques. Similarly, online code transformations could be used to enact short-lived profiling or instrumentation, allowing quick bursts of feedback about performance or other interesting characteristics of application behavior.

Beyond the security focus of Chapter VI, an enhanced system architecture with

an online code transformation capability opens up a new design space for other aspects of the OS. The OS hosts a number of services that can benefit from a dynamic compilation capability to improve performance. For example, a task scheduler could transform application code online to share resources more effectively [103, 169, 170] or to dynamically invoke core-specific optimizations in heterogeneous multi-core systems as it moves tasks among cores with different capabilities [45, 74]. Similarly, a memory manager could use page management policies in concert with online code transformation to improve the performance or predictability of the TLB [14, 93, 134].

The low overhead approach to online code transformations could also be used to apply additional security measures such as Control Flow Integrity protections [4] and software diversity [175] into running application code. Such measures could be used selectively during periods of elevated threat, or used *on-demand* in response to a security-related event (e.g., a process that may be malicious begins to run).

7.2 Hardware Design

By providing a mechanism to dynamically change the running code, online code transformations make possible a number of dynamic software compilation strategies that have the capacity to influence the way we think about designing hardware.

The ability of software to adapt to and take advantage of architectural knobs exposed by the hardware to fine-tune the behavior of the hardware means that such knobs should be more readily exposed by hardware designers. One example of the importance of such knobs is evidenced by the system described in Chapter IV that takes advantage of non-temporal prefetch instructions to optimize co-running datacenter applications.

Energy efficient execution on top of existing and emerging microarchitectural features can be leveraged more successfully using online code transformations. For example, instructions could be dynamically scheduled (reordered), achieving schedules

that bring the performance of superscalar in-order cores closer to that of out-of-order cores. Such an optimization could radically increase the performance of in-order cores and make them a more performance-competitive design.

Moreover, the ability to transform code to avoid unreliable or failing architectural units (e.g., a certain functional unit) may encourage the adoption of low-voltage or otherwise unreliable designs. Similarly, heterogeneity and accelerator designs may be facilitated by online code transformations, which can be used to generate highly optimized and specialized code that take advantage of the unique features offered by such designs.

The capability to easily transform code may also have implications for increasing the adoption of increasingly thread- and data-parallel hardware. For instance, the runtime compiler may be able to guarantee that certain dependence conditions hold in a particular execution environment (e.g., due to the peculiarities of the input) that do not hold generally, and may find parallelization opportunities not available to a static compiler. Alternatively, the runtime compiler could find a number of opportunities where such dependencies are *usually* true, resulting in an upsurge in parallelization opportunities that could effectively use architectural speculation support, leading to faster adoption of that support.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] musl libc. `http://www.musl-libc.org/`. [Online; accessed 10-November-2015].

[2] ARMv8 Instruction Set Overview. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.genc010197a/index.html`, 2011. Online; accessed 10-November-2015.

[3] Intel 64 and IA-32 Architectures Software Developers Manual. Volume 2: Instruction Set Reference A-Z. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, 2014. Online; accessed 10-November-2015.

[4] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Computer and Communications Security (CCS)*, 2005.

[5] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *European Conference on Computer Systems (EuroSys)*, 2013.

[6] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger. General-purpose code acceleration with limited-precision analog computation. In *International Symposium on Computer Architecture (ISCA)*, 2014.

[7] AMD. *AMD64 Architecture Programmer's Manual: System Programming*, volume 2.

[8] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. *Code Generation and Optimization (CGO)*, 2011.

[9] Apple. Siri. `https://www.apple.com/ios/siri/`, 2014. [Online; accessed 10-November-2015].

[10] ARM. *ARM Architecture Reference Manual*.

[11] W. Arthur, B. Mehne, R. Das, and T. Austin. Getting in control of your control flow with control-data isolation. In *Code Generation and Optimization (CGO)*, 2015.

[12] J. Auslander, M. Philipose, C. Chambers, S. J. Eggers, and B. N. Bershad. Fast, effective dynamic compilation. In *Programming Language Design and Implementation (PLDI)*, 1996.

[13] K. Bache and M. Lichman. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2014. [Online; accessed 10-November-2015].

[14] D. F. Bacon, J.-H. Chow, D.-c. R. Ju, K. Muthukumar, and V. Sarkar. A compiler framework for restructuring data declarations to enhance cache and tlb effectiveness. In *IBM Centre for Advanced Studies Conference (CASCON)*, 1994.

[15] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Programming Language Design and Implementation (PLDI)*, 2010.

[16] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Programming Language Design and Implementation (PLDI)*, 2000.

[17] B. Bao and C. Ding. Defensive loop tiling for shared cache. In *Code Generation and Optimization (CGO)*, 2013.

[18] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and A. Tumeo. *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, 2013.

[19] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: an introduction to the design of warehouse-scale machines, 2nd edition. *Synthesis Lectures on Computer Architecture*, 2013.

[20] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Programming Language Design and Implementation (PLDI)*, 2006.

[21] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium (SEC)*, 2005.

[22] C. Bienia, S. Kumar, and K. Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *International Symposium on Workload Characterization (IISWC)*, 2008.

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[24] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely rerandomization for mitigating memory disclosures. In *Computer and Communications Security (CCS)*, 2015.

[25] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In *Security and Privacy (SP)*, 2014.

[26] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Computer Security Applications Conference (ACSAC)*, 2011.

[27] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *Computer and Communications Security (CCS)*, 2011.

[28] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<T>: A first-order type for uncertain data. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[29] B. Breech, A. Danalis, S. Shindo, and L. Pollock. Online impact analysis via dynamic compilation technology. In *International Conference on Software Maintenance (ICSM)*, 2004.

[30] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization (CGO)*, 2003.

[31] S. Byna, J. Meng, A. Raghunathan, S. Chakradhar, and S. Cadambi. Best-effort semantic document search on GPUs. In *Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.

[32] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *Programming Language Design and Implementation (PLDI)*, 2012.

[33] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[34] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium (SEC)*, 2014.

[35] L. Ceze and J. Larus. Report on ISAT/DARPA workshop on accuracy trade-offs across the system stack for performance and energy (aka approximate computing). In *DARPA Innovative Space Based Radar Antenna Technology (ISAT) Workshop*, 2014.

[36] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *Foundations of Software Engineering (FSE)*, 2011.

[37] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Computer and Communications Security (CCS)*, 2010.

[38] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Electronic Voting Technology Workshop (EVT)*, 2009.

[39] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In *International Conference on Information Systems Security (ICISS)*. 2009.

[40] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies. Mojo: A dynamic optimization system. In *Feedback-Directed and Dynamic Optimization (FDDO)*, 2000.

[41] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *IEEE Symposium on Computers and Communications (ISCC)*, 2006.

[42] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Network and Distributed System Security (NDSS)*, 2014.

[43] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. Fx! 32: A profile-directed binary translator. *IEEE Micro*, 1998.

[44] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with revgen. In *Dependable Systems and Networks (DSN)*, 2011.

[45] N. Chitlur, G. Srinivasa, S. Hahn, P. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer. QuickIA: Exploring heterogeneous architectures on real prototypes. In *High Performance Computer Architecture (HPCA)*, 2012.

[46] J. Cohen. *Statistical power analysis for the behavioral sciences*. Lawrence Erlbaum, 1988.

[47] C. Consel and F. Noël. A general approach for run-time specialization and its application to c. In *Principles of Programming Languages (POPL)*, 1996.

[48] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[49] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Security and Privacy (SP)*, 2015.

[50] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. *Network and Distributed Systems Security (NDSS)*, 2015.

[51] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium (SEC)*, 2014.

[52] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Computer and Communications Security (CCS)*, 2011.

[53] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Code Generation and Optimization (CGO)*.

[54] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[55] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *Programming Language Design and Implementation (PLDI)*, 2015.

[56] T. Dullien, T. Kornau, and R.-P. Weinmann. A framework for automated architecture-independent gadget search. In *Workshop on Offensive Technologies (WOOT)*, 2010.

[57] O. J. Dunn. Multiple comparisons among means. *Journal of the American Statistical Association*, 1961.

[58] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *Internet Measurement Conference (IMC)*, 2014.

[59] K. Ebcioğlu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *International Symposium on Computer Architecture (ISCA)*, 1997.

[60] E. Eng and D. Caselden. Operation clandestine wolf – Adobe flash zero-day in APT3 phishing campaign. `https://www.fireeye.com/blog/threat-research/2015/06/operation-clandestine-wolf-adobe-flash-zero-day.html`, 2015. [Online; accessed 10-November-2015].

[61] D. R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Programming Language Design and Implementation (PLDI)*, 1996.

[62] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[63] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *International Symposium on Microarchitecture (MICRO)*, 2012.

[64] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. Evaluating the error resilience of parallel programs. In *Dependable Systems and Networks (DSN)*, 2014.

[65] P. Feiner, A. D. Brown, and A. Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[66] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[67] R. Fu, J. Lu, A. Zhai, and W.-C. Hsu. A study of the performance potential for dynamic instruction hints selection. In *Asia-Pacific Computer Systems Architecture Conference (ACSAC)*. 2006.

[68] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium (SEC)*, 2012.

[69] I. Goiri, R. Bianchini, S. NagaraKatte, and T. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[70] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium (SEC)*, 2014.

[71] Google. Glass. `https://www.google.com/glass`, 2014. [Online; accessed 10-November-2015].

[72] Google. Google Now. `http://www.google.com/landing/now/`, 2014. [Online; accessed 10-November-2015].

[73] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in dyc. In *Programming Language Design and Implementation (PLDI)*, 1999.

[74] P. Greenhalgh. big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 2011.

[75] B. Grigorian, N. Farahpour, and G. Reinman. Brainiac: Bringing reliable accuracy into neurally-implemented approximate computing. In *High Performance Computer Architecture (HPCA)*, 2015.

[76] B. Grigorian and G. Reinman. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2014.

[77] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino. Marlin: Making it harder to fish for gadgets. In *Computer and Communications Security (CCS)*, 2012.

[78] J. R. Hauser. Handling floating-point exceptions in numeric programs. *Transactions on Programming Languages and Systems (TOPLAS)*, 1996.

[79] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[80] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006.

[81] F. B. Hildebrand. *Introduction to numerical analysis*. Courier Corporation, 1987.

[82] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *Security and Privacy (SP)*, 2012.

[83] H. Hoffmann, J. Eastep, M. Santambrogio, J. Miller, and A. Agarwal. Application heartbeats for software performance and health. *MIT CSAIL Technical Report*, 2009.

[84] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. *MIT CSAIL Technical Report*, 2009.

[85] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[86] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 1979.

[87] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Programming Language Design and Implementation (PLDI)*, 1992.

[88] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz. Librando: transparent code randomization for just-in-time compilers. In *Computer and Communications Security (CCS)*, 2013.

[89] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX Security Symposium (SEC)*, 2009.

[90] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*.

[91] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *High Performance Embedded Architectures and Compilers (HiPEAC)*. 2010.

[92] A. Jimborean, P. Clauss, J.-F. Dollinger, V. Loechner, and J. M. M. Caamano. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 2014.

[93] M. Kandemir, I. Kadayif, and G. Chen. Compiler-directed code restructuring for reducing data tlb energy. In *International conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2004.

[94] H. Kasture and D. Sanchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[95] M. Khan, M. A. Laurenzano, J. Mars, E. Hagersten, and D. Black-Schaffer. Arep: Adaptive resource efficient prefetching for maximizing multicore performance. In *Parallel Architecture and Compilation Techniques (PACT)*, 2015.

[96] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. Rumba: an online quality management system for approximate computing. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[97] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference (ACSAC)*, 2006.

[98] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 2008.

[99] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[100] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, 2004.

[101] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization (CGO)*, 2004.

[102] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems and Software (ISPASS)*, 2010.

[103] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[104] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *Design, Automation and Test in Europe (DATE)*, 2010.

[105] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In *European Conference on Computer Systems (EuroSys)*, 2010.

[106] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. SWAT: an error resilient system. In *Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2008.

[107] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture (HPCA)*, 2008.

[108] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving DRAM refresh-power through data partitioning. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[109] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *International Symposium on Microarchitecture (MICRO)*, 2003.

[110] 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 *Programming Language Design and Implementation (PLDI)*, 2005.

[111] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Programming Language Design and Implementation (PLDI)*, 1999.

[112] J. Mars and M. L. Soffa. Mats: Multicore adaptive trace selection. In *Workshop on Software Tools for MultiCore Systems (STMCS)*, 2008.

[113] J. Mars and M. L. Soffa. Synthesizing contention. In *Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[114] J. Mars and L. Tang. Whare-map: heterogeneity in homogeneous warehouse-scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[115] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *International Symposium on Microarchitecture (MICRO)*, 2011.

[116] J. Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[117] Meta. Meta Pro. `https://www.spaceglasses.com/`, 2014. [Online; accessed 10-November-2015].

[118] Metz, Cade. Facebook Catapults $1.5 Billion Datacenter int Iowa. `http://www.wired.com/2013/04/facebook-iowa-data-center/`. Online; accessed 10-November-2015.

[119] Microsoft. Meet Cortana. `http://www.windowsphone.com/en-us/how-to/wp8/cortana/meet-cortana`, 2014. [Online; accessed 10-November-2015].

[120] R. G. Miller. *Simultaneous Statistical Inference*. Springer, 1981.

[121] Miller, Rich. The Billion Dollar Datacenter. `http://www.datacenterknowledge.com/archives/2013/04/29/the-billion-dollar-data-centers/`, 2013. Online; accessed 10-November-2015.

[122] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *Transactions on Embedded Computing Systems (TECS)*, 2013.

[123] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *Static Analysis Symposium (SAS)*. 2011.

[124] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard. Quality of service profiling. In *International Conference on Software Engineering (ICSE)*, 2010.

[125] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Network and Distributed System Security (NDSS)*, 2015.

[126] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. Snnap: Approximate computing on programmable socs via neural acceleration. In *High Performance Computer Architecture (HPCA)*, 2015.

[127] S. Nanda, W. Li, L.-C. Lam, and T.-c. Chiueh. Bird: Binary interpretation using runtime disassembly. In *Code Generation and Optimization (CGO)*, 2006.

[128] C. H. Nguyen. Samsung poised to release 'Gear Glass' wearable late in 2014. `http://www.androidcentral.com/samsung-poised-release-gear-glass-wearable-late-2014`, 2014. [Online; accessed 10-November-2015].

[129] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In *Computer Security Applications Conference (ACSAC)*, 2010.

[130] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP)*, 2012.

[131] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium (SEC)*, 2013.

[132] Pelletier, Alexandre. Advanced exploitation of internet explorer heap overflow (Pwn2Own 2012 exploit). `http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php`. [Online; accessed 10-November-2015].

[133] C. Poynton. *Digital video and HD: Algorithms and Interfaces*. Elsevier, 2012.

[134] I. Puaut and D. Hardy. Predictable paging in real-time systems: A compiler approach. In *Euromicro Conference on Real-Time Systems (ECTRS)*, 2007.

[135] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *International Symposium on Microarchitecture (MICRO)*, 2006.

[136] T. Rain, M. Miller, and D. Weston. Exploitation trends: From potential risk to actual risk. In *RSA Conference*, 2015.

[137] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 2010.

[138] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener. Programming with relaxed synchronization. In *Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

[139] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *International Conference on Supercomputing (ICS)*, 2006.

[140] M. Rinard. Probabilistic accuracy bounds for perforated programs. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2011.

[141] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2010.

[142] M. Ringenburg, A. Sampson, I. Ackerman, L. Ceze, and D. Grossman. Monitoring and debugging the quality of results in approximate programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[143] S. Rus, R. Ashok, and D. X. Li. Automated locality optimization based on the reuse distance of string operations. In *Code Generation and Optimization (CGO)*, 2011.

[144] S. Russell and P. Norvig. Artificial intelligence: A modern approach. *Prentice Hall Press*, 1995.

[145] J. Salwan. Ropgadget tool. `http://shell-storm.org/project/ROPgadget/`, 2012. [Online; accessed 10-November-2015].

[146] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[147] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. SAGE: Self-tuning approximation for graphics engines. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[148] 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 *Programming Language Design and Implementation (PLDI)*, 2011.

[149] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *International Symposium on Microarchitecture (MICRO)*, 2013.

[150] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *Programming Language Design and Implementation (PLDI)*, 2014.

[151] D. Sanchez and C. Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *International Symposium on Computer Architecture (ISCA)*, 2011.

[152] A. Sandberg, D. Eklöv, and E. Hagersten. Reducing cache pollution through detection and elimination of non-temporal memory accesses. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[153] J. Sartori and R. Kumar. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. In *IEEE Transactions on Multimedia*. 2013.

[154] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *USENIX Security Symposium (SEC)*, 2011.

[155] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *Code Generation and Optimization (CGO)*, 2003.

[156] J. Seibert, H. Okkhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *Computer and Communications Security (CCS)*, 2014.

[157] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Computer and Communications Security (CCS)*, 2007.

[158] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[159] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Foundations of Software Engineering (FSE)*, 2011.

[160] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.5702&rep=rep1&type=pdf`, 2008. [Online; accessed 10-November-2015].

[161] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance. In *Dependable Systems and Networks (DSN)*, 2010.

[162] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP)*, 2013.

[163] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an os-level, software-only pollute buffer. In *International Symposium on Microarchitecture (MICRO)*, 2008.

[164] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti. A compiler-directed data prefetching scheme for chip multiprocessors. In *Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[165] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *Conference on Embedded Networked Sensor Systems (Sensys)*, 2007.

[166] B. Stancill, K. Z. Snow, N. Otterness, F. Monrose, L. Davi, and A.-R. Sadeghi. Check my profile: Leveraging static analysis for fast and accurate detection of ROP gadgets. In *Research in Attacks, Intrusions, and Defenses (RAID)*. 2013.

[167] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Security and Privacy (SP)*, 2013.

[168] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared l2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*, 2007.

[169] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: Mitigating contention for qos in warehouse scale computers. In *Code Generation and Optimization (CGO)*, 2012.

[170] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[171] P. Team. PaX address space layout randomization (ASLR), 2003.

[172] D. Ungar, D. Kimelman, and S. Adams. Inconsistency robustness for scalability in interactive concurrent-update in-memory molap cubes. *Inconsistency Robustness (IR)*, 2011.

[173] M. J. Voss and R. Eigemann. High-level adaptive program optimization with adapt. In *Principles and Practices of Parallel Programming (PPoPP)*, 2001.

[174] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Computer and Communications Security (CCS)*, 2012.

[175] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security and Privacy Magazine*, 2009.

[176] J. S. Witte, R. C. Elston, and L. R. Cardon. On the relative sample size required for multiple comparisons. *Statistics in medicine*, 2000.

[177] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Programming Language Design and Implementation (PLDI)*, 1991.

[178] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Dependable Systems and Networks (DSN)*, 2012.

[179] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: avoiding long tails in the cloud. In *Networked Systems Design and Implementation (NSDI)*, 2013.

[180] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2013.

[181] T. Y. Yeh, P. Faloutsos, M. Ercegovac, S. J. Patel, and G. Reinman. The art of deception: Adaptive precision reduction for area efficient physics acceleration. In *International Symposium on Microarchitecture (MICRO)*, 2007.

[182] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP)*, 2013.

[183] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium (SEC)*, 2013.

[184] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Parallel Architectures and Compilation Techniques (PACT)*, 2005.

[185] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *European conference on Computer Systems (EuroSys)*, 2009.

[186] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *European Conference on Computer Systems (EuroSys)*, 2013.

[187] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. Smite: Precise qos prediction on real system smt processors to improve utilization in warehouse scale computers. In *International Symposium on Microarchitecture (MICRO)*, 2014.

[188] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and scalable memory shadowing. In *Code Generation and Optimization (CGO)*, 2010.

[189] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *Principles of Programming Languages (POPL)*, 2012.

[190] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.