University of California
Santa Barbara

Performance-driven Analysis and Optimization in the Emerging System Architecture Communication Paradigms

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Zhizhou Zhang

Committee in charge:

Professor Timothy Sherwood, Chair
Professor Jonathan Balkind
Professor Yu Feng
Milind Chabbi, Ph.D., Uber

December 2022

The Dissertation of Zhizhou Zhang is approved.

Professor Jonathan Balkind

Professor Yu Feng

Milind Chabbi, Ph.D., Uber

Professor Timothy Sherwood, Committee Chair

December 2022

Performance-driven Analysis and Optimization in the Emerging System Architecture Communication Paradigms

by

Zhizhou Zhang

To my family.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Timothy Sherwood, for providing me with guidance, encouragement, and support throughout my Ph.D. journey. His unwavering support, encouragement, and insights have been invaluable in the completion of this dissertation. I learned tremendously from him in both academic research and beyond.

I would also like to thank the committee members, Professor Jonathan Balkind, Professor Yu Feng, and Dr. Milind Chabbi, for their time, expertise, and constructive feedback throughout my research. I enjoy and appreciate collaboration and discussion with Jon, teaching and feedback from Yu, and mentoring from Milind.

In addition, I would like to extend my sincere thanks to the following:

To all labmates and friends in UCSB ArchLab, CS Department, my collaborators at Uber, and my coauthors at the University of Rochester. Special thanks to Jeremy Lau, Deeksha Dangwal, Georgios Tzimpragos, Alvin Glova, Jennifer Volk, and Rhys Gretsch of ArchLab for the amazing experience. To Professor Yufei Ding, Yuke Wang, and Boyuan Feng of UCSB for the warm welcome and inspiring learning. To Adam Welc, Murali Krishna Ramanathan, Prithvi Raj, and Abhishek Parwal of Uber for the great collaboration. To Chen Ding, Sandhya Dwarkadas, Chencheng Ye, Sayak Chakraborti, and Noah Bertram of the University of Rochester for your fascinating cooperation. Special thanks to Chen, who inspires me in the research and Ph.D. program and is always willing to help. To my parents Minxing Zhang and Xiaoyuan Zhang for the enormous help to achieve this milestone.

Last, to my beloved partner, Wei Jin, for the continuous support, encouragement, and tutoring along this journey.

Thank you all.

Curriculum Vitæ

Zhizhou Zhang

Education

2022	Ph.D. in Computer Science, University of California, Santa Barbara.
2021	M.S in Computer Science, University of California, Santa Barbara.
2018	B.S. in Computer Science, University of Rochester.

Publications

[6] **Zhizhou Zhang**, Alvin Glova, Timothy Sherwood, and Jonathan Balkind. A Prediction System Service. ASPLOS 2023.

[5] Alvin Oliver Glova, Yukai Yang, Yiyao Wan, **Zhizhou Zhang**, George Michelogiannakis, Jonathan Balkind, and Timothy Sherwood. Establishing Cooperative Computation with Hardware Embassies. SEED 2022.

[4] **Zhizhou Zhang**, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. Critical Path Analysis of Large-scale Microservice Architectures. ATC 2022.

[3] Deeksha Dangwal, **Zhizhou Zhang**, Jedidiah R Crandall, and Timothy Sherwood. Context-Aware Privacy-Optimizing Address Tracing. SEED 2021.

[2] **Zhizhou Zhang**, Milind Chabbi, Adam Welc, and Timothy Sherwood. Optimistic Concurrency Control for Real-world Go Programs. ATC 2021.

[1] **Zhizhou Zhang**, Chencheng Ye, Rahman Lavaee, Ning Gu, and Chen Ding. Fine-grained data usage analysis by access sampling: seeing the data that is not there. MEMSYS 2018.

Abstract

Performance-driven Analysis and Optimization in the Emerging System Architecture Communication Paradigms

by

Zhizhou Zhang

Computation is increasingly complex, and our languages, systems, and design practices are changing to meet that shifting reality. That shift comes also with new challenges in debugging and optimization. In this thesis, we demonstrate that by adopting performance-driven analysis and optimization around communication paradigms, we are able to facilitate the development process and improve the performance of the programs. Specifically, we show such improvements in improving the synchronization mechanism in Golang by Hardware Transactional Memory (HTM), Critical Path Analysis (CPA) of Remote Procedure Call (RPC), and optimizing programs by novel service of prediction at Operating System (OS) level.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are increasing amounts of data generated every year and more computing power is needed to process the data. To meet the raising requirement of the computation, simply stacking more cores or machines is not enough. Each machine has physical limitations and many of the computations are not easily scalable. Instead, more novel programming languages, specialized tools and libraries, and new system architectures designs have been developed to provide fast and effective support for demanding computation.

For instance, new programming languages like Rust, Go, and Swift are developed since they can provide better support for concurrent programming, garbage collection, modularity, and scalability. Such new features help programmers develop, debug, and maintain more complex applications dramatically. At the same time, machine learning is more widely applied to optimize the performance of complicated programs. The strong capability of machine learning is able to find out solutions that easily outperform human experts in many scenarios. For application development, microservice designs are gradually more popular in the industry and the modern data center. Compared with the traditional monolithic design, microservices are noticeably superior in modularity, scalability, and deployability.

However, the shift comes also with new challenges because of the complexity. Specifically many of the features like concurrent programming are error-prone, machine learning methods are known for their indescribability, and microservices architecture is infamous to debug and optimize due to the extreme complexity. We will break down the detailed challenges of the new systems and tools below.

1.0.1 Concurrent programming

Golang [1] is one of the most representative new programming languages that has gained substantial popularity recently. It has a wide application range including both enterprise software [2] and open-source applications including Kubernetes [3] and libraries [4]. One of the reasons Go is popular is that Go has many syntax similarities to C, which is prevalent to many developers already. But unlike C, Go supports concurrency as a first-class language component. Specifically, Go provides straightforward and easy-to-use features for concurrent programming — any functions in Go can be scheduled to run concurrently by simply prefixing the function call with the `go` keyword.

Although such a feature makes writing concurrent programs much easier in Go, programmers still need to explicitly manage the interaction between concurrently running code. Normally there are two ways for the management: via passing messages through channels [1] or explicitly synchronization control to the shared memory. In reality, the latter approach is more widely used by developers. Among different synchronization access control mechanisms, mutual exclusion via lock [5] is still the most prevalent way by developers due to its simplicity [6].

However, it is well-known that locks can slow down the program by unnecessarily serializing the concurrent execution, even if there is no shared memory accessed by concurrently running code. There are many studies from the past in order to boost the

lock-based program execution. One of the noticeable solutions to locks is *transactional memory* (TM) [7]. Unlike lock, TM allows multiple pieces of code to be executed concurrently and speculatively. If there is a *conflict* [8] detected, i.e. more than two pieces of code touch the same memory and at least one of them is a memory write, at least one piece of code will be *rolled back* and restart the computation. Otherwise, all executions can proceed in parallel if no conflict has been detected.

TM is originally supported in software as STM [9, 10, 11, 12] and later available in commodity hardware as Hardware Transactional Memory (HTM), including IBM, Intel, and Arm [13, 14, 15]. However, despite the invention and research of TM over three decades, the real-world application of TM to accelerate software has not gone too far. There are several reasons that can potentially cause the gap.

One possible reason is that a lot of TM works from the past focused on designing and implementing TM algorithms but only measuring the performance on synthetic benchmarks (e.g., STMBench [16]) or evaluating selected concurrent data structures. When it comes to the real-life applications [9, 17], TM was not able to generate convincing performance.

Therefore, how to utilize the existing HTM techniques to boost real-world programs, specifically in Go because of its concurrency-first philosophy, is an unsolved but worthwhile challenge.

1.0.2 Runtime Optimization

The runtime optimization problem is a set of challenges faced by developers frequently. Consider when capable performance programmers find out the business-critical application is running slowly but there is no obvious cause, normally they need to navigate the potential optimization spaces for the trade-offs. However, the potential optimization

spaces can be huge, and quite often it is difficult to make the decisions right away. For instance, is it faster to run the code with Mutex lock as explicit synchronization control than wrapping it optimistically in a transaction via TM? Should certain functions be inlined or compiled with different optimization levels? To answer the questions mentioned above needs to balance a set of constraints. As our applications, system, and hardware grow complicated, it is harder to characterize and understand all potential parameters in the decision-making process, and even more difficult to find out an ideal solution using ad-hoc heuristics.

Luckily, *machine learning* has proved a good fit to navigate the complex optimization space and find out a good solution. The use case of machine learning approaches is not limited to optimizing data structures [18], implementing state-of-the-art recommendation systems [19], learning the structure and optimal access of the database [20], and improving anomaly detection [21]. For another case of TVM [22], machine learning is able to produce computing kernels that significantly outperform the design from human optimizers'.

Such a style of optimization is becoming more and more common in the past several years and it makes little sense for each application to use its own internal ML framework for parameter runtime optimization, where it elides the opportunities for sharing of memory or exploiting common hardware resources.

On the other hand, many of the works mentioned above utilize complex machine-learning models. Although they are capable to generate performant solutions, the high cost of the training and inference hinders a broader application, especially when the target itself is relatively simple. An ideal model should be able to generate accurate results with relatively low latency.

Therefore, it is time to consider what type of new *abstractions* is necessary to facilitate broader use of this important style of the runtime optimization problem and how to provide low-cost but good quality results for runtime optimization.

1.0.3 Microservices

Microservice architectures [23, 24, 25, 26, 27, 28, 29] have become the cornerstone of modern service-oriented software systems. Unlike monolithic software development and deployment, the microservice environment breaks down the business logic into individually deployable programs. Such an approach allows fast development and scalable deployment. Between the microservices, individual *instances* interact with each other via remote procedure calls (RPCs). As the business grows larger, microservices increase in number, and the interaction becomes much more complex. In addition, the RPCs are often deeply nested, asynchronous, and hierarchical, which makes it extremely difficult to identify the source of the problems. However, understanding microservices of the business-critical request and how to optimize them is very important for developers.

We can consider Uber's backend as an example of microservice architecture. There are around ∼4,000 microservices interacting with each other via RPCs. What is more, each microservice hosts several APIs. In total, there are more than $40,000$ unique RPC endpoints that can call each other in complex ways.

When a service request arrives at an entry point API to the backend system, it needs multiple "hops" through many RPCs before being fully serviced. The interactions are also asynchronous, deeply nested, and can invoke multiple other downstream APIs. As a result, it is extremely hard to identify which service(s) contribute to the overall end-to-end latency when top-level request performance degradation is detected [30, 31, 32, 33, 34, 35, 36]. The answer to the question is very important in many scenarios, including

- Identifying potential opportunities to optimize top-level microservice
- Spotting bottleneck APIs that affect other endpoints
- Setting appropriate time-to-live threshold for RPCs
- Diagnosing error and outages conditions

- Advising for computing and other datacenter management

Therefore, certain analysis is needed for better microservice architecture at the datacenter scale.

1.0.4 Commonality of the Challenges

Although the three cases mentioned above are quite different in terms of the exact reasons for the challenge, they share certain commonalities if we examine them carefully.

The fundamental challenge for the three cases is around communication paradigms. Hardware lock elision (HLE) is focused on the synchronization mechanism between different cores or threads. How to utilize different communication schemes including transactions and Mutex locks is the region of interest. Meanwhile, the key to solving runtime optimization problems using machine learning is to find the proper abstraction of the fundamental machine learning services and provide efficient implementation at the right location accordingly. The communication paradigm is crucial to consider when we need to design a component that allows Operating System (OS) to communicate with user-level applications efficiently. Last but not least, communication paradigms are the cornerstone of Remote Procedure Calls (RPCs), which implement the microservices at the datacenter. In order to analyze the microservices well, it is required to understand the nature of the RPCs and come up with appropriate tools.

In terms of how to solve these challenges, we propose to use performance-driven analysis and optimization. The challenges are from real-world applications with specific performance requirements. They are also complex and dynamic in nature, hence performance-driven analysis and optimization is one of the best-fitting methodologies.

Therefore, we present the thesis statement below after observing those commonalities.

Thesis Statement

In this thesis, we demonstrate that by adopting performance-driven analysis and optimization around communication paradigms, we are able to facilitate the development process and improve the performance of the programs. Specifically, we show such techniques in improving the synchronization mechanism in Golang by Hardware Transactional Memory (HTM), optimizing programs by novel service of prediction at the Operating System (OS) level, and analyzing the complex system within a datacenter by Critical Path Analysis (CPA) of Remote Procedure Call (RPC).

For synchronization mechanism improvement in Golang, we propose a framework, named GoCC, to utilize Hardware Transactional Memory to speed up certain Mutex locks. The framework identifies Mutex locks from the source file, analyzes the valid replacement candidates, rewrites the code with HTM automatically, and provides an efficient runtime library. The analysis includes detailed and thorough static analysis at the intermediate representation (IR) level so that the transformation from Mutex locks to HTM is always safe and valid. The runtime library contains a lightweight controller that can choose HTM or Mutex locks efficiently and effectively at runtime.

For prediction system service (PSS), we propose a new system-level service that can provide a simple but effective prediction inspired by GoCC. The prediction can be shared among different applications and tasks at both the user level and kernel level. The prediction-based mechanism is particularly helpful for certain types of runtime optimization problems. By effectively and efficiently guiding the decision at the runtime, PSS is able to improve the performance of applications in different fields.

For microservices at datacenter scale, we adopt critical path analysis (CPA) on the sampled trace to pinpoint the top contributor to the end-to-end latency. We built a tool named CRISP to extract the critical path from the collected trace at Uber and developed

tools to help programmers debug the service. CRISP also enhances anomaly detection by reducing training and inferencing time and improving model accuracy.

In summary, we adopted performance-driven analysis and optimization around communication paradigms for emerging programs and systems. We believe such an approach is able to further help developers optimize and debug complex applications.

The structure of the rest of the thesis follows:

- In Chapter 2, we study on how to accelerate Mutex lock with Hardware Transactional Memory (HTM) in Golang. We propose a framework named GoCC that is able to identify, analyze, and replace the Mutex with HTM in the given Golang source code. The procedure in the framework is sound, automatic, and efficient. With the help of a newly designed runtime controller, GoCC is able to speed up Golang programs with limited downside.

- In Chapter 3, we show how to solve several dynamic optimization problems with a novel prediction system service (PSS). PSS is able to provide a cheap but efficient prediction mechanism for a variety of tasks at both user-level and kernel-level and allows model updates based on the correctness of the prediction. To validate the broad applicability of PSS, we apply it in three different scenarios: Hardware Lock Elision (HLE), parameter tuning for Just-In-Time (JIT) compiler, and page reclaim of memory management in Operating System.

- In Chapter 4, we illustrate how to perform critical path analysis (CPA) on the microservice traces and enable better debugging and optimization abilities for the developers. We build a tool named CRISP, which is able to provide several popular tools for debugging, including flamegraph and heatmap. CRISP can also offer significant speedup for anomaly detection and improve prediction accuracy noticeably. Besides, CRISP can also guide the hardware selection given the emerging

microservice workload.

Chapter 2

Optimistic Concurrency Control for Real-world Go Programs

2.1 Introduction

Golang [1] (or simply Go) is a modern programming language that has gained significant popularity over the last decade. It is being used to write enterprise software [2] (e.g., to implement backend services) in some of the largest technology companies as well as to develop large and widely used open-source applications (e.g., Kubernetes [3]) and libraries (e.g., Tally [4]). The design of Go is inspired by C, but unlike C, it supports concurrency as a first-class language construct. Even more importantly, and unlike other popular languages with first-class concurrency support (e.g., Java), the Go language goes to great lengths to simplify concurrent programming by making concurrency easy to use (and thus frequently used) by the developers [6] — any function in Go can be scheduled to execute concurrently with the rest of the code as a *goroutine* [1] by simply prefixing its call with the `go` keyword.

Although Go makes writing concurrent programs easier, it still requires programmers

to manage interactions between concurrently executing code — this can be accomplished either via passing messages through channels [1] or explicitly synchronizing accesses to shared memory. Shared memory is used more often than message passing by Go developers, and mutual exclusion via locks [5] remains the most widely-used synchronization mechanism across several applications [6]. It is, therefore, the focus of our work.

Locks may unnecessarily serialize concurrent execution, even if the code operates on disjoint data. Our work aims to improve the performance of concurrent Go code, particularly code hiding behind needlessly held locks. Our goal is to accomplish this while retaining the correctness of concurrent execution. We utilize the concept of *transactional memory* (TM) [7] to achieve this goal. The general idea behind TM is to decide on whether two (or more) pieces of code can be executed concurrently based on whether their accesses to the underlying data are *conflicting* [8] or not, that is, if at least one of the accesses is a write. Conflict-free executions are allowed to proceed in parallel. On the other hand, upon encountering a data access conflict, execution effects of at least one piece of code have to be *rolled back* (i.e., undone), and the computation must be restarted. TM machinery, which originally started in software (STM) [9, 10, 11, 12], is now available in commodity hardware as Hardware Transactional Memory (HTM) [13, 14, 15]. However, despite almost three decades of work in this area, TM's promise of accelerating concurrent computations for real-life software has not been quite fulfilled. We speculate that there are two reasons why this is the case.

The first reason is that TM while being a single concept, may have different realizations in terms of algorithms and implementations (e.g., eager vs. lazy versioning [37]) and different integration strategies at the language level (e.g., API-level solutions [12] or the compiler-assisted `atomic` construct used to demarcate TM-managed concurrent code [38]), resulting in different behavior from the programmer's perspective. Consequently, attempts to introduce TM as a separate language-level mechanism lead

to significant semantic dissonance with respect to existing concurrency-related mechanisms [39, 40].

The second reason is that a lot of TM (particularly STM) work was focusing on designing and implementing TM algorithms but limiting empirical evaluation to synthetic benchmarks (e.g., STMBench7 [16]) or measuring the performance of only selected concurrent data structures. Unfortunately, unlike what was expected, TM techniques did not easily generalize to real-life applications [9]. A few attempts to apply TM to production code were unsuccessful (e.g., an attempt to rewrite the Quake game server to use TM [17]).

In this work, we attempt to rectify some of these limitations and show that TM can be effective in accelerating real-life concurrent code. We focus less on the algorithmic side of TM (we use state-of-the-art off-the-shelf HTM implementation from Intel), and more on how and when to apply the TM machinery to maximize the benefit. Additionally, we replace Go locks with HTM constructs without changing the code's behavior in any way, which allows us to completely bypass complications related to transactional memory semantics. More specifically, we employ *transactional lock elision* (TLE) [41] — a well-known technique that attempts to execute a lock-protected critical section as an atomic hardware transaction, reverting to using the lock if these attempts fail.

Figure 4.9 depicts our solution. At a high level, our solution starts with using static analysis to identify candidate lock-protected critical sections to be instead protected by the HTM. Then we filter out non-desirable candidates using both static analyses (e.g., to eliminate regions containing I/O operations) and dynamic analysis (to eliminate regions where the application of the HTM would not be beneficial based on profile data collected at runtime). Finally, we rewrite the code to have candidate regions use HTM constructs provided by the HTM library we developed instead of Go locks [5]. GOCC transformations are guaranteed to be safe; developer involvement is optional but highly recommended to

let developers ultimately decide whether or not they want to use HTM.

In summary, this work makes the following contributions:

1. We present the design and implementation of a framework for identifying lock-protected critical sections and select the best candidates for lock elision based on static analysis and execution profiles of Go programs.
2. We describe the source-to-source code transformation to replace mutual-exclusion locks in Go programs with HTM concurrency control constructs.
3. We introduce a library extending vendor-provided HTM primitives with intelligent features such as runtime contention management. Specifically, we devise a lightweight perceptron [42, 43] that learns whether eliding a lock via HTM at a call site [44] is beneficial at runtime.
4. We demonstrate the effectiveness of GOCC for improving performance of real-life concurrent Go code by up to 10×.

2.2 Challenges

Locks are widely used in the real-world Go code and a significant amount of execution time can be spent waiting to acquire them [6, 45, 46, 47, 48, 49, 50] [1]. It is possible to replace a lock with a transaction that enables a critical section to be speculatively executed without actually holding the guarding lock. With the support of the HTM, such replacements can result in significant speedups. However, there are several challenges in performing these replacements correctly and robustly and ensuring that they deliver high performance reliably.

First, automatically and accurately matching a lock with its corresponding unlock op-

[1] A limited study we performed in a large-scale industrial setting using thousands of different Go services showed up to 30% execution time being spent in lock-related code in certain Go programs; 5-10% was quite common.

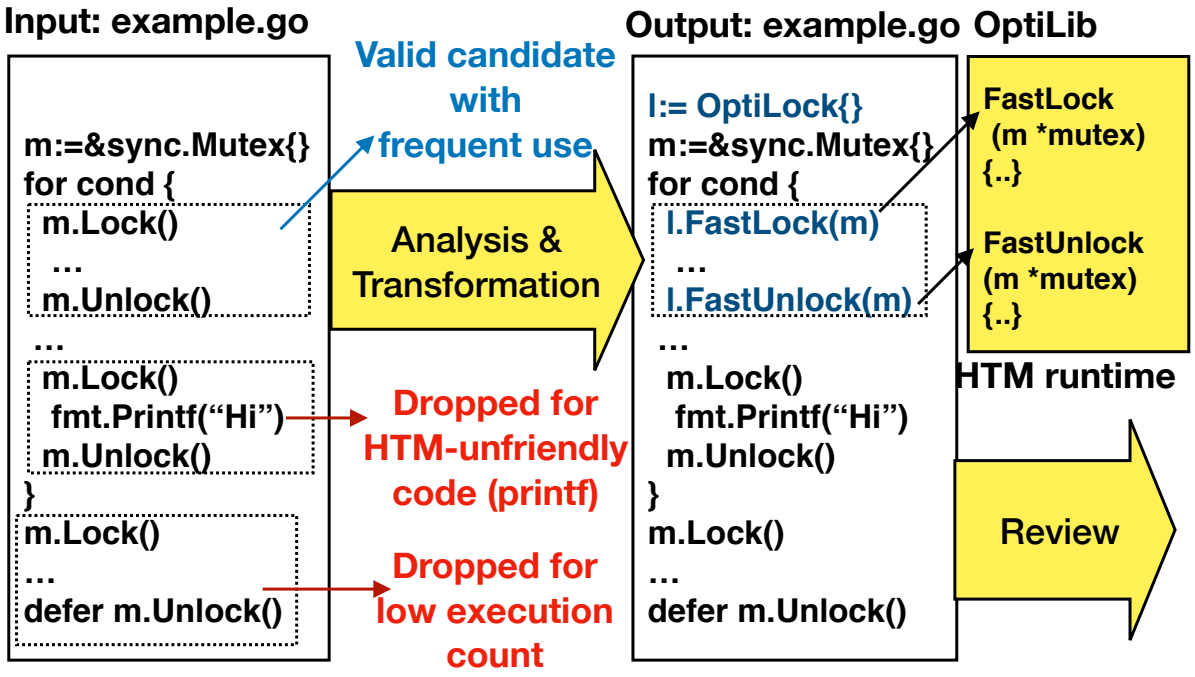


Figure 2.1: GOCC schematic diagram. Static analysis detects three legal lock-unlock pairs in the input file `example.go`. The top one is a valid replacement candidate. The middle one is filtered since it contains I/O operations in its critical section. The bottom one is dropped due to the infrequent use via the information provided by profiles. The transformed code calls `optiLib`, which executes the critical section via HTM. The resulting diff is given to the developer for review.

eration to precisely identify critical sections is a complex problem. Real-world programs can use locks with nesting intra- or inter-procedurally, which makes it significantly more involved. Additionally, certain lock-compatible instructions (e.g., IO and privileged instructions) will not work with HTM. A critical section including such instructions will not benefit from HTM.

Furthermore, Go provides a keyword that enables delaying lock release operation to all exit points of a function by prefixing the `Unlock()` operation with the `defer` [51] keyword[2]. It not only complicates matching an unlock with a lock operation, it may unnecessarily lengthen a critical section, which according to a synthetic benchmark we

[2]Any function can be deferred in Go.

wrote shows performance degradation. A scan of 21 million lines of industrial Go code, which includes about 8000 `Unlock()` operations, shows that about 76% are prefixed with the `defer` keyword. This indicates that handling `defer` statements is important.

Second, the Go language nuances [1] (e.g., pointer vs. value syntax, anonymous `Mutex` fields, lambda functions, etc.) make it non-trivial to transform lock-based code to HTM-based code.

Third, HTM has startup and commit overheads. Even in non-concurrent code, where data-access conflicts do not happen, HTM can fail [52], and locks may outperform HTM, particularly on tiny critical sections [53].

Fourth, the critical section size can be hard to estimate in general. If we make the conservative design choice and do not replace the lock if the critical section size is unknown, we can miss the opportunity to generate significant performance improvement. Thus, we need some runtime mechanism that can handle critical sections of arbitrary sizes with low overhead.

Fifth, when HTM aborts for a genuine data-access conflict, naively falling back to using a lock can be detrimental to performance [54, 55]. Deciding when and how to retry HTM-based executions or fall back to using fine-grained locks must be handled very carefully to avoid pathologies [54, 55, 56].

Our tool, GOCC, attempts to solve the above challenges. GOCC is an end-to-end system for improving the performance of lock-based Go code using HTM. We devise a sophisticated program analysis to identify lock-protected critical sections (§ 2.5.2), support lock-to-HTM code transformation including non-trivial Go features (§ 2.5.3), and develop an efficient HTM library to handle issues manifested at runtime (§ 2.5.4).

2.3 Related Work

Herlihy and Moss proposed transactional Memory (TM) [7] in 1993 as an alternative to locks. While locks proactively prevent two or more threads from concurrently accessing shared data, TM takes the opposite approach — concurrent accesses are allowed as long as they do not conflict. A lot of work has been done around both software and hardware implementations of transactional memory [10, 11, 12, 13, 14, 15, 57], but only a few [9, 17, 58, 59] focused on evaluating the approach with real-life workloads, and none have done this for Go.

Intel's TSX extension of x86 instructions set [60] implementing HTM is of specific interest here as it underlies parts of our implementation. It is widely available in modern Intel CPUs and offers software interfaces providing subtly different functionality. The RTM (Restricted Transactional Memory) interface allows programmers to execute arbitrary code as a hardware transaction. All operations within a transaction have atomic execution behavior — they all either appear to happen instantaneously or the entire transaction aborts and reverts the architectural state to before it was started. This can be trivially used to emulate the behavior of mutual-exclusion locks. In fact, this is precisely the kind of functionality that the HLE (Hardware Lock Elision) interface provides. However, HLE has been introduced mainly for backward compatibility with architectures that are not TSX-enabled and is not only very simplistic (e.g., with respect to contention management) but has also been shown to perform poorly compared to RTM [61]. Consequently, our solution uses the RTM interface as the low-level implementation mechanism to build a comprehensive TM-based alternative for mutual-exclusion locks.

Lock elision, whether in software or hardware or a hybrid fashion, including gaining insights into them, has been extensively studied [62, 56, 63, 57, 64, 65, 66, 67, 68, 69, 57, 70, 14, 71, 72, 73, 74, 75, 76, 77]. Our work uses many of those techniques; for example,

the basic design of our runtime controller was inspired by Wang et al. [75]. Additional possibilities to bring more solutions from the literature to the design and implementation of both our static analysis tool and runtime controller also exist. Other attempts to use transactional memory for emulating mutual-exclusion locks exist as well [78, 79], but they have to cope with higher overheads and semantics-related complications due to using the STM, they target the Java language whose synchronization lock-like primitives (i.e., *monitors*) are easier to handle due to their lexical scoping and, most importantly, their evaluation is based exclusively on synthetic benchmarks.

2.4 Gocc Overview

A Go `Mutex` is a runtime object with `Lock()` and `Unlock()` operations on it. Two (or more) critical sections guarded by the same `Mutex` will not execute concurrently. When transforming locks into HTM, there are two possibilities.

1. A given `Mutex` guarding a set of critical sections is replaced with another object supporting operations analogous to `Lock()/Unlock()` but provided by the HTM. As a result, all critical sections previously guarded by the `Mutex` are now executed under HTM's control.
2. `Lock()/Unlock()` operations of the `Mutex` are replaced with their HTM equivalents on a per critical section basis. As a result, some critical sections for a given `Mutex` are still guarded by the same `Mutex`, while the others execute under HTM's control.

The former is doable only if it is beneficial to transform all `Lock()/Unlock()` operations using a given `Mutex`, and the `Mutex` object is defined in the code that we are rewriting. Assessing the benefit of transforming the `Mutex` object would require inspecting every critical section it protects. A "may alias" pointer analysis [80, 81] can answer such a question. The "all-or-none" coarse-granularity of this approach makes it

unattractive because the imprecision of pointer analysis overapproximates the critical sections protected by a `Mutex`, disqualifying too many `Mutex`es from transformation.

This work adopts the latter approach, where we consider pairs of `Lock()/Unlock()` operations in the code for transformation, which provides fine-grained control over transformation. This approach has to handle pairing a lock with its corresponding unlock and support interoperability of HTM (where the code is transformed) with locks (where the code is not transformed). This kind of interoperability is well-studied in the literature [82, 83, 13, 84, 55] and is handled by our library.

Recall, from Figure 4.9, that input to GOCC is the source code for a Go package along with its execution profiles. The output is a source code patch, where candidate `Lock()/Unlock()` operations are replaced with calls to a custom HTM library. GOCC consists of the following key components:

- Analyzer: performs static analysis on the input program and collects lock-unlock pairs for transformation (§ 2.5.2).
- Transformer: rewrites the program by replacing `Lock()/Unlock()` with HTM, i.e. `FastLock()/FastUnlock()`, which elide the lock (§ 2.5.3).
- Adaptive runtime (`optiLib`): implements HTM in Go and provides required runtime mechanisms including retry and rollback (§ 2.5.4).

The source code patch choice, rather than a compiler transformation, is motivated by the desire to keep the developers in the loop. Using HTM without developers' knowledge can prove unwelcome because developers often demand full visibility into their programs. Developers are becoming performance and variance sensitive [85, 86, 87], and an accidental regression can become hard to diagnose. As a side effect, the choice of source-code patch demands us to be surgical — injecting large, complicated HTM-handling boilerplate code is a non-starter. Consequently, we perform `Lock()/Unlock()` operations replacements with API calls to HTM logic hidden in the `optiLib` open-source library

and do so only in places where benefits of HTM are likely (e.g., we minimize the number of modified code locations using execution profiles).

2.4.1 Gocc Guarantees and Limitations

- GOCC will transform properly synchronized code (i.e., where every lock operation will have a corresponding unlock operation) into the equivalent code without changing the code's behavior. Code not meeting this criterion will be either not transformed or transformed and its runtime behavior will be unchanged.
- GOCC considers only those lock-unlock pairs that seem to operate on the same lock within the same function — inter-procedural `Lock()/Unlock()` operations are disregarded. Note, however, that in a critical section protected by GOCC transformed lock can make arbitrary function calls. The requirement to have both `Lock()` and its matching `Unlock()` operation be present in the same procedure scope is only our implementation choice and pragmatic in nature. Over 70% of the locks, we inspected met this criterion.
- GOCC makes no effort to identify critical sections or code reachability in the presence of reflection [88].
- GOCC, as implemented, does not *statically* detect HTM conflicts or capacity limitations (see § 2.5.2 for the details).

2.5 Gocc Design and Implementation

Before diving into the details of GOCCs design and implementation, we define some common terminology.

2.5.1 Terminology

Go's `sync` package provides two kinds of shared memory objects: `Mutex` and `RWMutex`. GOCC handles them both, but in the following sections, without the loss of generality, we will only use the term `Mutex` for simplicity. From an HTM transformation viewpoint, an `RWMutex` is no different from a `Mutex`, except `RWMutex` offers additional APIs for read-only accesses.

A *critical section* **CS** is all code regions protected by a pair of lock and unlock operations on the same mutex object `m` — the notation for calling lock/unlock operations on `m` is `m.Lock()`/`m.Unlock()` where `m` is referred to as a *receiver*. *Lock-point*, abbreviated with letter L (*Unlock-point* abbreviated with letter U), is a static location in the code where the `Lock()` (`Unlock()`) function is invoked on a `Mutex`. **LU-points** is a set of L and U points. **LU-pair** is a candidate pair of one *lock-point* paired with an *unlock-point*. In the runtime context, fastpath/HTM-path means the use of HTM, and slowpath/fallback-path means the use of the original lock.

We utilize the Abstract Syntax Tree (AST), program Control Flow Graph [89] (CFG), and Static Single Assignment (SSA) [90] form of program representation prevalent in the compiler literature. In a CFG, nodes are basic blocks [89] of straight-line code, and edges are control flow relationships among them. GOCC first transforms the source code to the AST form (which is also used for code transformation as described in § 2.5.3) and then to the SSA form for CFG construction.

2.5.2 Analyzer

The goal of the analyzer is to find as many LU-pairs as possible. The LU-pairs that protect HTM-incompatible critical sections (e.g., those including IO operations) must be pruned. This filtering serves two purposes: it reduces the number of code changes

and non-beneficial HTM transformations. Complicated lock usage patterns, several Go language quirks, and pointer imprecision complicate the static analysis. A comprehensive call-graph analysis is vital because critical sections often contain function calls.

Conflicts: A sophisticated static analysis may detect whether transactions conflict. Answering this question, however, is unlikely to be valuable because developers typically do not use a lock if a conflict is impossible. Assuming conflicts happen, there is no easy way to statically determine whether transactions do not "typically" conflict. We do not try to solve this problem and leave conflict resolution to `optiLib`.

Capacity: Although one can perform static analysis to estimate the memory footprint of a critical section, it may not be possible if the bounds of a loop are unknown. Also, without knowing the target architecture's HTM capacity, it would be premature to filter out candidate critical sections this way. We leave the capacity-related decisions also to `optiLib`.

In the rest of this section, we, first, define the scope of our transformation (§ 2.5.2); then, describe the process of matching a lock with an unlock operation within a code region assuming no lock nesting and no function calls in a **CS**; extend our analysis to include nested locks (§ 2.5.2); expand the analysis scope to **CS**s that may contain function calls (§ 2.5.2); detail special case of Go's `defer` statement (§ 2.5.2); and finally discuss profile-based filtering (§ 2.5.2).

Scope of Transformation

To simplify the analysis, if a `Lock()/Unlock()` operation is executed in the middle of a basic block, we break such basic blocks in the CFG so that each lock-point begins a new basic block and each unlock-point ends a basic block. A single-entry single-exit (SESE) region [91] (simply *region*) of a CFG is our smallest granularity of lock transformation. A region is a subgraph of a CFG. Control reaching any basic block in a region is guaranteed

to have already executed a designated entry basic block; control leaving from any basic block in the region is guaranteed to eventually pass through a designated exit basic block.

A function is the largest granularity of our lock transformation; a function always forms a region because all exits from a function are considered to go through a dummy basic block. This choice is pragmatic in nature since LU-pairs spanning multiple functions are uncommon.

Regions can be nested within one another. A Program Structure Tree (PST) organizes regions into a hierarchical tree [91]. We visit regions inside out from most-nested to least-nested. Section 2.5.2 describes the region identification and visiting strategies, which are not central to this work.

Splicing SESE regions for maximal LU-pairing

Although the classic definition of a program structure tree PST [91] provides hierarchical access to SESE regions, one may miss certain lock matching opportunities if multiple lock and unlock statements happen in a straight-line sequence, as shown in Figure 2.2. In this example, there is no conditional execution or loops from the first statement to the last hence, even though LU-points split the basic blocks, the standard SESE would place all such basic blocks in a single innermost region without further nesting. This would complicate which lock to pair with which unlock since the region has more than one LU-pair.

We solve the pairing problem in such straight-line code sections by performing additional processing to splice a region further based on LU-points. The idea is simple; we utilize the dominator and post-dominator trees [89] for the region. Only the nodes that contain a lock-point are of interest in the dominator tree ($DomTree$). Only the nodes that contain an unlock-point are of interest in the post-dominator tree ($PDomTree$).

We perform a post-order traversal of the $DomTree$. For each lock-point L in this

traversal in $DomTree$, we look up L's immediate post-dominator in the $PDomTree$, which is an unlock-point U. We then perform the reverse test, we look up the immediate dominator of U in the $DomTree$ that is a lock-point, say, $\hat{L}$. If $L = \hat{L}$, we have the innermost matching of a candidate LU-pair. We also check that their points-to sets intersect. If no matches are found, we traverse up in the $DomTree$ till a match is found and drop the candidate L if none is found.

Matched pairs are no longer considered when matching other lock-points or unlock-points during the rest of the post-order traversal of nodes higher-up in the $DomTree$. This strategy breaks a straight-line sequence into the maximum number of LU-pairs, as shown with demarcated SESE-regions in Figure 2.2.

Matching LU-pair in the Absence of Nested Locks

This subsection discusses analyzing a candidate region R.

LU-points in R may be operating on different locks, which should be pruned. Some lock (unlock) operations may escape R, without a corresponding unlock (lock) operation in R, which should also be pruned. Below, we formalize these aspects.

Definition 2.5.1 (Points-to set $\mathcal{M}(L)$ of a Lock point L)**.** *Every lock-point (L) operates on some receiver mutex pointer* `p`.[3] *Such a mutex pointer may point to one or more mutex objects allocated in the program. The set of all possible* `Mutex` *objects that* `p` *may point to in the program is the Points-to Set of L, denoted by $\mathcal{M}(L)$.*

Similarly, the Points-to set of an Unlock point U is $\mathcal{M}(U)$. We employ Anderson's flow-insensitive may-alias analysis [92] to obtain $\mathcal{M}(L)$ and $\mathcal{M}(U)$ on the whole program.

[3]At the source level `p` can be either a pointer or an actual object value, but at the SSA level it is always a pointer.

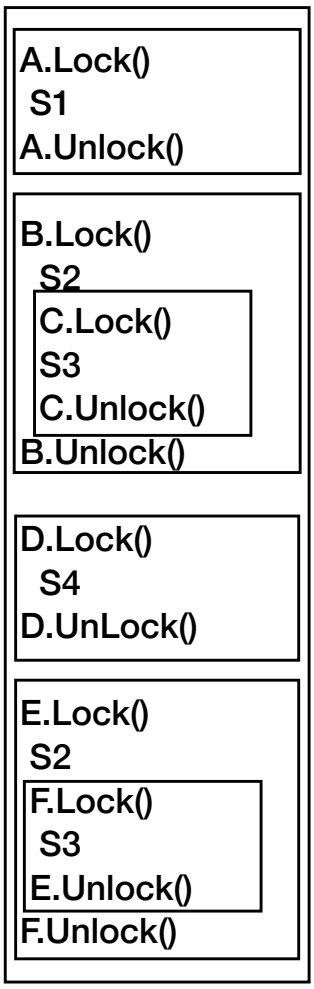


Figure 2.2: A straight-line sequence of statements that fall into a single region are further spliced into regions based on matching a lock-point with the nearest post-dominating unlock-point and unlock-point with the nearest dominating lock-point.

```
1
2 m := &sync.Mutex{}
3 m.Lock()
4 m.Unlock()
```

Listing 2.1: Original lock-based code.

```
1 l := OptiLock{}
2 m := &sync.Mutex{}
3 l.FastLock(m)
4 l.FastUnock(m)
```

Listing 2.2: Transformed HTM code.

Definition 2.5.2 (Downward Exposed Lock-point (**DELock**))**.** *A lock-point, L, with points-to set $\mathcal{M}(L)$, is downward exposed in the region R, if there exists at least one path from L to R's exit without any unlock-point on any mutex in $\mathcal{M}(L)$.*

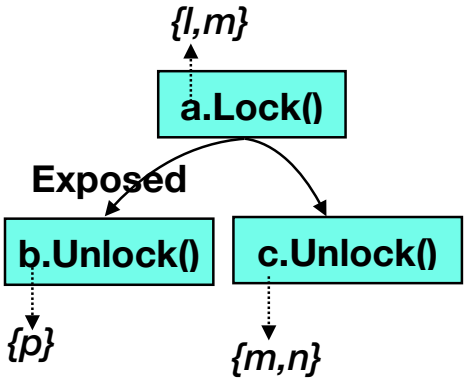


Figure 2.3: `a.Lock()` is Downward Exposed.

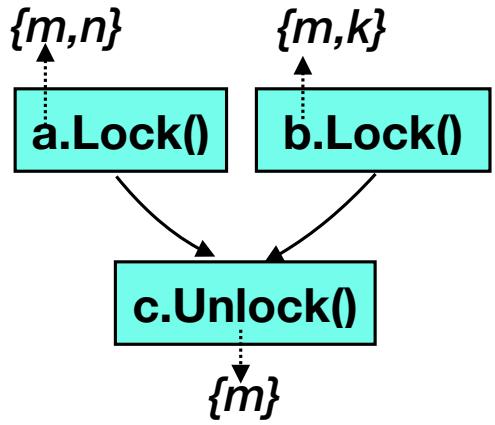


Figure 2.4: `c.Unlock()` is **not** Upward Exposed.

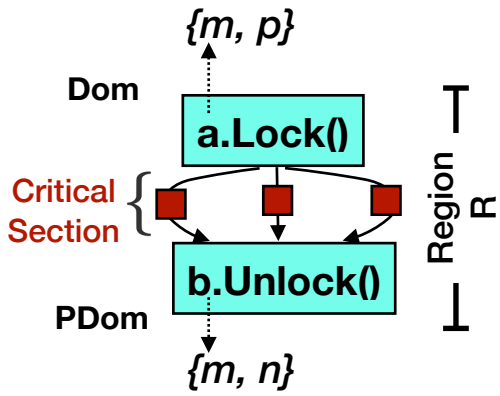


Figure 2.5: Region dominated by lock and post-dominated by unlock.

Definition 2.5.3 (Upward Exposed Unlock-point (**UEUnlock**)). *An unlock-point, U, with points-to set $\mathcal{M}(U)$, is upward exposed in the region R if there exists at least one path from R's entry to U without a lock-point on any mutex in $\mathcal{M}(U)$.*

DELock identifies lock-points that *definitely* do not have any corresponding unlock-points in some execution paths in R; and *UEUnlock* identifies unlock-points that *definitely* do not have corresponding lock-points in some execution paths in R.

Figure 2.3 exemplifies a downward exposed lock-point. Mutex pointer a's points-to set $\{l, m\}$, has an empty intersection with b's points to set $\{p\}$; although it has a non-empty

intersection with c's points-to set $\{m, n\}$. Figure 2.4 exemplifies an unlock-point that is *not* upward exposed. Mutex pointer c's points-to set $\{m\}$, has non-empty intersection with a's points-to set $\{m, n\}$ and b's points-to set $\{m, k\}$.

We eliminate all $DELock(R)$ and $UEUnlock(R)$ from the transformation in R. The remaining lock-points in R are the complement of $DELock(R)$, which is denoted by $\overline{DELock(R)}$. Similarly, the remaining unlock-points in R are the complement of $UEUnlock(R)$, which is denoted by $\overline{UEUnlock(R)}$.

Definition 2.5.4 (Feasible-HTM-Pair)**.** *Let* $L \in \overline{DELock(R)}$*. Let* $U \in \overline{UEUnock(R)}$*. L and U form a feasible HTM pair if all of the following conditions are true,*

(1) $\mathcal{M}(L) \cap \mathcal{M}(U) \neq \phi$*,*

(2) $\big(L \text{ Dom } U\big) \bigwedge \big(U \text{ PDom } L\big)$*,*

(3) The critical section $C \subseteq R$ guarded by L and U contains no LU-point X such that $\mathcal{M}(X) \cap \big(\mathcal{M}(L) \cup \mathcal{M}(U)\big) \neq \phi$*, and*

(4) C contains no HTM-unfriendly instructions.

Condition (1) filters out those LU-points that are guaranteed to be operating on different `Mutex`es.

Condition (2) filters out infeasible control flows where unlock happens before lock and vice-versa. Dom and PDom respectively represent dominator [89] and post-dominator [89] relationships in a CFG. Figure 2.5 shows an example, where all paths from lock-point `a.Lock()` are post-dominated by unlock-point `b.Unlock()`, whose all incoming paths are dominated by `a.Lock()`. Additionally, the set-intersection of the points-to set of mutex pointers $a = \{m, p\}$ and $b = \{m, n\}$ is non-empty. Any Feasible-HTM-Pair on L and U, forms an SESE-region by itself, where the entry basic block has L as its first instruction and the exit basic block has U as its last instruction. Condition (2) intuitively finds correct candidate LU-pairs in the absence of nested locks because if

a lock operation L is performed on every path reaching any code in C and an unlock operation U is performed on every path exiting C, then LU must be operating on the same `Mutex`. Section 2.5.2 justifies our choice of DOM/PDOM relationships.

Condition (3) ensures that if we match an L with a U, there does not exist another lock-point or unlock-point in the same region that *may* operate on a `Mutex` in the same points-to set as that of L or U. The next subsection elaborates on lock nesting.

Condition (4) is an obvious requirement to ensure HTM does not abort. A region is unsafe if it contains any IO instructions.

Since we use "may alias" to match a lock-point with unlock-point, it is possible (but less likely) for our transformation to pair a lock with an unlock that may be operating on two different mutex objects at runtime. However, at runtime, we can obtain and memorize the address of the mutex object used at the lock-point, and compare it against the mutex object offered to the runtime at the unlock-point. In case of an address mismatch of the mutex objects used in the same LU-pair, we can abort the transaction and revert to a safe state and fall back to using the locks. A mismatch is impossible without nested locks because of the dominance and post-dominance relationship between the lock and unlock in an LU-pair.

Justification for using the dominance and post-dominance relationship

The DOM and PDOM requirements proposed in Definition 2.5.4: Condition(2) may seem rather too strong. In a more complex control flow shown in Listing 2.3, Condition (2) does not hold good because neither lock-point dominates any unlock-point; however, both lock-points taken together guarantee a lock-point execution before executing an unlock-point and similarly, both unlock-point taken together ensure that after executing any lock-point, one unlock-point is guaranteed to execute. Thus, it would be valid to transform all LU-points with HTM here.

```
1 if cond1 {
2   m.Lock()
3 } else {
4   m.Lock()
5 }
6 if cond2 {
7   m.Unlock()
8 } else {
9   m.Unlock()
10 }
```

Listing 2.3: Complex locking control flow amenable for HTM.

```
1 if cond1 {
2   m.Lock()
3   // in critical section
4 }
5 // may be critical section
6
7 if cond2 {
8   // in critical section
9   m.Unlock()
10 }
```

Listing 2.4: Complex locking control flow unsuitable for HTM.

A slightly different control flow in Listing 2.4, however, shows that the lack of DOM and PDOM relationship is not easy to handle. In this case, the execution of the lock-point does not ensure the execution of unlock-point also or vice versa. Hence, transforming both LU-points does not provide any guarantee.

Any "umbrella covering" analysis may artificially drag non-critical sections into critical sections. With this observation, we enforce condition (2).

Lock Nesting

Go supports nested locks, but reentrant [93] locks are not allowed. Condition (3) in Definition 2.5.4 allows nested locks but demands that they operate on disjoint `Mutex` objects.

HTM via Intel TSX allows nesting: if a nested transaction succeeds, hardware does not commit it until the outermost transaction commits. If a nested transaction fails, the control jumps to the starting code address of the nested transaction. This facility allows us to safely transform locks into HTM even when they are nested.

Condition (3) in Definition 2.5.4 disqualifies a candidate LU-pair from the transfor-

```
1 a.Lock() //outer region start
2
3
4 b.Lock()  // inner region start
5 b.Unlock() // inner region end
6
7 a.Unlock() //outer region end
```

Listing 2.5: Nested Locks.

```
1 a.Lock()
2
3 l := OptiLock{}
4 l.FastLock(b)
5 l.FastUnlock(b)
6
7 a.Unlock()
```

Listing 2.6: HTMized.

mation in the region R if there exists any other lock or unlock point whose lock/unlock operation *may be* operating on the same mutex as those in the LU-pair.

As an example, in Listing 2.5, assume the mutex pointers a and b point to the same points-to set. When inspecting the "inner region", we find only one LU-pair, which obeys all Feasible-HTM-Pair conditions in Definition 2.5.4. Consequently, the lock usage on b in the inner region can be transformed to HTM. When inspecting the "outer region", however, we see conflicting LU-points, and hence the locking operations on a will not be transformed. The resulting transformed code is shown in Listing 2.6, which is correct.

This approach complicates hand-over-hand locking [94, 95], sometimes used in the concurrent linked-list traversal, shown in Listing 2.7. As before, assume all four LU-points have a non-empty intersection of their points-to sets. When inspecting the inner region, the LU-pair `b.Lock()` and `a.Unlock()` passes all tests in Definition 2.5.4. Hence, they will be, incorrectly, paired and transformed to use HTM, as shown in Listing 2.8. This transformation violates the programmer's intention. Subsequently, when visiting the outer region, condition (3) is violated, and hence the outer LU-pair will not be transformed. One could have discarded the transformation of the inner region when the conflict is visible in the enclosing region. However, we cannot distinguish this incorrect pairing from the correct pairing in the previous case. Our solution is to always apply the transformations on the candidates found in inner regions, and handle mismatches at runtime via HTM aborts iff executing on the fastpath. As mentioned at the end of § 2.5.2, a mismatch is easy to recognize at runtime by, first, making `FastLock()` store the address

```
1 a.Lock() //outer region start
2
3
4 b.Lock()  // inner region start
5 a.Unlock() // inner region end
6
7 b.Unlock() //outer region end
```

Listing 2.7: Hand-over-hand lock.

```
1 a.Lock()
2
3 l := OptiLock{}
4 l.FastLock(b)
5 l.FastUnlock(a)
6
7 b.Unlock()
```

Listing 2.8: HTMized.

```
1     a.Lock()
2         b.Lock()
3         b.Unlock()
4     a.Unlock()
```

Listing 2.9: Perfectly nested locks.

```
1     a.Lock()
2     b.Lock()
3     a.Unlock()
4     b.Unlock()
```

Listing 2.10: imperfectly nested locks.

of the `Mutex` used at the lock-point in a field in `OptiLock` and, second, checking whether the `Mutex` passed to `FastUnlock()` is the one present in `OptiLock`. The transactional abort is needed (and possible) only on the fastpath. Section 2.5.2 details the correctness of transforming nested locks into HTM via GOCC.

Interoperability of lock-nesting with HTM

Programs may use perfect nesting as shown in Listing 2.9 or imperfect nesting as shown in Listing 2.10 (aka hand-over-hand locks [94, 95]).

The following cases arise depending on the kind of lock nesting and whether or not the LU-pair is converted into HTM.

- Neither the inner nor the outer lock is transformed to HTM. This is always safe because the behavior is the same as the original code in both perfect and imperfect nesting.
- Both the inner and outer locks are transformed to HTM. If both inner and the outer locks use the fast path and the transaction commits successfully, it ensures

mutual exclusion and atomicity of both critical sections, and hence it is safe; it also ensures that the entire transaction did not conflict with any other concurrent operation. If both inner and outer locks use the respective slow paths, the behavior is the same as the locks being untransformed (previous case) and hence obviously safe. These two are true whether perfect or imperfect nesting. If one of the inner or outer HTM falls back to its slowpath, the behavior becomes the same as one of the following cases.

- **Perfectly nested locks**

  Only the inner lock is transformed to HTM. This is safe because all instructions inside the HTM will appear to execute atomically and mutually exclusive to the external observer. While performing the inner (and the only) transaction, if the inner lock gets acquired by another thread, the transaction will abort, and none of its state changes will be visible to the external world; the outer lock will continue to be held. The inner transaction may be retried or fall back on its slowpath. Falling back to the slowpath is the same as both inner and outer locks not being transformed.

  Only the outer lock is transformed to HTM. This is safe because the inner lock operation, including lock acquisition and release, will be done transactionally. If another thread acquires either the inner or the outer lock while the transaction is in flight, the outer transaction will be aborted, and the inner lock acquisition (if already done) will be rolled back, ensuring mutual exclusion and atomicity of the entire region and the availability of the inner and outer locks to others. If the transaction commits, mutual exclusion and atomicity of the entire region are assured.

- **Imperfectly nested locks**

  Only the inner lock is transformed to HTM. This is represented by transforming `b.Lock()` and `a.Unlock()` in Listing 2.10. If at runtime, fallback path is taken at `b.Lock()`, `a.Unlock()` will also use the fallback lock because our implementation recognizes mismatched mutexes from `FastLock()` to `FastUnlock()` on the same `OptiLock` object; hence, the behavior is as if we did no transformation; and hence it is functionally same as the original code. If at runtime the fastpath is taken at `b.Lock()`, at `a.Unlock()` we recognize a mismatched mutex `a` vs. `b` and abort the transaction. The abort rolls back all changes done between `b.Lock()` till `a.Unlock()` and falls back to the slowpath. As stated before, slowpath is always correct, and hence the entire behavior is the same as the original untransformed code.

  Only the outer lock is transformed to HTM. This is represented by transforming `a.Lock()` and `b.Unlock()` in Listing 2.10. If at runtime, the fallback path is taken at `a.Lock()`, `b.Unlock()` will also use the fallback lock because our implementation recognizes mismatched mutexes from `FastLock()` to `FastUnlock()` on the same `OptiLock` object; hence, the behavior is as if we did no transformation; and hence the behavior is functionally same as the original code. If at runtime, the fastpath is taken, at unlock time in `b.Unlock()` we recognize a mismatched mutex `a` vs. `b` and abort the transaction. The abort rolls back all changes done between `a.Lock()` till `b.Unlock()` including the inner `b.Lock()` and `a.Unlock()`, thus discarding all changes in the entire region. Subsequently, we'll always take the slowpath route due to mismatched mutexes, and hence the effect is the same as both inner and outer locks not being transformed, which is always safe.

It is straightforward to extend the argument to any depth of nesting using an inductive argument that assumes that N-level nesting works (because 2-level nesting works) and uses the same argument as above to prove that $N+1$-level nesting is also correct.

Critical Sections with Function Calls

When the critical section protected by a candidate LU-pair contains function calls, we need to extend the analysis beyond the current function. Conditions (1) and (2) in Definition 2.5.4 are local to R. Conditions (3) and (4) require inter-procedural analysis.

We need to ensure that the transitive-closure of all code regions protected by a candidate LU-pair, including the blocks reachable via function calls, neither contains any HTM-unfriendly instructions nor contains any LU-points whose points-to set may overlap with the points-to sets of L or U. Otherwise, we discard the candidate LU-pair.

To accomplish this, we first build a static call graph using rapid type analysis [96, 97]. Next, we precompute summary information for each function on its own without its transitive closure of function calls; the summary contains (a) the fit of the function for HTM based-execution (i.e., no HTM-unfriendly instructions), and (b) the union of all points-to sets of all LU-points in the function, denoted by $\mathcal{P}$.

For a candidate LU-pair meeting all the conditions in Definition 2.5.4 within the region R, we proceed to do inter-procedural analysis. Let F^* be the transitive closure of all the function calls invoked inside the critical section $C \subseteq R$ protected by a candidate LU-pair. LU-pair is discarded if (a) $\exists F \in F^* s.t.$ F's summary contains HTM-unfriendly instructions or (b) $\exists F \in F^* s.t.$ $\mathcal{P} \cup \big(\mathcal{M}(L) \cup \mathcal{M}(U)\big) \neq \phi$. The former is simply condition (4) expanded to all functions, and the latter is condition (4) expanded to all functions. We note that nested locks discussed in § 2.5.2 can be in different functions.

The `defer` Statement

The `defer` [51] statement in Go, introduced in § 2.2, needs special attention. Go defers the execution of functions prefixed with the `defer` keyword to the calling function's return point. The presence of `defer Unlock()` complicates our CFG-based dominance/post-dominance analysis. Deferred unlocks extend the critical sections till function exit points. Listing 2.11 shows a legal Go code, where the `defer m.Unlock()` appears even before the call to `m.Lock()`. Condition (2) in Feasible-HTM-Pair will treat this pair as an invalid candidate for transformation because the lock-point does not dominate the unlock-point.

We address this case by interpreting `defer m.Unlock()` in a CFG by (a) introducing a synthetic `m.Unlock()` statement at the end of each basic block that returns control from the function, and (b) discarding the presence of `m.Unlock()` in its original position during the analysis. This allows us to reuse the previously described dominance relationship. During transformation, however, GOCC simply replaces a `defer Unlock()` with a `defer FastUnlock()` in its original place, as shown in Listing 2.12.

Multiple `defer` calls are executed in a last-in-first-out (LIFO) order of encountering the `defer` statement at runtime. This complicates the dominance and post-dominance relationship; for simplicity, we discard any function that contains multiple `defer Unlock()` statements. We found none in the packages used in our evaluation.

Profiles to Filter Hot Critical Sections

Profiling is a built-in feature in Go, which takes callstack samples via timer [98] or hardware performance counter [99] interrupts. One can take CPU profiles of a go program either at launch time by simply passing a `-cpuprofile` flag or from an already running program, for a specified duration, by accessing an exposed profiling port.

Go profiles are in the `pprof` format.

```
1 func DeferExample() {
2
3   m := &sync.Mutex{}
4   defer m.Unlock()
5   m.Lock()
6   // critical section
7 }
```

Listing 2.11: defer Unlock.

```
1 func DeferExample() {
2   l := OptiLock{}
3   m := &sync.Mutex{}
4   defer l.FastUnlock(m)
5   l.FastLock(m)
6   // inside HTM
7 }
```

Listing 2.12: HTMized.

The pprof Go package [98] allows us to programmatically navigate the callstack samples presented as weighted call graphs, where the nodes represent functions and edges represent caller-callee relationships. The functions are annotated with their inclusive and exclusive execution times.

When profiles are available, we use them to filter the regions where negligible execution time is spent, even before applying the static analysis. In fact, this is the first filtering step we perform before making the aforementioned LU-pair identification. We treat any critical section (including the entry and exit) where the aggregated execution time is less than 1% of the total execution time as insignificant.

2.5.3 Transformer

Since our end product is a code patch, we perform the transformation on the AST form of the program. Go AST can be serialized into source code via Go `format` [100] package. To this end, the transformer maps the candidate set of LU-pair operations found during the SSA-based analysis phase (described in § 2.5.2) to AST nodes [101]. It then replaces the LU-pair operations with calls to `FastLock()/FastUnlock()` in `optiLib`. It also passes the original `Mutex` object as a pointer to the calls to `FastLock()/FastUnlock()` since the underlying lock object is necessary for lock elision (fastpath) as well as for slowpath. Figure 2.6 shows an example AST transformation. The transformation itself is mechanical but challenging. In the following sections, we discuss several Go features that pose special challenges in transforming the AST.

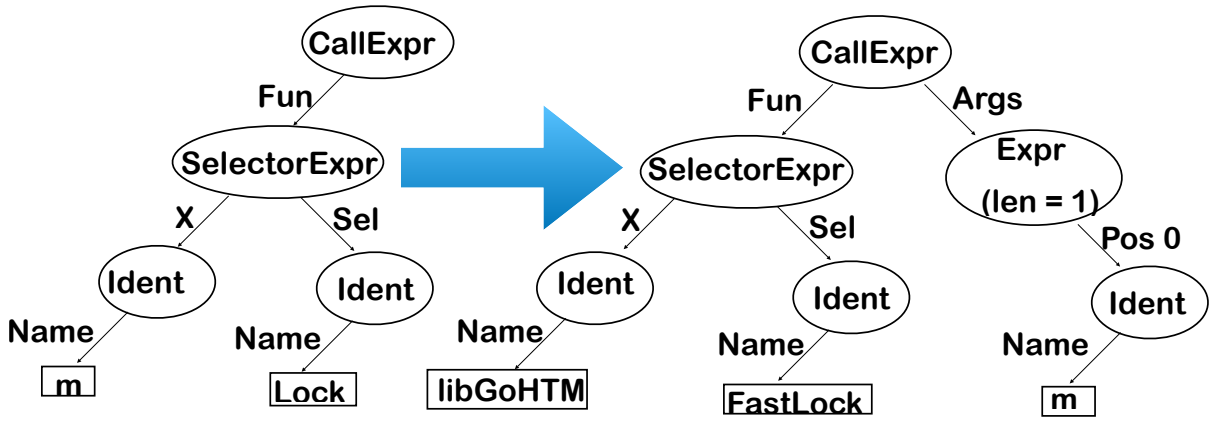


Figure 2.6: Example of AST transformation from `m.Lock()` to `optiLib.FastLock(m)`. Some AST nodes are omitted for brevity.

```
1
2 // pointer form
3 m := &sync.Mutex{}
4 m.Lock()
5 m.Unlock()
6
7 // value form
8 n := sync.Mutex{}
9 n.Lock()
10 n.Unlock()
```

Listing 2.13: Both Mutex pointer `m` and Mutex value `n` invoke Lock/Unlock using the same `dot` dereferencing operator.

```
1 l := Optilock{}
2 // pointer form
3 m := &sync.Mutex{}
4 l.FastLock(m)
5 l.FastUnlock(m)
6
7 // value form
8 n := sync.Mutex{}
9 l.FastLock(&n)
10 l.FastUnlock(&n)
```

Listing 2.14: GOCC transformation passes `m` as-is to FastLock()/FastUnlock() but `&n` to FastLock()/FastUnlock().

Go pointer vs. value: Go syntax does not distinguish accessing a field of a composite type (e.g., `struct`) via an object-pointer or an object-value; both use the same AST `dot` operator as exemplified in Listing 2.13. However, `FastLock()` and `FastUnlock()` must receive a pointer to the `Mutex` object to perform the elision correctly. Hence, if the LU-pair uses a `Mutex` value, its address needs to be passed to `FastLock()/FastUnlock()`, and if the LU-pair uses a `Mutex` pointer, it should be passed as is.

We address this issue in the transformer by querying the type information [102] for each receiver object subject to transformation. If the receiver identifier is a `Mutex` value type, we insert the additional `address-of` operator before the `Mutex` identifier in the AST and pass it to `FastLock()/FastUnlock()`. If the receiver identifier is a pointer to

```
1 type AStruct struct {
2   x int //not anonymous
3   *sync.Mutex //anonymous
4 }
5 func main{} {
6
7   a := AStruct{}
8   a.Lock()
9   a.Unlock()
10 }
```

Listing 2.15: Locking on an unnamed field of a struct.

```
1 type AStruct struct {
2   x int // not anonymous
3   *sync.Mutex //anonymous
4 }
5 func main{} {
6   l := OptiLock{}
7   a := AStruct{}
8   l.FastLock(a.Mutex)
9   l.FastUnlock(a.Mutex)
10 }
```

Listing 2.16: Unnamed mutex transformed to HTM.

a `Mutex` type, we pass it as is.

Go anonymous fields: Go allows programmers to define a struct that has fields without names. For instance, Listing 2.15 shows a struct `AStruct` that has an anonymous `*sync.Mutex` field. Operations on this anonymous mutex are performed by simply using the name of the enclosing struct variable as shown on Line 8. Hence, our transformation needs to be cognizant of whether the original LU-pair operations are performed on an anonymous mutex. By inspecting the type information [102] of the *access path* [103] used to invoke the lock/unlock operation in the AST, we determine whether or not the operation is performed on an anonymous mutex field. Upon determining the operations to be on an anonymous mutex, we pass the address of the anonymous `Mutex` to `optiLib` by simply suffixing the operation access path with `Mutex` as shown in Listing 2.16, Line 8 (where access path simply consists of variable `a`). This transformation composes of the previously described pointer vs. value operations.

Anonymous goroutines: Go supports anonymous goroutines [104], which are nested inside other functions as shown in Listing 2.17; these goroutines run concurrently. Our transformation introduces a new `OptiLock` variable in transformed functions. `OptiLock`'s declaration should be in the scope that encloses both Lock and Unlock operations, but it should not be shared by other concurrent executions because it maintains a goroutine-

```
1  m := &sync.Mutex{}
2  for i:=0;i<10;i++ {
3   go func() {
4
5      m.Lock()
6      // CS
7      m.Unlock()
8    }()
9   }
10 //wait all
```

Listing 2.17: Anonymous goroutines create concurrent units of execution on anonymous functions.

```
1  m := &sync.Mutex{}
2  for i:=0;i<10;i++ {
3  go func() {
4      l := OptiLock{}
5      l.FastLock(m)
6      // CS
7      l.FastUnlock(m))
8    }()
9   }
10 //wait all
```

Listing 2.18: The `OptiLock` needed for the transformation should be placed in the innermost function scope.

specific state. Hence, we make `OptiLock` a variable in the stack of each goroutine. We add the declaration to the innermost function body as shown on line 4 in Listing 2.18. A bottom-up AST walk from LU-pair to be transformed allows us to easily detect the innermost enclosing anonymous function scope.

2.5.4 Adaptive HTM Runtime: `optiLib`

`optiLib` implements all the intelligent runtime control needed to perform HTM in lieu of locks. It is in charge of starting and committing transactions in critical sections, as well as falling back to the lock when necessary. It is responsible for inter-operating with locking operations on the same mutex that may not be transformed to use HTM. In the event of aborts, it is responsible for determining the cause of the abort and deciding whether and how many times to retry. If, accidentally, the code rewriting matches lock-point with a programmer-unintended unlock-point, `optiLib` is responsible for detecting and recovering from the mistake. Finally, it is responsible for understanding and dynamically adjusting to changing contention.

We implement `optiLib` using TSX [60] for Intel platforms. `optiLib` is carefully implemented to ensure correctness under all circumstances. Equally important, it is implemented with the utmost attention to performance. Every instruction and its placement are carefully planned to minimize any overhead of its own. `optiLib` uses Intel

RTM; it does not use the Hardware Lock Elision (HLE) [61] because it does not provide the fine-grained control we need. `optiLib` introduces a data structure: `OptiLock`, which has two fields: a boolean `slowPath` and a `*sync.Mutex` `lkMutex`. `slowPath` is set to true if the lock is not elided at runtime. `lkMutex` always holds the address of the fine-grained lock being elided. `OptiLock` supports `FastLock()/FastUnlock()` operations, both need a `*sync.Mutex` argument, which is the mutex receiver object being elided at the original call sites of `Lock()/Unlock()`. Try locks and timed locks [105, 106] are absent in Go; it is trivial to support them in `optiLib`.

The `FastLock()` implementation uses sophisticated mechanisms described previously by others [82, 55, 83] to interoperate slow and fast paths concurrently. Stated succinctly, the `FastLock()`, waits for the fine-grained lock to be available before starting the hardware transaction; on starting a transaction, it first checks if the fine-grained lock is already held and unconditionally aborts if it is already held; if it is not held, the act of checking adds the lock word to the transaction read-set, and hence, if a concurrent execution on the slowpath acquires the same lock during the transaction, the fastpath immediately aborts ensuring mutual exclusion. Any two threads in the fastpath can run concurrently as long as they do not conflict with their memory accesses.

Reading the internals of the original `sync.Mutex` object is straightforward and has no performance penalty; `FastLock()` simply de-references the first word of the `Mutex` pointer passed into the function, which contains the lock status. The `FastUnlock()`, based on `slowPath` value, either commits the transaction or invokes the unlock on the mutex object. It also safeguards against accidental incorrect code patches by ensuring that the mutex object passed into `FastLock()` and stored in the `lkMutex` field of `OptiLock` matches the mutex object presented to `FastUnlock()`. In case of a mismatch, `FastUnlock()` restores safety by aborting the transaction (iff on fastpath), and subsequently enforces the slowpath.

Dynamic adjustment via perceptron

It would not be fruitful to attempt HTM if doing so has already proved to be unsuccessful for whatever reason. GOCC learns and adapts to HTM behavior and decides whether to use HTM for the already transformed LU-points, the fallback being the original lock. For this purpose, GOCC uses a featherlight, hardware-inspired "hashed perceptron" [43].

The hashed perceptron predictor hashes feature weights into one or more tables. Then at the prediction time, it uses indexes to access feature weights from the tables and adds up all the relevant weights. If the sum exceeds a threshold, the prediction will be regarded as positive (e.g., "HTM should be taken"). Otherwise, the result will be viewed as negative (e.g., "HTM should not be used"). The weights will be updated based on the correctness of the predictions. If operations on a given `Mutex` have been HTM-friendly/unfriendly, we want to utilize this information. Similarly, if a code location has been HTM-friendly/unfriendly irrespective of the `Mutex` used, we want to use this information as well. Hence, the two input features for the perceptron are the `Mutex` and the calling context [107, 44] of lock/unlock invocation. The address of the `Mutex` serves as the `Mutex` feature, and the address of `OptiLock` serves as a unique identifier for the calling context feature. Updating the same perceptron weight for the `Mutex` feature by different goroutines would create a conflict (and potentially a performance collapse). Hence, we instead XOR the `Mutex` address with the address of the `OptiLock` to produce a conflict-free feature input.

Our perceptron implementation creates two 4K-entry arrays as the global weight tables (`GWT`). The weights take an integer number ranging from -16 to 15. At runtime, `FastLock()` and `FastUnlock()` functions index into `GWT` by taking the lower-12 bits of the two features. Perceptron operations are done outside the transaction. The up-

repo	stars	contributors	commits	LoC	Lock points	Unlock points total (defer)	violates dominace	Candidate pairs	unfit for HTM intra/interproc	Nested alias locks intra/interproc	Transformed Pairs w/o profiles total (defer)	Transformed Pairs w/ profiles total (defer)
tally	450*	27	95	2.4k	54	56 (28)	2	52	2/29	0/0	21 (14)	7 (7)
zap	4.5k*	7	163	3.3k	8	8 (4)	0	8	3/0	0/0	5 (1)	6 (0)
go-cache	11.6k*	71	322	18k	96	230 (6)	68	28	0/2	0/0	26 (4)	10 (2)
fastcache	59k*	40	673	33k	24	24 (2)	2	22	2/2	0/0	18 (0)	7 (4)
set	967*	8	48	2.4k	16	16 (10)	0	16	0/2	0/0	14 (8)	8 (2)

Table 2.1: Go package characteristics and their behavior using GOCC

dates and reads from `GWT` are lock-free but racy — perfection is not required here, but high-performance is necessary. Experiment results from § 2.6.2 show the effectiveness of perceptron learning in protecting against poor HTM performance.

Perceptron weight update: Perceptron weight updates happen in the `FastUnlock()` function after successfully finishing the critical section, whether on fast or slow path.

If the perceptron decides that the lock should be used, there will be no update to the weights as the lock will always succeed. When the perceptron indicates to use the HTM and the execution finishes on the fastpath, the corresponding weight in the cell will be increased (because the perceptron makes a correct decision, it should be encouraged to use the HTM more frequently). On the other hand, if perceptron determines to use HTM, but HTM fails and falls back to slowpath, the weights will be decreased (because HTM does not work for the current call, perceptrons should be penalized for incorrect recommendations to improve future predictions).

Weight decay: We keep a counter, in each cell in `GWT`, to record the number of lock calls that go to the slowpath directly as a result of perceptron decision. If a lock has been used consecutively for a certain number of times and exceeds the threshold, we reset the weight of the perceptron cell and subsequently try HTM. Without this reset, perceptron would get stuck on the slowpath, preventing the benefits of the HTM execution in the future. We set this threshold to 1000 continuous decisions. Section 2.5.4 summarizes our `FastLock()/FastUnlock()` implementations including the perceptron logic.

```
func (ml *OptiLock) FastLock(l *sync.Mutex) {
    ml.lkMutex = l
    compute indices and fetch perceptron weights
    if lockCounter > threshold {
        reset  perceptron weights
    }
    if perceptron decision == HTM {
        trial := MAX_ATTEMPTS
        for {
            if trial > 0 {
                if lock already held {
                    spin with pause till lock held
                }
                status := TxBegin()
                if status == Txstarted {
                    if lock is held {
                        abort LockHeldError
                    }
                    return
                } else {
                    trial--
                    if (aborted for LockHeldError) {
                        continue // retry
                    }
                    if ( ! TxAbortRetry) {
                        // includes MutexMismatchError aborts
                        slowpath := true
                        htm fails := true
                        take original lock
                        break
                    }
                    // retry
                }
            }
        }
    } else {
        lockCounter++
        slowpath = true
        take original lock
    }
}

func (ml *OptiLock) FastUnlock(l *sync.Mutex) {
    if slowPath {
        l.Unlock()
        if htm fails == true {
            compute indices decrease perceptron weights
        }
    } else {
        if l != ml.lkMutex {
            abort MutexMismatchError
        }
        TxCommit()
        compute indices and increase perceptron weights
        lockCounter = 0
    }
}
```

Listing 2.19: Pseudo-code of FastLock() and FastUnlock()

Algorithm of optiLock

Listing 2.19 provides a sketch of the algorithm we use in `optiLib`.

Alleviating HTM overhead

HTM brings overhead for very short critical sections as described in § 2.2 above, even under single-core execution. `optiLib` avoids using HTM if it recognizes a single OS-thread in a Go process. `optiLib` employs `runtime.GOMAXPROCS(0)` API for this purpose.

2.6 Gocc Evaluation

We evaluate Gocc on an 8-core (×2-way SMT [108]) Intel Coffee Lake CPU with a total 32GB memory, running Linux `5.4.0`. The CPU has 32KB L1I and L1D cache, 256KB L2 cache, and 16MB L3 cache. The Go version is 1.15.2.

Table 2.1 shows the list of applications and libraries we employ. In the absence of standard benchmarking for Go, we selected packages that are popular open-source Go projects (column 2 in Table 2.1), focus on high performance, utilize lock-based Go concurrency, and provide thread-safe APIs. In particular, Zap and Tally are foundational logging and metrics collection packages used in production go programs by many organizations. Additionally, since we evaluate the projects using their own benchmark suites (more on this below), we only selected projects that feature concurrent benchmarks or whose benchmarks could be straightforwardly converted to be concurrent.

From a static analysis viewpoint, we see that all applications contain several locks. Defer unlocks are common (column 7). The "violates dominance" column shows how many LU-points were discarded since they did not meet the dominance relationship. The number is typically low except for `go-cache`, which has several functions with a repeating pattern of unlocks that do not post-dominate the candidate lock. The "candidate pairs" column shows how many LU-points remain for further analysis. Each column to the right progressively shows the reasons for which a candidate LU-pair was rejected. Rejection

due to nested aliased locks is not found in these packages. The second-to-last column shows how many LU-pairs were finally rewritten to use `OptiLock`, including how many of them contain `defer Unlock()`. The last column shows the numbers after we retain only those locks where the functions contain at least 1% of execution time in execution profiles. Overall, GOCC transforms several LU-pairs in each application. Using profiles significantly reduces the number of transformed LU-pairs.

We run all the benchmarks within each repository five times and report the median. We believe the benchmarks accompanying the code best represent its desired characteristics. As some benchmarks are written for a single thread setting, we rewrite them to introduce concurrency to utilize HTM-enabled parallelism fully. We adopt the standard testing package from Go[109], which runs each benchmark for a certain amount of time and reports the throughput as nanoseconds per operation. We wrap the benchmark codes with RunParallel [110] helper function to get parallel performance if it was not already done so. Using more CPU cores, ideally, increases throughput (i.e., reduces average nanoseconds per operation). Then we compare the throughput from locks vs. HTM — a positive percentage means GOCC's rewrite did better, and a negative percentage means the baseline did better. We vary the number of CPU cores available for benchmarks from 1 to 8. Unlike HPC codes that run on all cores on a server, Go services often use 2-4 cores.

2.6.1 Results on Popular Go Programs

We categorize the benchmarks in each package into two groups:

1. **Concurrency non-sensitive** benchmarks either have no locks or do not spend much time in critical sections, or our transformation does not result in any performance difference. For these benchmarks, we only show the aggregate (geomean) results

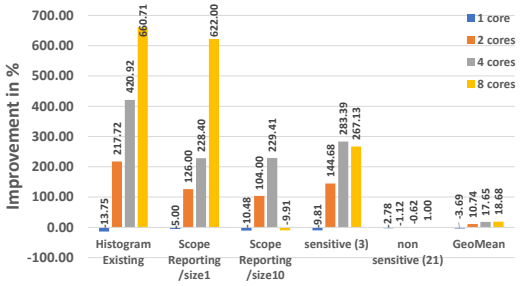


Figure 2.7: Results on Tally with different core numbers.

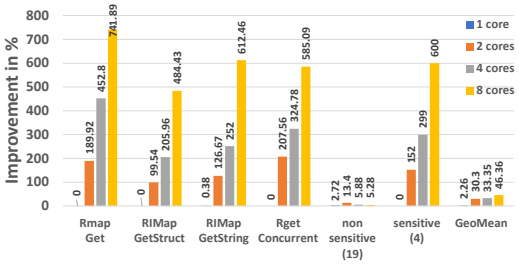


Figure 2.8: Results on go-cache with different core numbers. (benchmark names reflect abbreviated names of go-cache's benchmark functions).

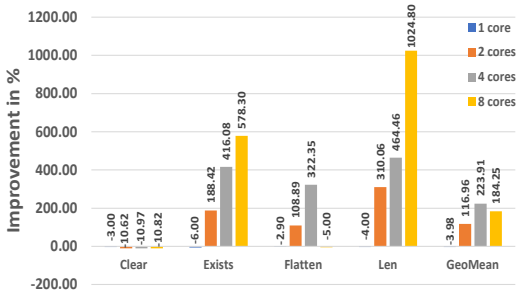


Figure 2.9: Results on concurrent set with different core numbers.

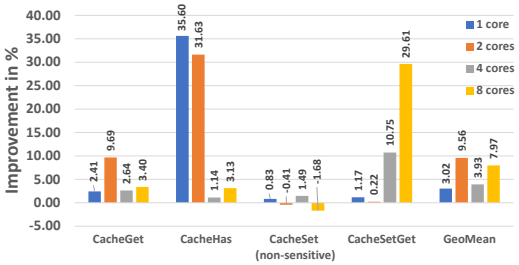


Figure 2.10: Results on fastcache with different core numbers.

unless noted otherwise. They appear as "non sensitive" in our charts, and the number in the parenthesis indicates how many benchmarks are in this group.

2. **Concurrency sensitive** benchmarks exercise modified locks non-trivially. We might have impacted them positively or negatively. For these, we present data from each benchmark and also present an aggregate result ("sensitive" in our charts).

The "all" part of our charts is the geomean taken over all benchmarks. Sometimes this number looks small because of a large number of non-sensitive benchmarks.

In what follows, we provide the details of performance evaluation on the aforementioned Go packages. The total number of benchmarks is large; hence, we dive deeply only into benchmarks with large speedups. **Tally** [4] is a fast, buffered stats collection library and Figure 2.7 shows its results. For the `HistogramExisiting` benchmark, GOCC

achieves more than 660% speedup on 8 cores reducing the original time per operation from 65 ns/ops down to around 8.47 ns/ops at 8 cores. Moreover, the HTM delivers scalable performance. This benchmark uses a `Mutex` lock on a read-only `Exists` operation, and hence, is a natural candidate to demonstrate speedup as HTM eliminates unnecessary serialization. Conversely, the baseline has a scalability collapse, where the time per operation increases from 20.4 ns/ops to 65 ns/ops for 1 to 8 cores. `ScopeReporting1` holds three independent `RWMutex`es at different points in time and accesses read-only data. However, since the `RWMutex` also involves a counter increment and a decrement, its overhead as a result of cache invalidation does not scale well. Thus, even eliding `RWMutex` proves highly beneficial. The speedup for `ScopeReporting10` is lower than that for `ScopeReporting1` because it performs 10x more work inside the critical section. Overall, in the sensitive group, we see a 10% performance drop with a single CPU but 145%, 283%, and 267% improvements with 2, 4, and 8 CPUs, respectively. In the non-sensitive group, the overall performance drop is within the margin of error. Among all the 27 benchmarks of tally, we see up to 18.7% speedup at 8-CPUs.

go-cache [111] is an in-memory key-value store. It contains benchmarks that exercise repeatedly accessing the same item in a small map. The benchmarks contain both non-cached accesses, similar to how Go programmers often use a map and cached accesses provided by the `go-cache` layer to demonstrate the effectiveness of the library. All benchmarks employ `RWMutex`s for concurrent map read access. Unlike the rest of the use cases, the benchmark files themselves contain locks, which GOCC transforms into using HTM.

Figure 2.8 shows our empirical results. GOCC speeds up four benchmarks in `go-cache` that were directly accessing the map without the library-provided cache. In each case, we can see more than 100% speedup; the biggest speedup is 742%. The speedups come from eliminating contended atomic operations involved in entering and exiting from a reader

lock. The performance scales well with increased parallelism because while the lock-based approach incurs more and more contention, the HTM approach remains conflict-free throughout. The other benchmarks, the majority of which employ the `go-cache`, are mildly improved, but more importantly, they were not degraded as a result of transformation via GOCC.

go-datastructures [112] is a collection of performant, thread-safe data structures. We apply GOCC on the `set` subdirectory, which contains concurrency benchmarks. The results are shown in Figure 2.9. The `Len` benchmark computes the length of the set, and it is sped up by ∼1000% in the 8 cores setting. `Len` has a short critical section that has a higher entry and exit cost due to atomic operations when using a`RWMutex`. HTM performance shows scalability since the HTM version remains conflict-free, whereas the lock-based version collapses with increased contention. The `Exists` benchmark is similar to `Len`, where each goroutine searches one item in a set containing only one item. It scales almost as well as `Len`, but more work is done in the critical section, which amortizes `RWMutex`'s overhead, and slightly reduces HTM's advantage. The `Flatten` benchmark reads 50 elements from a shared map into a private array, with a layer of caching that eliminates repeated map scanning. It holds a `Mutex` to serialize concurrent accesses to the map/cache. The HTM version avoids serialization and shows scalable performance for 1-4 cores. At 8 cores, the number of conflicts resulting from updating the cache rises, which makes perceptron not use the HTM, and hence there is no speedup. The `Clear` benchmark has true conflicts, and there is no speedup, but the HTM does not significantly degrade the performance. Overall, utilizing GOCC results in more than 100% geomean performance gain while introducing less than 4% slowdown in a single core setting.

Zap [113] is a library that implements fast and structured logging in Go. Being a logging library, it has several IO operations, and hence GOCC rewrote fewer locks. Compared with other repositories, the improvement on zap is relatively mild. Due to

arguably mild speedups on Zap, a large number of benchmarks, we omit a deeper analysis of Zap results. Slowdowns are rare, the biggest being 7%. Overall, we observed a mild $\sim 4\%$ geometric mean speedup with the best case 28% speedup.

Fastcache [114] is a fast, scalable, in-memory cache. The transformed code delivers a maximum of 35.60% speedup and a geomean of 15.65% speedup across all benchmarks.

In the `CacheGet` benchmark, goroutines repeatedly invoke the `Get` function, which uses an `RWMutex` to protect a shared map. `Get` has inter-procedural nested but non-conflicting locks, all of which are transformed into HTM. `Get` looks up a key in the map and returns a value blob. The critical section of `Get` contains a few atomic add instructions, which update shared variables. Transactional conflicts on the shared atomic adds are fewer at low core numbers, and the speedup is visible; however, at larger core counts, the conflicts increase, and the speedup vanishes. Fortunately, the perceptron kicks in and avoids any performance collapse.

The `CacheHas` benchmark is virtually the same as `CacheGet`, but its critical section is shorter since it does not return a populated value buffer. Hence, the speedups are higher due to fewer conflicts, but it follows the same performance pattern as `CacheGet`.

In the `CacheSetGet` benchmark, each goroutine has two loops: the first loop repeatedly invokes `Set` and the second loop repeatedly invokes `Get`. The `Set` function, which inserts a key-value pair into the map, may raise a `panic` if certain constraints are violated. Hence, GOCC does not modify a `Lock()` present in `Set`. The `Get` function is already described previously.

Since all goroutines first attempt `Set`, where Go's default locks are being used, the runtime recognizes it as a starved mutex and takes away the time slice of some of the goroutines. This runtime behavior reduces the number of lock contenders and, as a result, a few goroutines monopolize the lock. These goroutines quickly finish their series of `Set` operations and proceed into calling `Get` in a loop. The contention is lower on `Get` also

since the load is now split between `Get` and `Set` with some goroutines on hold. The net effect is a high throughput for the whole benchmark.

It is worth noting that the only other benchmark which invokes the `Set` function is the non-sensitive benchmark `CacheSet`. Even though `CacheSet` exhibits no performance improvement, and `CacheGet` shows mild performance improvement, their composition in `CacheSetGet` leads to secondary effects causing much higher performance gain at higher core counts.

2.6.2 Perceptron Evaluation

We assess the effectiveness of perceptron using the `Tally` benchmarks. We compare the performance with and without the perceptron machinery. In the absence of the perceptron, we always attempt HTM. In the results presented in Figure 2.11, we can observe that the perceptron is effective in eliminating any performance loss. For example, `CounterAllocation` and `SanitizedCounterAllocation` are HTM-unfriendly benchmarks and cause aborts frequently. Perceptron quickly learns to move away from HTM and keeps using the slowpath. Therefore, there is a minimal performance loss for the perceptron case.

Finally, we set up a synthetic benchmark — a conflict-free critical section with 1000 counter updates — to evaluate the overhead of the perceptron machinery. We measured the perceptron prediction overhead to be 0.65% and weight update overhead to be 0.73% for a total of only 1.38%.

2.7 Conclusions

GOCC is a source-to-source transformation tool to speed up lock-based pessimistic concurrency control in Go programs with Hardware Transactional Memory. GOCC com-

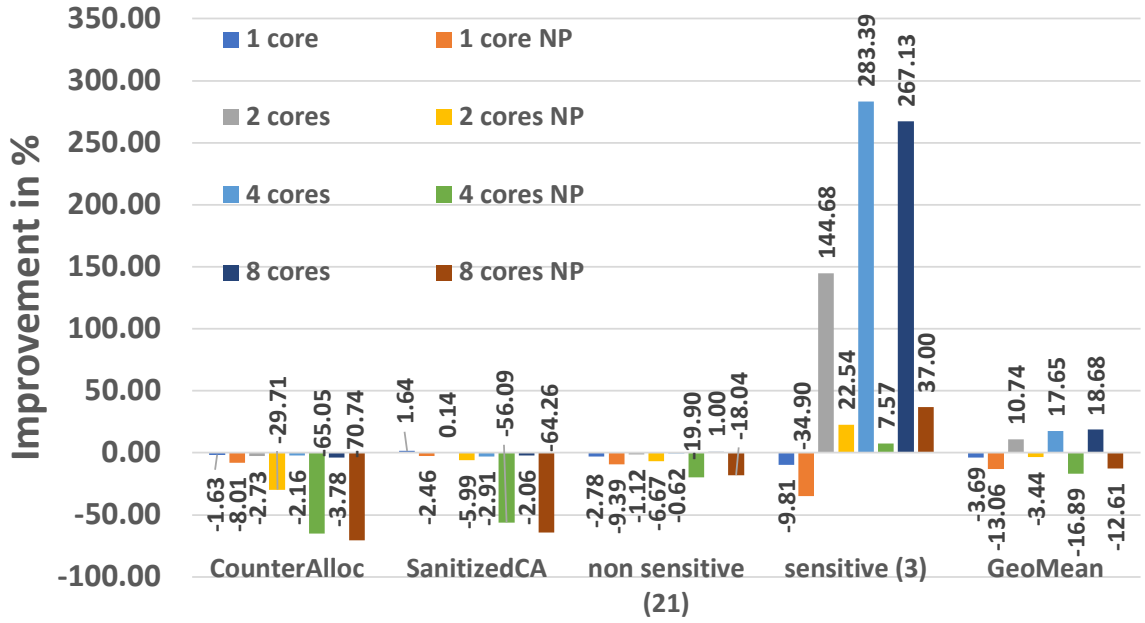


Figure 2.11: Results on `Tally` to show the effectiveness of perceptron. NP $\Longrightarrow$ No Perceptron.

bines thorough static analysis with intelligent runtime control to expose additional parallelism available in Go programs. GOCC keeps the developer in the loop, minimizes code changes via execution profiles, and targets only those critical sections that are likely to improve with HTM. The experimental results from real-world Go packages show that GOCC delivers significant (up to 10×), scalable performance for concurrent Go code that uses locks while exhibiting rare and relatively small slowdowns.

The reason GOCC works is because it focuses on the communication paradigm, i.e. concurrency control, between different goroutines. To achieve a performance improvement, GOCC includes performance-driven analysis and components like the hash-based perceptron runtime controller. The use of perceptron also inspires us with broader applications and the design of Prediction System Service (PSS) in the next chapter. Also, the use between lock and TM can be generalized into slow-path fast-path paradigms that will be covered in the following chapter. We believe other synchronization mechanisms

like channels can be also studied for potential benefits in performance in the future.

Chapter 3

A Prediction System Service

3.1 Introduction

When the low-hanging fruit of obviously inefficient implementation has been stripped away and the performance of an application is still a critical concern, capable performance programmers often find themselves attempting to navigate a complex set of trade-offs. When is it faster to just lock this data structure versus wrapping it optimistically in a transaction? When should I just execute the unoptimized version of this function versus investing the time to make it faster? When should the operating system pull this resource so it can go to a better use somewhere else? Answering such questions is a matter of balancing a set of conflicting forces. As our applications, systems, and hardware grow increasingly complex, it is hard to understand (or even characterize) all of the forces relevant to good decision making – and even more difficult to navigate those forces with simple ad-hoc heuristics.

Of course, the fact that *machine learning* has proved particularly capable of navigating exactly this type of complex optimization space is not something that has been lost on application researchers. Machine learning techniques have been demonstrated

for optimizing data structures [18], implementing state-of-the-art recommendation systems [19], improving anomaly detection [21], and learning the structure and optimal access of databases [20]. In the case of TVM [22], the optimising compiler could produce machine learning kernels that beat human optimisers', leading to significant performance improvements.

This style of optimization is bound to become increasingly common in the coming years and it makes little sense for each and every application to roll out their own internal embedded ML framework for dynamically controlling a few parameters. Such an approach requires each application to support their own machine learning code base and elides opportunities for sharing of memory or exploiting hardware resources. Instead, it is time to consider the question of what new *abstractions* are necessary to lower the barrier to entry and sustainably support this important style of optimization.

While the process of learning a good response from noisy examples is well covered in the machine learning literature, actually deploying the ability to make predictions in a manner useful for software optimization requires some innovation. Because these predictions are often (by the nature of targeting performance-critical code) directly on the critical path, their utility is a function of both their accuracy *and* their latency. A prediction service must be both cheap (in terms of computational overhead) and it must be good enough (providing enough performance benefit to be comparable to or better than a hand-tuned approach).

In this work, we argue that it is possible and worthwhile to introduce a common, simple, shared prediction mechanism to a variety of runtime tasks and that the right location for this mechanism is as a system service. A system-wide prediction service can operate usefully with as few as two API functions and can be made to both be easily reusable across the software stack and allow for additional innovation on both sides of the interface. By functioning as a service, the operating system can enable sharing of

training information across user applications when desired or restrict usage according to system policy. The service can also be provided within the kernel for use by runtime services that otherwise rely on domain-specific heuristics to make performance decisions.

To explore the potential of a PSS to enable optimization, we prototype this change to a full operating system and examine, both qualitatively and quantitatively, the capability of our nascent service to ease optimization across three different scenarios calling back to the questions at the beginning of the introduction: transactional lock elision, JIT parameter tuning, and page reclaim. These scenarios exercise the interface in user and system mode, across multiple languages, and in both aiding online decisions and parameter tuning settings. Specifically, Transactional Lock Elision [115] via Hardware Transactional Memory (HTM) is a classic example of a fastpath-slowpath heuristic employed by software; we will show how our predictor guides this decision on when to use HTM (fastpath) and when to fallback to locks (slowpath). Just-in-Time (JIT) compilation always has a tension between high compile time if highly optimized and low code quality if not well optimized while offering a whole search space of possible optimization parameters; we will illustrate a way to employ PSS to quickly arrive at optimization parameters that improve program speed as compared to the existing parameter tuning solution provided by the human-optimised PyPy runtime. Page reclaim in the Linux kernel under high congestion relies on a careful, heuristic-driven consideration of memory usage and storage device utilisation in order to maintain global performance; we show that introducing PSS to the kernel can significantly outperform human-optimised heuristics developed within the last year. Specifically we:

- Introduce the novel concept of prediction as a system service
- Demonstrate that an exceedingly simple interface providing only `predict`, `update`, and `reset` is all that is required to be useful for software optimization.

- Develop a complete proof-of-concept implementation capable of providing Linux processes with useful and actionable predictions in 4.19 ns.

- Evaluate the effect of these prediction-driven optimizations across a variety of both user and kernel mode applications and demonstrate the resulting system performance improves Transactional Lock Elision by 34% on average, PyPy JIT parameter tuning by 15% on average over microbenchmarks and 12% over macrobenchmarks, and provides a 33% average latency reduction for page reclaim.

We begin with more description of the concepts and requirements of Prediction as a Service in Section 3.2 followed by details of our prototype implementation and the reasoning behind our latency-optimized software architecture in Section 3.3. Section 3.4 describes the application use scenarios in detail and is followed by a more detailed quantitative evaluation, related work, and conclusions in Sections 4.7.3, 4.8, and 3.7 respectively.

3.2 Prediction as a Service

If one were to take a careful catalog of all of the performance optimization techniques available, there is no question that a particularly large chapter would be required for those driven by prediction. Operating systems can predict the next set of disk pages required by applications and speculatively bring them into main memory [116, 117]. Memory access patterns of CPU cores or OS threads can be learned and the OS can automatically migrate page frames from a remote NUMA socket to a local socket to reduce latency [118]. Lock implementations can have a spin-and-then-block [119, 120] logic which spins for a set time before falling back to heavyweight OS-facilitated blocking. Transactional memory [7] (both in software and hardware) can dynamically and speculatively adjust to observed contention [121, 122, 123].

While these techniques rely on a prediction, most are not *explicit* about the predictive nature of their ability to achieve a speedup. Instead, most hide their predictive nature in the choice of a parameter or in a set of criteria used to make a selection. Unfortunately: **1)** *Parameter choices are often ad-hoc*, relying on limited use cases and/or hard-won domain-specific expertise making such approaches fragile and hard to scale. **2)** *Even when well informed by data, most parameter choices are still static*, meaning they are unable to adapt to the changing machine state or objectives. Profiles can help gather information on effective parameters, but profiling requires either well-understood use-cases or the ability to gather useful information in production with low overhead. Both of these are possible, but **3)** *Complex dynamic approaches for either prediction or profiling increase application complexity* which, in turn, makes the system harder to support across multiple platforms and increases the code footprint significantly. Finally **4)** *There is no effective way to share developments.* Programmers can spend non-trivial amounts of time optimizing the code in one specific language given a predetermined interface, but as we cross languages, as we have collections of smaller services, and as we seek to exploit hardware to help in the process, there is little opportunity for reuse.

In contrast, an ideal system would be **straightforward** to understand and simple to use. Users should only need to specify a target function, candidate solutions, and feedback. A prediction service would generate a prediction (informing, for example, which equivalent code path take) and update the model based on feedback. To be effective the prediction service must be **low overhead** both in terms of training and inference. The service will need to provide useful predictions as early as possible, to avoid long warm-up overheads, and provide those predictions with very low latency, to avoid eating into all of the potential performance improvements such predictions might provide. The prediction service should also be suitably **general purpose**, allowing it to be applied to a wide range of applications, possibly written in multiple programming languages. It should not

only work with one or a few domain specific scenarios.

An effective target for such an optimization, in turn, needs to be both *measurable* (meaning that it is possible to determine the "goodness" of the prediction to inform learning) and *correctness preserving* (meaning paths under all possible predictions are equally correct even if not equally desirable).

3.3 Design and Implementation of a PSS

Informed by the requirements above, a Prediction System Service (PSS) provides a standard interface and straightforward prediction and update procedures. At a high level, the PSS takes input of the programmer's choice and returns the value of the prediction. In our proof of concept we limit ourselves to predictions along a single dimension where the return values can be interpreted as "predict true" when positive and "predict false" when negative and the magnitude of the return value shares some degree of confidence in the prediction (particularly useful when the costs of mispredictions are asymmetric or when true and false are used iteratively to narrow in on some balance point). The system then attempts to optimize its predictions over time based on feedback in the form of updates.

3.3.1 System Interface

PSS can be implemented with two core functions, `predict` and `update`, and one state management function, `reset`, with behavior as follows:

Predict: Given input features and stored weights, `predict` generates a binary result prediction which determines which path to take. The format of the input features can

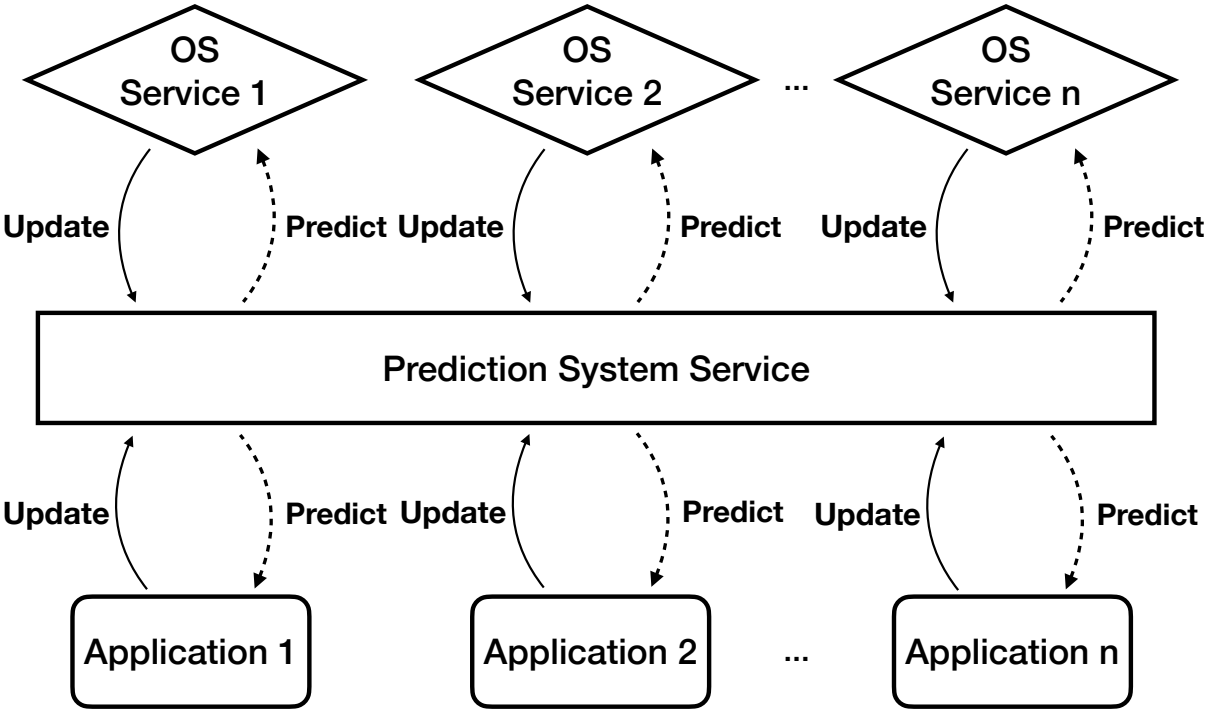


Figure 3.1: Design of Prediction System Service

be different depending on the prediction scenario. The function signature is

```
int predict(int* features, int len)
```

where the input is an array of user-specified features for predict with length of `len` and the returned prediction value is an integer. The number and value of features can be changed by users for different scenarios.

Update: Based on the predicted and observed results, PSS will `update` the stored model parameters accordingly depending on whether the prediction was correct or not. The function signature can be viewed as

```
void update(int* features, int len, bool dir)
```

where the input parameter contains a feature array and its length like `predict` and one Boolean variable to indicate whether the prediction is correct or not.

Reset: This function allows the users to initialize the stored PSS data, either by section or in totality. It can be called if certain environment parameters of the prediction have been changed or to completely wipe the PSS data. As an example, when some data need to be reused without initializing all data used by PSS for prediction, we can use this function to selectively clear only some data. The function signature can be viewed as

```
void reset(int* features, int len, bool all)
```

where the input feature array and length are similar to the previous two cases and there is an additional boolean variable to indicate whether to wipe out the entire PSS data or clean a specific entry.

3.3.2 Prediction Unit Design

While there are many possible implementations of a PSS, for our proof of concept implementation we wanted to pick a design that we knew would have consistently low latency and that would help us test our hypothesis that even relatively simple predictions would be an important step beyond the state of the art in many potential optimizations. As such, for this effort, we limit our evaluation to an online perceptron predictor [42]. Given an input feature vector, the predictor simply calculates the weighted sum of the input and compares it with a threshold value. If the sum surpasses the threshold, the result will be regarded as positive, otherwise the return value will be negative. During `update`, the prediction from the perceptron is compared to the actual outcome. If the prediction is correct, the weight will be increased. Otherwise, the weight will be decreased

as a penalty.

The hash-based perceptron predictor has been proven to be highly versatile yet can be both executed and updated in very short order (in either software or hardware) [124, 125, 126]. While more sophisticated predictor designs are possible to consider with hardware support in the future, we prioritize low latency software implementations in this work.

Currently, PSS is designed to support up to 16 features with 1024 entries for each feature. Each feature can be a parameter that potentially has an impact on the target function and it can have up to 1024 values for each individual feature. The feature data is hashed to reduce the chance of conflict with other features and stored in a weight matrix. Once the predicted result is obtained, it can be compared with a threshold (like zero) to generate binary decisions. If the value surpasses the threshold, the prediction will be regarded as true, otherwise the prediction is considered false.

Predictor Model Extensibility

Since the system interface is not tied to the implementation, the underlying predictor model can be replaced easily if the users have specific needs. When low latency is preferred, other relatively simple models can be used, such as decision trees [127], linear regression [128], and naive Bayes algorithm [129]. On the other hand, if accuracy is prioritized more complicated model can be deployed, including XGBoost [130], k-nearest neighbors (KNN) [131], and neural networks [132].

Parameter Types

The API described above mainly focuses on numeric parameters like the length of the array. But PSS can accept categorical parameter types like the type of the color after some preprocessing or transformation. For example, if those categories exist in some sort of embedded space then they can be exposed to a predictor through hierarchy or

projection.

3.3.3 Reduced Latency Predictions with vDSO

In modern Operating Systems (OS), a user-level application cannot touch the kernel's memory space directly for a whole variety of reasons. Instead, interactions with the kernel are typically supported by system calls (syscall) — unfortunately, syscalls carry with them a significant amount of context switch overhead which conflicts directly with our stated goal of achieving low latency.

Fortunately, we are not the first to grapple with such a limitation and there are now multiple different mechanisms to build from, most notably virtual system calls (vsyscall) and virtual dynamic shared objects (vDSO). A vDSO is a Linux kernel mechanism that allows a portion of kernel memory space to be accessible in user space via a small shared library. The system presents to user space a map to the corresponding kernel data and programs such that it can access that memory directly. This facilitated direct read-out means there is no context switch involved in satisfying a vDSO read request which, in turn, leads to significant speedups [133]. In our experiments, this reduces the latency by more than a factor of 16x (from 68ns with syscall down to 4.19 ns) and, even more importantly, translates to real and noticeable improvements in application runtime.

Of course, vDSOs have their own limitations. By definition, it can be only used in a read-only manner since user mode cannot modify the kernel memory without a syscall and we must provide data as part of the update process.

Therefore, we design PSS in a way that combines a mix of syscalls and vDSOs. Specifically, we implement `predict` via vDSO since no writing to kernel data is involved. For `update`, we choose a syscall as the means to modify PSS model within kernel space. To further reduce the syscall overhead from `update` calls, we adopt a batch update

mechanism that pools together multiple `update` calls into a single system call. A local buffer aggregates updates and allows us to amortize the boundary crossing. In addition, asynchronous syscall can be also explored to further reduce the overhead.

Advantages of a System Service One of the most interesting aspects of a system-service approach to prediction is that learning can happen across application invocations, a feature we demonstrate in application studies. While this is technically possible in application space, it requires the system to save and restore application-level files which is a poor match for the model of increasingly short-lived processes called in reaction to dynamic events. A system library has the additional advantage of being able to be used across kernel-space applications. Lastly, by utilising a vDSO that connects to kernel space, system policy can be enforced around the use of PSS, for example, to restrict which users or which programs can use the service and how information is shared across those programs.

3.4 Use-Case Scenarios

To demonstrate the usefulness of the services described above, in this section, we present the application of PSS in three different scenarios chosen to demonstrate the generality of the service.

3.4.1 PSS in Hardware Lock Elision

Synchronizing accesses to shared variables is a critical performance limiter in shared-memory multicore systems. While locks are one of the most frequently used mechanisms to safely manage sharing among many threads, locking is an inherently pessimistic method of synchronization where execution is potentially serialized and locking and un-

```
1 void TxLock(mutex *m) {
2   int * features = {perf_cnt, remain_retry} // 2 features
3   if (predict(features, /*len=*/ 2) == USE_HTM) {
4     tryingHTM = true
5     while(m->isLocked()) ; // spin
6     slowPath = false;
7     for (int i := 0; i < MAX_RETRIES; i++) {
8       if (tx_begin() == SUCCESS) {
9         if (m->isLocked()) {
10            tx_abort(); //abort
11        }
12        // transaction started
13        return;
14      }
15    }
16  } else {
17    tryingHTM = false
18  }
19  slowPath = true;
20  m->lock(); // slow path
21 }
22 void TxUnlock(mutex *m) {
23  int * features = {perf_cnt, remain_retry} // 2 features
24  if (!slowPath) {
25    tx_commit();
26    update(features,/*len*/2, /*reward*/+1)
27  } else {
28    m->unlock();
29    if (tryingHTM)
30      update(features,/*len*/2, /*reward*/-1)
31  }
32 }
```

Listing 3.1: Hardware Lock Elision with PSS.

locking costs are paid whether or not concurrent executions conflict in accessing data. In contrast, Transactional Memory [7, 54, 134, 135, 136] (TM) allows threads to execute through a set of guarded transactions optimistically and relies instead on the run-time detection of conflicts with an accompanying roll back when serialization is determined to be required. If one wishes to keep to the semantics of critical sections assumed by locks, TM can still be useful in allowing the system to speculatively execute through lock-protected critical sections through a class of techniques known as Hardware Lock Elision [137, 59, 138].

Of course, there is a balance to be struck between optimism and pessimism. Each lock under different use scenarios may benefit from a different approach and it is not straightforward to achieve good performance in practice due to the high costs of both rollback and of overly pessimistic locking.

Listing 3.1 presents a typical method for eliding locks using HTM. The original code is shown with a white background color and the additions we made to patch with PSS are highlighted in a gray background. There are two functions at the heart of the eliding lock implementation: `TxLock`, which is called at the beginning of a critical section, and `TxUnlock`, which is called at the end of the critical section. The input to both functions is a mutex object with lock/unlock/isLocked operations on it that can potentially be replaced by HTM. For interoperability with the lock, the HTM path is not tried until the lock is held (Line 5).

The transaction starts at Line 8 and the lock status is checked again to ensure it was not taken in the meantime by another thread and an explicit abort (Line 10) is issued if that is the case. The successful start of the transaction (`tx_begin()`) will result in a return from the function and will allow execution to continue into the critical section. Any failure due to conflict, capacity, explicit abort, or unsupported instruction, will cause the `tx_begin()` to return a non-success return code. On failure, retries are attempted a fixed number of times after which the algorithm falls back to the slow path of taking the underlying lock at Line 19. A special flag `slowPath` is set to indicate the corresponding action at the end of the critical section. This design performs well when most transactions succeed. However, in reality, it may not be known ahead of time whether lock-elision for a critical section is beneficial. For various reasons, the transactions can fail: notably due to increased contention and increased conflicts, due to increased memory footprint that may not fit within the HTM implementation's capacity, or due to the execution path using unsupported instructions.

Listing 3.1 shows the minor modifications (gray background) to the baseline that are required to enable PSS to guide the HTM vs. lock decision at runtime. At a high level, the idea is to utilize HTM if it is likely to succeed and rollback to lock if the transaction is likely to fail. Instead of having a fixed trial number mechanism, PSS allows the system

to easily make the lock/HTM decisions at runtime. The output of the prediction directly informs the path through the code taken. In order to make reasonable predictions from the PSS, we use two parameters. The first is a thread-level performance counter from past transactions. We use an integer to store the past performance and each bit represents one transaction attempt. A value of '1' means the transaction finished successfully whereas '0' suggests the transaction failed. The second parameter is the number of retries left before hitting the maximum retry number ($MAX\_RETRIES$).

The same two features will be used for the calls to `predict` and `update`. The first argument to `predict` is this feature vector and the second argument is the feature length (2). If the result of `predict` is above the threshold (`USE_HTM`), the program attempts the HTM path, otherwise, it falls back to using the underlying lock without trying the HTM. The feedback to the prediction is given after the critical section ends, in the `TxUnlock()` function. If the perceptron recommended taking the HTM path (`tryingHTM` is true), then a successful fastpath rewards the perceptron by invoking the `update` API function with +1 (Line 9); however, if the perceptron recommended the HTM path but the HTM failed, we penalize it with a negative reward of -1 (Line 30). To avoid the perceptron becoming trapped in only the lock path after several failed predictions, a predetermined threshold is also set.

3.4.2 Page Reclaim and Congestion Wait

When memory gets tight, the operating system memory management subsystem starts to reclaim already used pages for later use. During the reclaim process, pages with modified contents need to be written out before the reclaim can occur. However, if the devices that the pages will be written out are already congested with other traffic, there is very limited benefit to adding extra I/O requests.

To mitigate congestion problems in the Linux kernel, a tracking mechanism for block devices was proposed in 2002 [139] which was adopted in v2.5.39 [140]. If the devices are congested, the memory management sub-system would not create any new I/O requests before the congestion is resolved. This idea has been extended in various ways and such a mechanism still exists in Linux kernel 5.15 as `congestion_wait()` [141].

Unfortunately, over the years developers have found that there are several limitations to the congestion-wait mechanism. First, congestion tracking suffers from an inherent race condition as the degree of device congestion can change before the query returns to the caller. Second, accurate tracking of congestion has become more difficult as storage devices have come to support longer command queues. As a result, *congestion_wait()* is used in practice only when the timeout expires, which is not at all what it was originally intended to do [142].

To overcome the limitation mentioned above, in 2021 it was proposed that all instances of congestion-wait in the source code should be completely eliminated [143]. The proposed new design reclassifies the original congestion wait into three sub categories and handles each one differently:

- When there are too many dirty or writeback pages, sleep until enough pages are cleaned or a timeout expires
- When there are too many isolated pages, sleep until enough of them are put back into the LRU system or reclaimed
- When there is no progress in page reclaim, the direct reclaim task sleeps until another reclaim task proceeds with some acceptable efficiency

The third point specifically measures the efficiency of another reclaim task by dividing the number of pages reclaimed by the number of pages scanned: $\frac{nr\_reclaimed}{nr\_scanned}$. In the most

recent patch, the efficiency threshold is set at a fixed value of 12.5%. However, as the proposer of this technique rightly points out, the fixed threshold value may not work for all scenarios. Here we see yet another opportunity to apply PSS to optimize control of the system, in this case dynamically optimizing the sleep condition instead of relying on a fixed ratio.

We input several parameters into PSS and dynamically decide if the current reclaim task should sleep or not. The parameters include the rounded values of nr_reclaimed and nr_scanned as well as the ratio of $\frac{nr\_reclaimed}{nr\_scanned}$. Since PSS only takes integer inputs currently, we use the reciprocal of the ratio and rounded to the closest integer, i.e. $floor(\frac{nr\_scanned}{nr\_reclaimed})$. If the returned result is greater than or equal to 0, the task will not go to sleep.

For this use case `update` is not as straightforward as `predict` — how does one know that the prediction was "wrong"? While we don't have direct access to ground truth, we can instead assume that entering the page claim throttle function is a negative sign for the last decision since overall page reclaim is a procedure that we want to minimize. Therefore, we keep a timer via *ktime_get()* to measure the timestamp of the last entrance of the function and the duration between two entrances. If the duration becomes longer, it means that the page reclaim has been invoked less frequently and we will reward the weights that lead to such a decision. Otherwise, it suggests that the reclaim happens more often and we will penalize the weights accordingly. In the end, even though we are inferring prediction and misprediction indirectly, we are able to limit the scope of our code changes to only the original function, *consider_reclaim_throttle*.

3.4.3 JIT Parameter Tuning for PyPy

Python is one of the most popular languages because of its simple syntax. However, the default interpreter implementation (CPython) suffers from slow execution speed brought by the extra interpreter layer. To recover some of that performance, Just-In-Time (JIT) compilation can be used to translate frequently executed code snippets into machine code that can be executed directly. PyPy is one of the most popular tools for doing so, in part because of its efficient tracing-based JIT compiler [144].

Unlike a method-based JIT that compiles an entire method at a time, a trace-based JIT only considers the frequently executed code paths (a.k.a., "hot path") within a method.

In PyPy specifically, there is a critical parameter named *threshold*, which decides whether a loop is *hot* or not. The default value of *threshold* happens to be 1039, meaning a loop will trigger the JIT tracing mechanism on the code path only after the loop has been executed 1039 times.

Like *threshold*, there are 16 other parameters in PyPy that control the tracing and compilation mechanism [145]. We detail the subset of the parameters we utilise and their default values in Table 3.1.

parameters	Default	Descriptions
decay	40	amount to regularly decay counters by
function_threshold	1619	number of times a function must run for it to become traced from start
loop_longevity	1000	a parameter controlling how long loops will be kept before being freed
threshold	1039	number of times a loop has to run for it to become hot
trace_eagerness	200	number of times a guard has to fail before we start compiling a bridge
trace_limit	6000	number of recorded operations before we abort tracing with ABORT_TOO_LONG

Table 3.1: List of selected PyPy JIT parameters.

While it is well understood that these parameters are critical, most prior work seeks to find an single static set that strikes that balance. One of the most commonly used methods of achieving such tuning is genetic algorithm (GA). For instance, Yu et al. [146] use

```
def main():
    features = {performance counters}
    for i in range(iteration):
        run the workload
        if predict(features, len) == True:
            set more aggressive JIT parameter
        else:
            set more conservative JIT parameters
        if curItrTime < prevItrTime:
            update(features, +1)
        else:
            update(features, -1)
```

Listing 3.2: Integration of PSS with PyPy JIT

GAs to optimize parameters for Spark while Li and Jiang [147] show that GAs can find PyPy parameters that can significantly outperform the default JIT parameters. However, GA and other static parameter tuning approaches require both expensive upfront overhead and a set of "representative" programs while to training. A large amount of data, machine and load dependence of results, and the significant design space exploration time required to make improvements all potentially limit the applicability of such an approach.

Using PSS, we can tune the JIT parameters on-the-fly without additional data collection and model training cost. Inspired by Li and Jiang [147], we choose the parameters within a set of prefixed values. The default value is multiplied by $\frac{1}{4}$, $\frac{1}{2}$, 2, and 4 to get the 4 new settings. The only exception is trace_limit of $4X$, which is set to 16000 instead of 24000 because of a range limit.

Listing 3.2 sketches the use of PSS in the PyPy JIT. After each iteration, we record the performance including the number of instructions and the execution time. More counter information can also be used if available, such as branch prediction and cache performance. We then feed this information into the perceptron as input features. The perceptron returns the decision on whether more aggressive optimization should be used or not and the JIT configurations will be set accordingly.

Once the timing information is collected with the new parameters, we compare it with the duration from the previous iteration. If the new parameters speed up the execution,

Benchmark	Description
intruder	Network intrusion detection
labyrinth	Maze routing
yada	Delaunay mesh refinement
SSCA2	Graph kernel
vacation	Travel reservation system
kmeans	K-means clustering
genome	Gene sequencing

Table 3.2: Benchmarks used from STAMP/HTMBench.

the corresponding weight will be increased; otherwise it will be decreased.

The input feature for the use case includes detailed information from PAPI [148] like the number of instructions and potentially different cache levels' hit rates. To better utilize them, we round the raw values before passing them to the perceptron. The rounding keeps only the most significant figures of a given integer. For example, 1234 will be rounded to 1000, 6276 will be rounded to 6000, and 1999 will be rounded to 2000. Rounding allows the perceptron to learn common input and prediction patterns.

3.5 Evaluation

We evaluate PSS on an 8-core (×2-way SMT [108]) Intel Coffee Lake CPU with a total 32GB memory, running Linux `5.15.0`. The CPU has 32KB L1I and L1D cache, 256KB L2 cache, and 16MB L3 cache.

For each applications in the three examples, we use STAMP/HTMBench [149, 75] as the HTM workload, MMTests for page reclaim benchmarks [150], and PolyBench-Python [151] and python-macrobenchmarks [152] as the PyPy JIT benchmarks.

In this section, we define the word *iteration* to describe the number of a dividable subroutine internally repeated in a benchmark program. And we use *benchmark run* to refer to one whole run of a benchmark.

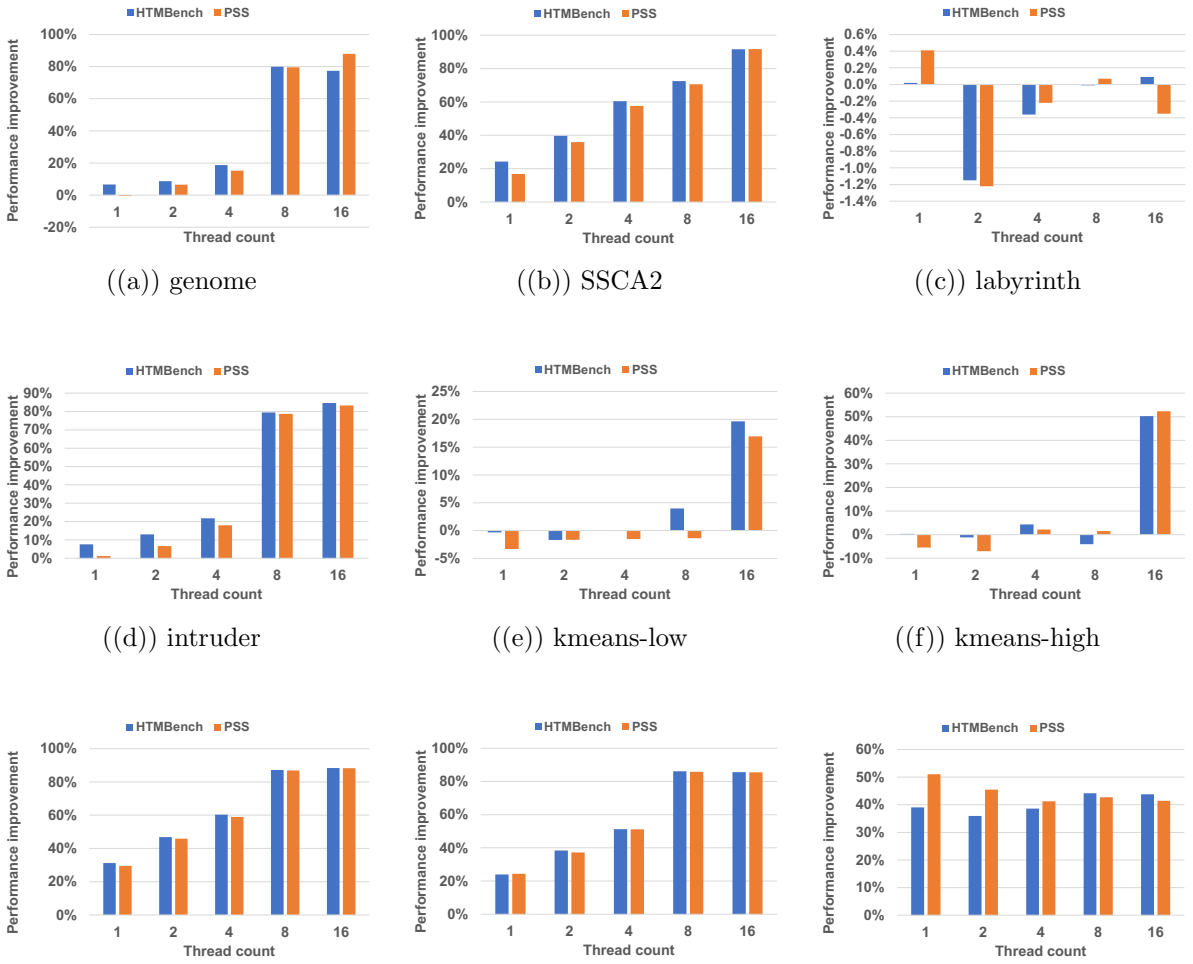


Figure 3.2: Performance of HTMBench and PSS HLE normalised to vanilla STAMP.

3.5.1 Hardware Lock Elision Results

We choose Stanford Transactional Applications for Multi-Processors (STAMP) [149] as the workload for HLE. STAMP is a collection of applications targeted for transactional memory research. The description of the benchmark programs can be found in Table 3.2. We use the recommended parameters for simulation setup.

HTMBench is the state-of-the-art benchmark suite of HTM and it is implemented using Intel's TSX [60]. It provides an efficient profiler to analyze HTM and offers op-

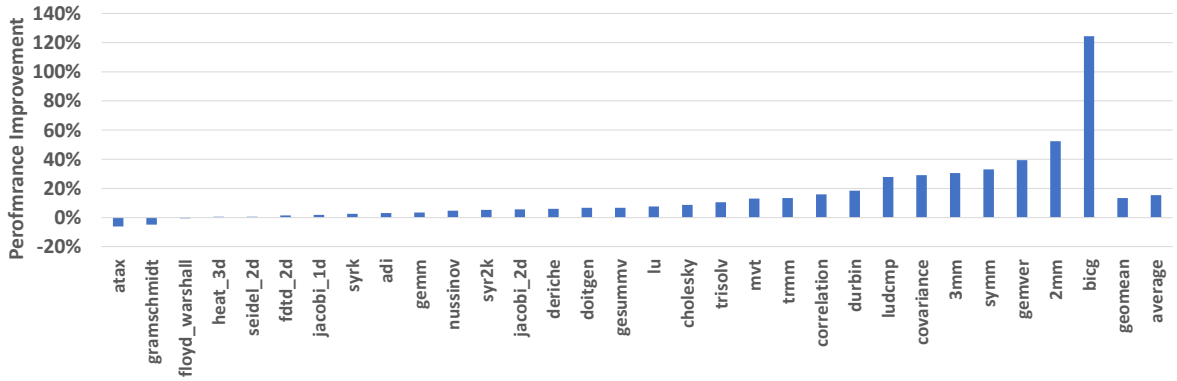


Figure 3.3: Performance improvement of PSS with 20 iterations on PolyBenchPython

timizations that generate nontrivial speedups. We compare our PSS implementation against HTMBench [75] and vanilla STAMP with HTM support as the baselines. We vary core count over 1, 2, 4, 8, and 16 cores. We run each program five times and report the median value of the results.

The result of STAMP is plotted in Fig 3.2, which shows the performance improvement of HTMBench and PSS over vanilla STAMP. Overall, the overhead of using PSS is relatively low. The most slowdown comes from 1 thread setting for kmeans-high in Fig. 3.2(f), where PSS optimized code generates 7.02% performance degradation. In most of the other cases, the slowdown is less than 5%. On the other hand, PSS optimized code can clearly show benefits over the vanilla baseline or even HTMBench in selected cases like Fig. 3.2(a) and 3.2(i). For instance, PSS leads to 87.62% of improvement for 16 threads setup in genome, which is 11% higher than HTMBench.

In terms of overhead, HTMBench has state-of-the-art implementations of STAMP after extensive profiling and optimization. On the other hand, baseline code patched with PSS is only trained a few hours and it performs very close to HTMBench or even outperforms it in several cases.

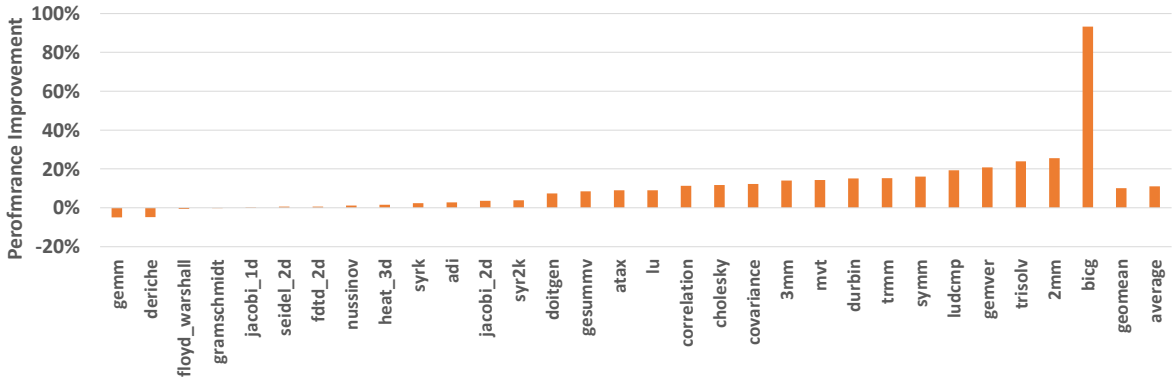


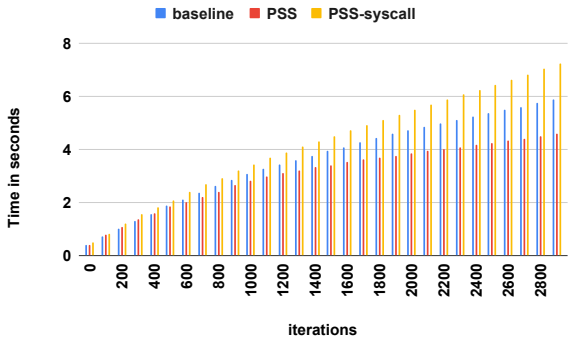
Figure 3.4: Performance improvement of PSS with 50 iterations on PolyBenchPython

3.5.2 PyPy JIT Results

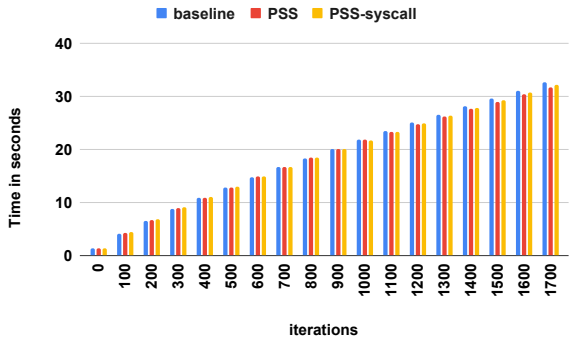
Benchmark Setup.

We use version 7.3.3 of PyPy as the JIT compiler and PolyBenchPython [151] with python-macrobenchmarks [152] as the workloads. PolyBenchPython is a benchmark suite with 30 commonly used kernels for scientific computing and it is representative as microbenchmarks. On the other hand, python-macrobenchmarks contains some of the most popular python applications on a macro-level, including Flask [153], Django content management system (CMS) [154], Gunicorn [155] and more.

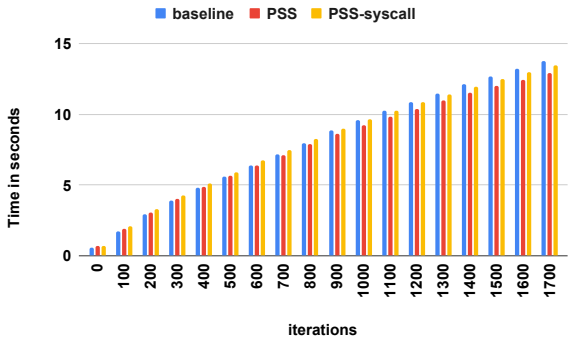
For PolyBenchPython, we run the benchmark using the default list implementation of the array and MINI as the input data size. Since the original PolyBenchPython already uses PAPI [156] counters, we include some of them as input features to PSS. Specifically, we use the execution time and the ratio between L1D hit and L1D miss as parameters for PSS. Each benchmark is executed 10 times and we report the time spent in the first 20 and 50 iterations. The baseline is the program with the default JIT setting while the modified JIT is the program patched with PSS, dynamically changing the JIT configuration parameters as we described in Section 3.4.3.



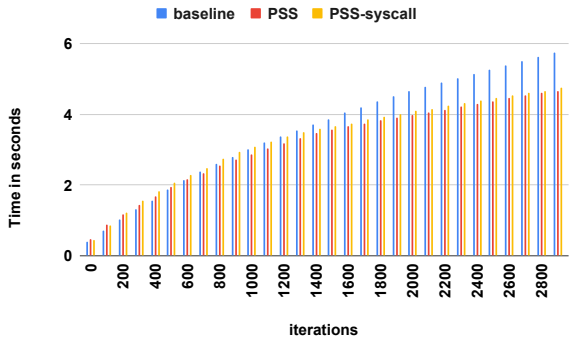
((a)) aiohttp



((b)) djangocms



((c)) flaskblogging



((d)) gunicorn

Figure 3.5: Result of macrobenchmarks.

PolyBenchPython Results.

The result of PyPy JIT parameter tuning is presented in Figure 3.3 and 3.4. On average, PSS can improve the performance of the 30 programs by 15.38% and 11.11% for 20 and 50 iterations, respectively. For the first 20 iterations, the largest improvement is over 120% while the largest slowdown is only around 6%. For 50 iterations, the largest performance gain and loss are smaller since most of the commonly executed code is already jitted in the late iterations. However, the improvement is still significantly larger than the slowdown. We believe this setup of optimization can be potentially useful for Function-as-a-Service (FaaS) applications, which tend to run short computation tasks over and over.

Macrobenchmark Result.

The result of the macro benchmark is plotted in Fig 3.5. We choose 4 benchmarks that can easily demonstrate performance iteration-wise and we simply use the iteration-wise runtime as the parameter for PSS. We run 3000 iterations for aiohttp and gunicorn and 1800 iterations for djangocms and flaskblogging. Each benchmark runs 5 times and we plot the averaged result iteration-wise.

It is clear that PSS can speed up the macro-benchmark with better dynamic parameter tuning. For the four benchmarks, the performance improvements are 22.17%, 2.54%, 6.3%, and 18.66%, respectively. From the two sets of benchmarks, we demonstrate the functionality and performance benefits of PSS for both micro and macro-benchmarks of Python.

Latency-Sensitive Applications.

Figure 3.5 also contains the result of using syscall as prediction instead of vDSO. From the results, it is clear that for the latency sensitive applications, implementation using vDSO performs better than syscall. The syscall-based results either have less speedup as shown in Figures 3.5(b), 3.5(c), and 3.5(d)) or generate significant slowdown as shown in Figure 3.5(a).

3.5.3 Page Reclaim and Memory Management

Benchmarks and Methodology.

We follow the experiments mentioned in the original patch [157]. We ran mmtests [150] on the original 5.15.0-rc3 kernel, the patched kernel [158], and the kernel with dynamic control from PSS. MMtests is a benchmark framework aimed at performance testing of the Linux kernel. Specifically, we ran a test named `stutterp`, which sweeps a different

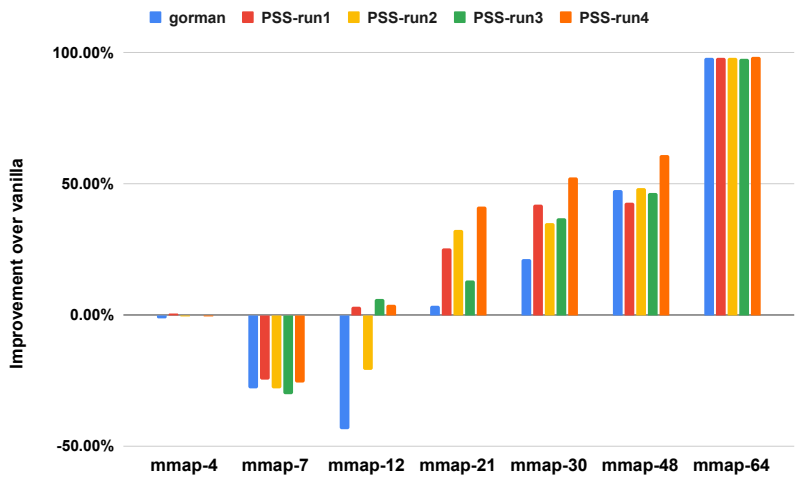


Figure 3.6: Average latency of MMTests. Each mmap-N represents a run with N worker threads. The larger the number, the higher the memory pressure.

number of "worker" processes and inspects the impact of the direct reclaim. There are four types of workers in `stutterp`:

- One "anon latency" worker: creates `mmap` mappings then measures the duration to fault the mapping.
- X file writers: flexible I/O tester (`fio`) that randomly writes X files. The total size of the files equals the preset *dirty_ratio*.
- Y file readers: `fio` that randomly reads small files.
- Z anon memory hogs: continually map memory with the ratio $(100 - dirty\_ratio)\%$.

The total estimated working set size (WSS) is $(100 + dirty\_ration)\%$ of memory. The motivation of `stutterp` is to maximise the total WSS with file and anonymous memory. During execution, some anonymous memory has to be swapped and it is very likely that dirty/writeback pages reach the end of the LRU.

Results.

The result of `stutterp` is plotted in Fig. 3.6. It shows the performance improvement compared with 5.15.0 vanilla kernel. The number after `mmap` indicates the number of the worker threads mentioned above and larger worker counts means higher memory pressure for the system.

From the plot it is clear that PSS can outperform the baseline implementation now merged into the kernel. The improvement is much higher for the 21, 30, and 48 workers setups and PSS achieved slight improvement where the baseline suffers significant performance loss for 12 workers. For 7 workers, all the implementations perform worse than the vanilla code, but the slowdown is less for PSS code after several iterations.

Another benefit we can observe from the figure is that the performance of PSS is improving over multiple benchmark runs. It does not show a monotonic increase, but shows improvement as the general trend over time. On the other hand, we tried to run the baseline version multiple times and we did not observe any noticeable improvement.

3.6 Related Work

Since prediction is a key feature of the system software stack, there have been many different implementations which have been demonstrated to take advantage of common system operations and communication patterns for prediction to improve performance and other system metrics. Kraska et. al. [159] and Mitzenmacher et. al. [160] survey recent work which make use of prediction and machine learning for systems.

SmartChoices [161] is the most relevant work to our proposed system service for prediction. Similar to PSS, it also proposes a set of interface functions that software can use to make predictions, as well as ability to do on-the-fly learning. However, their proposed system is based on Reinforcement Learning which requires significantly more resources

for training and incurs higher latency compared to a simple perceptron-based predictor. Thus, it has limited applicability in resource-limited systems which only require simple and fast predictions.

Other than the work mentioned on prediction memory access and synchronization mentioned in Section 3.2, there have been several other synchronization algorithms which have a fastpath/slowpath or other variants [162, 163, 164, 165, 123, 166, 167] and the decision to dynamically choose the correct variant is predicted based on the past behavior and current conditions.

Low-level runtime systems for dynamically adjusting to power and energy consumption employ lightweight prediction mechanisms [168, 169, 170, 171]. Esmaeilzadeh et al. present a learning-based technique to accelerate approximate programs [172]. In their work, programmers can label a code region to approximate and then a NN model will be trained to emulate the region. Once the training is complete, the original code region will be replaced by the invocations to a low-power Neural Processing Unit via an ISA extension. Furthermore, system failure prediction [173, 174, 175, 176, 177, 178] has also attracted a lot of attention in recent times due to very large scale systems and increased failure rate. Finally, searching for the set of compiler optimizations and their order of application employs various prediction techniques based on past learnings and behaviors [179, 180, 181, 182, 183].

There are many studies focused on how to automatically tune the configuration settings for different kinds of software systems. In general, those studies can be classified into two groups. The first group utilizes a certain type of search-based algorithm, including hill-climbing [184], genetic algorithms [146], and ParamILS [185]. The second group tries to find the optimal configurations by reduction, including the iterative experiment [186] and similarity measurement [187].

Our proposed system service prediction is flexible enough to be used and bring per-

formance benefits in most of these prediction scenarios and avoid the complications of parameter tuning. Compared to traditional approaches, our proposed service offers advantages in terms of lower effort and resources needed with on-the-fly tuning and better reusability.

3.7 Conclusion

The effective end of processor frequency scaling and the continued drive for higher performance and lower energy utilization means that application-targeted software optimization will only continue to grow in importance in the field. While there are sure to be many application specific optimizations that do not rely on prediction, a surprisingly diverse class of optimizations, from hot-path/cold-path, to parameter tuning, to resource optimization, and beyond are more easily and readily enabled through support from a simple to call and low-latency software service. The move to a new abstraction that is useful in the process of optimization helps us step away from both the fragile heuristics so common in production code today while avoiding inheriting the complexity of complete application-embedded prediction frameworks. A system service for prediction has the potential to enable performance optimizers to spend their time worrying more about the discovery of new opportunities for specialization and tuning, and less about how exactly one should navigate the space of trade offs such opportunities live in. Even if there are times when such a service might not be appropriate for a final deployment, a prediction service can still be helpful in the development process by speeding up the sorting of promising optimization opportunities from those that will offer little gain even with well-crafted heuristic control. A core idea of prediction as a service is the decoupling of the creation of optimizations and the specific decision of how and when exactly to apply those optimizations.

We demonstrate the utility of Prediction as a System Service across three application-targeted optimization scenarios, and in all three cases find performance improvements. As to be expected such an approach is highly latency sensitive, but we are able to demonstrate a creative new use of vDSOs that can allow applications to extract predictions in an average of 4.19ns. In all of the cases we examined this new optimization pattern with operating system service support is able to meet or beat the best known hand-crafted methods with a fraction of the complexity of existing hardware. We believe this approach would be of particular interest to the community as it also opens the door for new and creative uses of architectural support for assisting in prediction with low latency, language level opportunities for the exploitation of prediction services, and further innovation in the system-level abstractions appropriate to more fully support dynamic control of software optimization.

For future work, we believe multiple directions can be explored on top of the current proof-of-concept implementation. For instance, how to share PSS among multiple applications like VMs and containers and handle the security concern effectively. Another interesting topic is to measure the performance impact of multiple threads or processes to access the PSS simultaneously.

Chapter 4

CRISP: Critical Path Analysis of Large-Scale Microservice Architectures

4.1 Introduction

Microservice architectures [23, 24, 25, 26, 27, 28, 29] have become the lifeblood of modern service-oriented software systems. As opposed to monolithic software development and deployment, in a microservice environment, the business logic is broken into individually deployable programs, which allow fast development and scalable deployment. Individual microservice *instances* interact with one another via remote procedure calls (RPCs). As microservices evolve with the business, they grow in number and their interactions become complex.

Uber's backend is an exemplar of microservice architecture. Uber has ∼4,000 microservices interacting with each other via RPCs. Each microservice hosts a handful of APIs, leading to a total of about $40,000$ unique RPC endpoints that can call one another

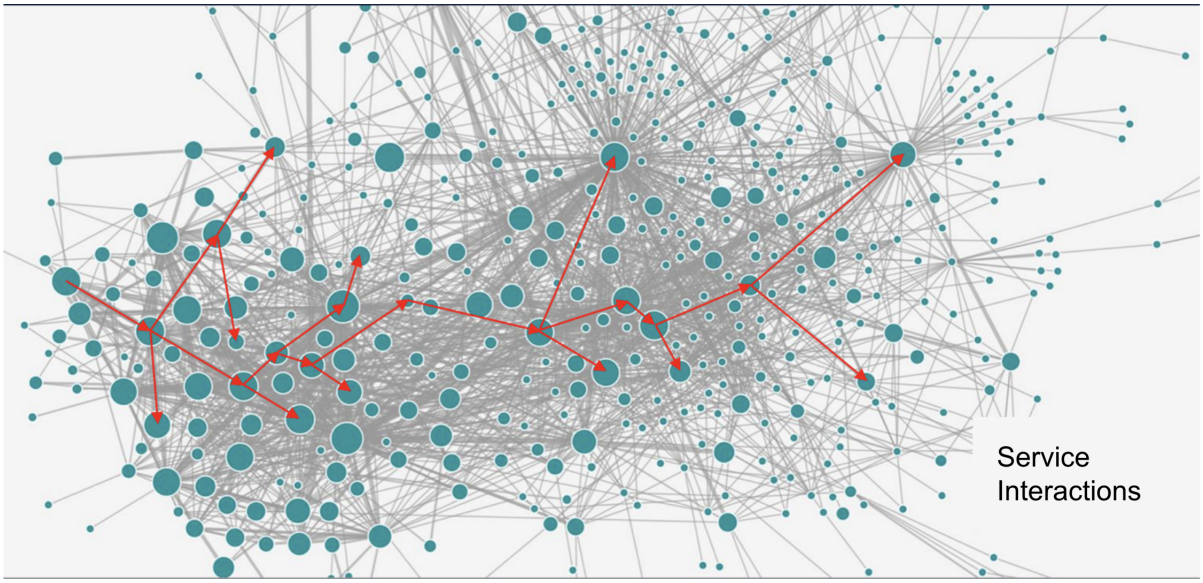


Figure 4.1: Complex microservice RPC call graph at Uber collected via Jaeger tracing.

in complex ways, as depicted in Figure 4.1. Hereafter, we use the terms *endpoint* and *API* interchangeably to mean a uniquely named functionality provided by a service. We use the terms *operation* and *RPC* interchangeably to mean an instance of invocation of such an API.

A service request arriving at an entry point API to the Uber backend systems undergoes multiple "hops" through numerous microservice RPCs before being fully serviced. The life of a request results in intricate microservice interactions. These interactions are deeply nested, asynchronous, and invoke numerous other downstream APIs. As a result of this complexity, it is very hard to identify which underlying service(s) contribute to the overall end-to-end latency [30, 31, 32, 33, 34, 35, 36] experienced by a top-level request. Answering this question is critical in many situations. For example:

- Identifying optimization opportunities for a top-level microservice (e.g., reducing tail latency)
- Identifying bottleneck APIs that affect numerous endpoints
- Setting appropriate time-to-live values for RPCs

- Diagnosing outages and error conditions
- Planning for computing and other capacity management

The critical path [188] is the longest chain of dependent tasks in a microservice dependency graph. Reducing the critical path length is necessary to reduce the end-to-end latency of a request. Hence, latency optimization efforts benefit from prioritizing the services that are on the critical path.

We have developed a tool, CRISP [1], to pinpoint and quantify performance problems in microservice architectures. CRISP uses the RPC tracing facility provided by Jaeger [189] and constructs the critical path through a request's graph of dependencies. The critical path may vary among requests; hence, CRISP computes the critical path per request. It then aggregates and summarizes critical paths from millions of requests. Finally, it presents them as digestible and actionable insights via rich heat maps [190] and flame graphs [191]. CRISP provides knobs to dissect the details with different percentile values that help in performance diagnoses.

As a full-fledged performance analysis tool, CRISP caters to various use cases via the following rich set of capabilities that scale to work on millions of traces:

- **Top-down analysis**: A top-down analysis of any specific endpoint of interest. This capability allows service owners to deep dive into their specific endpoint and pinpoints and quantifies bottlenecks encountered in the RPC dependency graph. Improving these bottlenecks should be the first-order priority to reduce the latency of the endpoint.
- **Bottom-up analysis**: A bottom-up analysis over all endpoints, which bubbles up and ranks by the impact of those interior APIs that cause the most latency across most endpoints. Optimizing these interior APIs reduces latency across numerous endpoints.
- **Neural network-based anomaly detection**: An automated anomaly detection system, which detects whether a request is exhibiting abnormal behavior compared

[1] named taking letters from **cri**tical and **sp**an

Figure 4.2: Critical path(s) of `createOrder` endpoint shown as a flame graph via CRISP after processing 100K Jaeger traces.

with the past history of the endpoint. The system is trained per endpoint using an autoencoder-decoder machine learning technique [192]. This system is set up to expedite problem detection and alert developers. Basing the abnormality detection on the divergence in the critical path as opposed to the full call graph [192] not only makes the training and inference faster but also reduces false alerts.

Practical deployment of CRISP at Uber over a fifteen-month period working on 40K endpoints while processing $\sim 200GB$ of traces with ∼18 million spans in $\sim$ 256 hours of CPU time per day has resulted in the following impact:

- Detection and narrowing down the causes of five latency impacting bugs in two business-critical services
- Identification of a 1.5× tail latency lengthening due to hardware choice and the resulting guidance for future hardware selection
- Up to 27.77× speedup in training, up to 66.85× speed up in inference, and 50% reduction in false alerts in identifying abnormality of service behaviors over the state of the art [192]

The rest of this work is organized as follows: Section 4.2 motivates CRISP with a use case at Uber, Section 4.3 describes the Jaeger tracing framework, Section 4.4-4.6 describe the methodology, internals, and features of CRISP, Section 4.7 evaluates CRISP at Uber, Section 4.8 discusses the related work, and Section 3.7 offers our conclusions.

4.2 Motivating Example for CRISP

Fulfillment [193] at Uber is a platform to orchestrate and manage the lifecycle of orders and user sessions with millions of active participants. The Fulfillment platform is a foundational Uber capability that enables the rapid scaling of new verticals. The platform handles more than a million concurrent users and billions of trips per year that span over ten thousand cities. The platform handles billions of database transactions a day. Hundreds of Uber microservices rely on the platform as the source of truth for the accurate state of the trips and driver or delivery sessions. Events generated by the platform are used to build hundreds of offline datasets to make critical business decisions. Over 500 developers extend the platform using APIs, events, and code to build more than 120 unique fulfillment flows.

The `createOrder` endpoint allows capturing the requester's intent in the Uber backend. Intent can be to request a ride from one of the ridesharing lines of products, food booked and dispatched by one of the courier partners, or a package be delivered to a customer. This endpoint has a complex task dependency graph necessary for: a) determining order risk such as user fraud, sufficient user balance via authentication hold, b) ensuring the fare presented to the requester in the shopping phase is still valid, c) determining the benefits the requester is eligible for, d) enriching data with location information, and e) creating an order in the backend to start the matching process.

The tasks in this endpoint have grown organically as requirements evolved. This has led to an increase in p95 latency to 6 seconds, affecting user experience. The service itself is written in Java, and highly (both macro and micro) optimized using periodic profiling. However, the profiling offered no insights into downstream calls, where most time is spent. Quantitative insight into the causes of the latency was hard to analyze by looking at individual traces because each trace contains thousands of nested and

overlapping RPCs.

There are numerous sampling- and instrumentation-based profilers [194, 195, 196, 98] for intra-service profiling. However, they do not collect metrics at the individual request level. The Fulfillment microservice (as most other microservices) is highly threaded; the work of an individual request may be partitioned among multiple threads within a process as well as multiple threads may be handling independent requests simultaneously. In such a setup, traditional profilers fail to highlight the causes of latencies incurred at an individual request level. Also, traditional profilers fail to capture IO waiting, task dependencies, and serialization patterns.

With CRISP, the development team performed a top-down critical path analysis of this endpoint over 100K traces (∼200GB of traces) and visualized the results as a flame graph as shown in Figure 4.2. Navigating the "hot" critical paths via the flame graph not only corroborated an existing hunch while offering quantitative guidance but also shed light on new optimization opportunities lurking in the wild. Below, we enumerate a few defects and optimization opportunities that became evident by inspecting CRISP-provided insights.

Async flow optimization: `decideOrderRisk` contributes to about 68% of the end-to-end P50 latency, revealing the following optimizations: a) aggressively use cache in `FraudScore` to reduce its latency and b) parallelize the calls beneath this big endpoint (e.g., `PaymentAuthHold` and `FraudScore`). In the long term, the team envisions using an asynchronous invocation of `paymentAuthHold` and using notification to the requester when a provider is assigned.

Unnecessary API serialization: There was an unnecessary serialization between `GetVenues` and `GetAccessPts`. These two RPCs can be done in parallel.

Avoidable server roundtrip for validation: `FareValidate` contributes to about 5% of the end-to-end P50 latency. This is a call that need not be performed every time.

Trusted edge devices (e.g., company mobile app) can validate at the edge improving performance for trusted users and falling back to server validation if the fare has expired based on fare expiry TTL; untrusted apps will use the full server validation.

Caching over DB fetch: `GetMarketplaceBenefits` contributes to about 5% of the P50 latency. This can be served via a cache rather than a database read to fetch requester benefits.

4.3 Background

In this section, we first describe the microservice tracing infrastructure at Uber and then enumerate its shortcomings.

4.3.1 Distributed Tracing at Uber

Microservices run over several physical hosts, usually owned by multiple teams, and written in multiple languages. It is impossible to use traditional profilers [195, 194, 197] to gain insight into the events involved in processing a request. Because each physical host can have a separate clock, it is intractable to infer causality using time alone. Distributed tracing [198] encodes causality information in a distributed context, which is propagated across process boundaries. It provides a way to infer states across various services for the lifetime of a request.

At Uber, Jaeger [189] is used as the distributed tracing system. Jaeger provides clients for generating trace data and components for storage and retrieval of traces. Microservices instrumented with Jaeger clients produce OpenTracing [189] -compliant spans when receiving new requests and attach distributed context information (trace ID, span ID, custom key-value pairs). The "span" [199] is the primary building block of a distributed trace, which represents a serial unit of work done in a distributed system.

Each span contains the following information:

- API name
- Start and finish timestamps
- Custom key-value pairs
- Span context and references (described below)

Each span may reference other spans with a causal relationship by span context. A span may reference a parent with the `ChildOf` relationship, indicating that the parent span waits for the child to finish a certain task. Multiple child spans can be referenced by the same parent and run concurrently.

While the source code is always instrumented, the overhead is controlled by a dynamic sampling rate, which is adjusted based on the traffic received by Internet-facing endpoints. No data is collected for traces that are not sampled. Specifically, adaptive sampling sets a target QPS for traces on a per root service-endpoint basis, which ensures that the number of samples on the external API request remains roughly constant. Jaeger does not support tail-based sampling [200].

Figure 4.3 depicts the Jaeger deployment at Uber. Jaeger is deployed as multiple components, with a `jaeger-agent` running on every host. All applications running on this host send spans to `jaeger-agent` over UDP [201]. `jaeger-agent` then forwards these spans to a `jaeger-collector`, which then buffers spans onto the Kafka [202] distributed event streaming platform. The spans buffered in Kafka have multiple consumers: `jaeger-ingester`, which inserts them into Docstore[203], a distributed SQL database, and allows for retrieving full traces; `jaeger-indexer`, which inserts them into Sawmill[204], a schema-agnostic logging platform that allows user-friendly search on spans fields. Additionally, spans are consumed by Apache Flink [205] jobs to produce multi-hop dependency graphs. Depending on the sampling configuration in effect, the backend processes around 400K-1M spans per second, which is approximately 20TB each

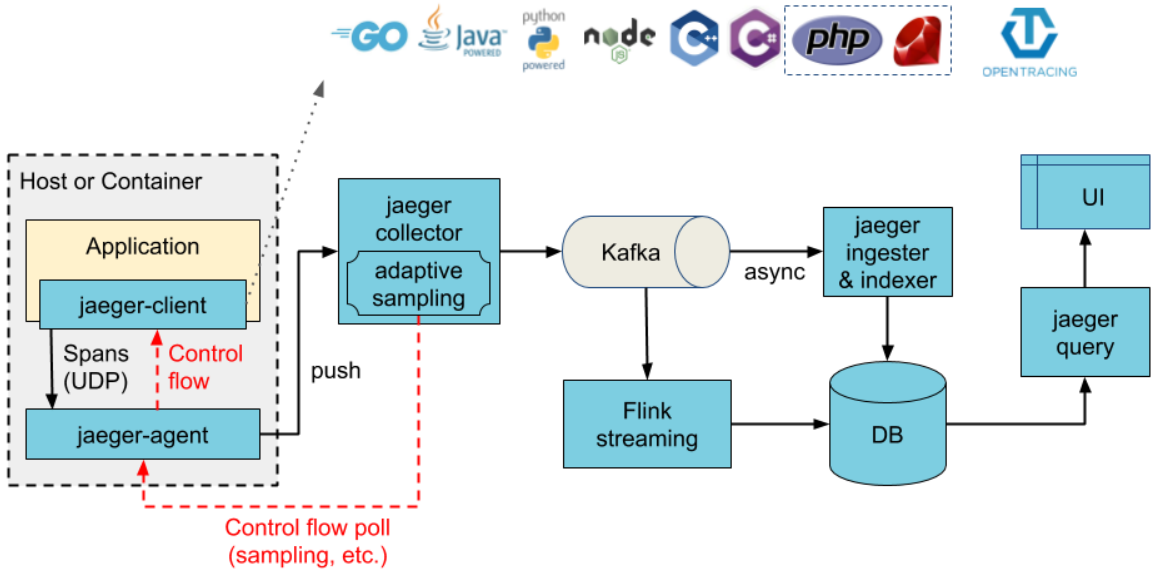


Figure 4.3: Jaeger deployment at Uber.

day. Variance is common due to diurnal patterns.

4.3.2 Difficulties with Large-Scale Jaeger Traces

Despite their power, Jaeger traces are highly complicated. Jaeger provides a UI to filter traces by time ranges and also provides a UI to view the trace as a callgraph, as well as an expandable tree over a timeline. In spite of these facilities, the users of this manual workflow often complained about the following limitations to analyze endpoint latencies:

- Only first-level insights are possible from drilling down into microservice latencies and errors.
- Using a few Jaeger traces is insufficient to reach a reliable conclusion. Users can visualize and navigate only one Jaeger trace at a time. There is no aggregate summary of traces.

- A single Jaeger trace can be so complex that it is not humanly possible to browse and understand the details. Endpoints commonly have thousands of nodes in the RPC graph with 25-deep call chains and up to 40 spans overlapping in time. It is cumbersome to manually understand the critical path due to the asynchronous nature of calls.
- There is a lack of regular, performance-driven feedback tooling to optimize the workflow or downstream systems.

These challenges introduce a barrier to our developers in effectively using Jaeger to either detect anomalous situations or identify optimization opportunities.

4.4 CRISP Methodology

The fundamental difficulty in making sense out of a Jaeger trace is due to the complexity of the graph. Our premise is that while the whole graph is interesting in terms of data richness, it brings a lot of noise. There are many RPCs and call paths that are insignificant for a high-level analysis and optimization task. With this understanding, we shrink the graph to its quintessential element—the *critical path*—and aggregate many traces into a single summary that is still rich with call path information.

Critical Path Analysis[188, 206] (CPA) is a well-studied concept over directed acyclic graphs (DAG) formed out of computing graphs in parallel computing. The nodes in the DAG represent tasks (units of serial execution) and the edges represent dependencies between tasks. A node with an out-degree greater than one "spawns" children's tasks and a node with an in-degree greater than one waits ("syncs") for the children to finish. Total work is the sum of weights of all nodes and the critical path is the longest weighted path in the DAG.

[Critical Path] In a task graph $G = (V, E)$ made of task vertices V and their depen-

dency edges E, with two special vertices S (start node) and F (finish node), the critical path is a maximal-weight path from S to F. G may contain more than one critical path.

The critical path identifies the sequence of dependent computations that consume the most time. To speed up the service, it is strictly necessary to boost the components on the critical path.

RPCs among microservice operations have a parent-child hierarchical relationship and can be construed as a parallel computation DAG. The deriving critical path from Jaeger traces, however, has the following challenges:

- Unlike a traditional parallel computing DAG seen in the academic literature, the Jaeger traces do not provide clear "spawn" and "sync" events in the DAG.
- The parent spans in Jaeger traces carry no dependence information and so the information of the last "sync" child span is not directly available.
- In order to obtain the last "sync" child span, clock information is needed. However, the clocks on different machines where spans are collected are not time-synchronized.
- The critical path across all requests may not be unique. Services have diurnal patterns and different traces may exhibit different critical paths, which need to be aggregated, and yet "hot" critical paths need to be bubbled up.
- Since the service codes keep evolving, the critical path keeps changing.

We address these challenges in the next section.

We also mention in passing that the CPA is not a performance analysis panacea. Once the exposed latency on the critical path is eliminated, a new critical path may emerge which necessitates the need for an iterative profiling and optimization approach.

4.5 Critical Path Analysis

In this section, we detail how we compute, aggregate, and represent critical paths from many Jaeger traces for a given endpoint.

4.5.1 Deriving Critical Path from a Single Trace

CRISP's trace analysis exploits a map-reduce paradigm to process millions of traces belonging to each endpoint. To this end, each process loads an input Jaeger trace file (JSON format) and builds an n-ary tree, where each parent node is the RPC caller and the children nodes are the immediate downstream callees.

In order to compute the critical path through the trace, we need a computational DAG. To accomplish this under Jaeger/Opentrace trace format, we make use of the start and end times of children's spans. The start time in every immediate child creates a "spawn" event in the parent and splits its span at that point in time. Similarly, the end time in every immediate child creates a "sync" event in the parent and splits its span at the point in time. Thus, we transform the tree into a logical DAG for critical path construction.

Figure 4.4 shows an example DAG constructed from Jaeger traces by looking at span start and end times. In Figure 4.4, the span A is the root span, which invokes spans B, X, and D. The span B in turn invokes span C. The start time T_1 and its end-time T_6 of B create a spawn and sync points on A, respectively. Similarly, the spans X and D, create further segments in A. Similarly, B's child C, creates the spawn and sync points on B at T_3 and T_4, respectively.

Limitations of Jaeger/Opentrace format: One key limitation of the Jaeger is that the parent spans (a.k.a., `caller`) do not contain dependence information. Specifically,

they lack the information of both start and end of `callee` RPC. Instead, it is the `callee` that stores both the ID of its parent and `callee`'s start and end time (per callee's local clock) in its own span. The implication of the constraint is that the dependency relationship needs to be inferred via clock information recorded in the callee span.

In addition, the inference can be inaccurate because of the clock skew that will be discussed in Section 4.5.2. Traditionally, the computation of the critical path depends on the last returned child of the parent spans [207]. In Jaeger traces, the last arriving child information is not directly recorded in the parent span. Instead, the last arriver needs to be inferred using the span end time for each child, which will be based on each child's local clock. Without correctly handling the clock skew, the critical path analysis can go wrong.

One may extend Jaeger tracing by making the callee return additional data to the caller. Unfortunately, ensuring that these changes are adopted universally across thousands of services is an engineering hurdle. Such changes also require support from different RPC libraries used by our system. Our solution, in contrast, does not require such large-scale system-wide changes but yet produces high quality results as we describe in the rest of this section.

Critical Path Algorithm

We, first, describe how we compute the critical path in a trace assuming perfectly synchronized clocks in this subsection. We expand to handle unsynchronized clocks in Section 4.5.2.

The process of computing the critical path (CP shown in Listing 4.1) on the logical DAG starts at the root node R—the endpoint under study. We sort all its children by their span *end time* and pick the last finishing child (LFC). The entirety of LFC is on the critical path. Let LFC_s be the start time of the LFC; we ignore all children spans

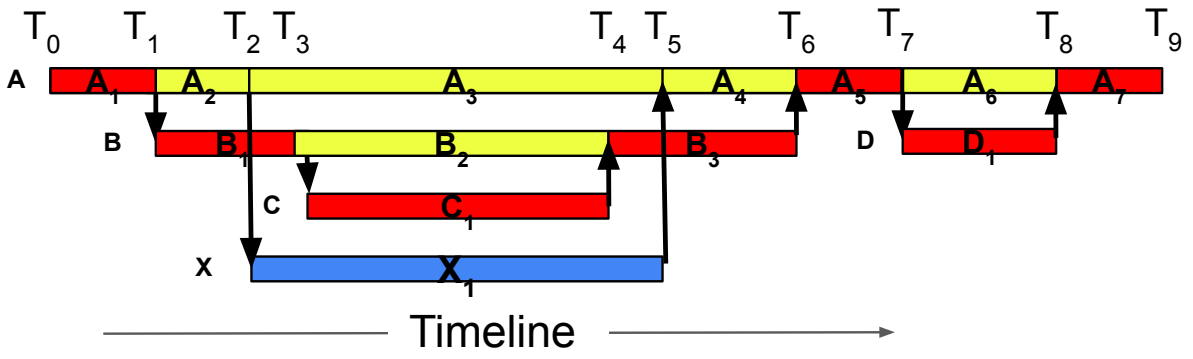


Figure 4.4: Trace with root span A, its children B, X, and D. B further calls C. CRISP further segments each parent traces based on the start and end time of its children. The red-colored blocks represent the critical path through the trace.

```
1  def CP(root):
2    path = [root]
3    if len(root.child) == 0:
4      return path
5    children = sortDescendingByEndTime(root.children)
6    lfc = children[0]
7    path.extend(CP(lfc))
8    for c in children[1:]:
9      if happensBefore(c, lfc):
10       path.extend(CP(c))
11       lfc = c
12   return path
```

Listing 4.1: Pseudocode to compute critical path.

of R that may start or end in the time intervening between the start and the end of LFC. We now look for the next child of R whose end time immediately precedes LFC_s and perform the same procedure iteratively until no child is left to process. Time not attributed to any child of R is attributed to the root span itself.

The process is also recursive. Once an LFC is identified, it recursively calls CP on its own children to distribute its time under its children. The result of the CP algorithm is a sequence of graph nodes with time associated with each one of them. Applying this algorithm to the trace shown in Figure 4.4, the critical path is represented by the fragments $A_1B_1C_1B_3A_5D_1A_7$.

There are two types of metrics associated with each node of the critical path — inclusive time and exclusive time. The "exclusive" time does not include the time spent

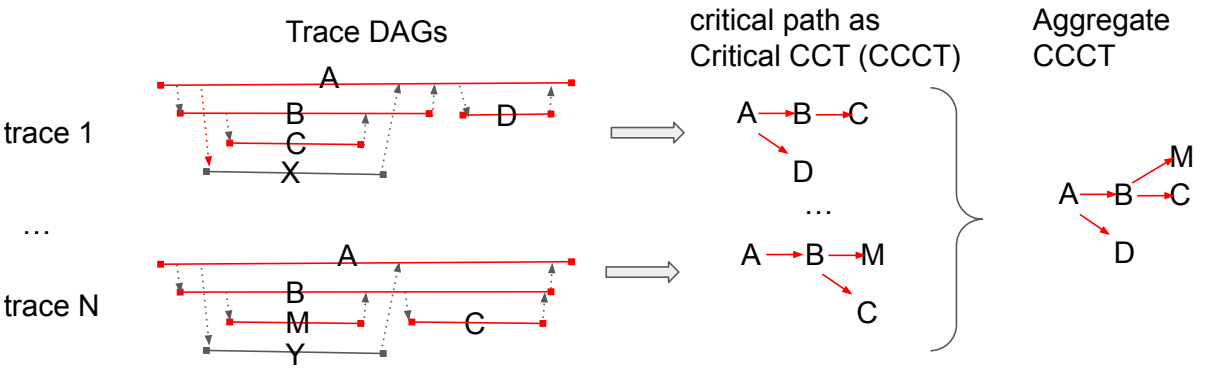


Figure 4.5: From trace to DAG to critical path (CCCT) to aggregate critical calling context tree. In the trace DAGs (left of the diagram) the x-axis is the flow of time. Horizontal lines are Jaeger spans and vertical lines are caller-callee relationships. Red-colored horizontal spans are on the critical path.

in a node's callees. The "inclusive" time is the total wall clock time from the start to the end of the RPC on the specific node.

Since every node on the critical path encodes the information on how it was called, and since all call paths originate from a common root — the endpoint under investigation — it enables us to merge all call paths into a calling context tree (CCT) [208] by looking at their common prefixes. Consider the critical path $A_1B_1C_1B_3A_5D_1A_7$ for the trace in Figure 4.4. This path encodes the following call and return information: A calls B calls C returns to B returns to A calls D returns to A. With this, we can infer that there are the following call chains involved on the critical path: A, $A \rightarrow B$, $A \rightarrow B \rightarrow C$, $A \rightarrow B$, A, $A \rightarrow D$, and A. We can merge all these call paths into a CCT and call it a Critical Calling Context Tree (CCCT). This process is presented in the center section of Figure 4.5. The calling context information makes it not only rich but also helps in aggregating critical paths from multiple traces described later in Section 4.5.3. A level of aggregation happens immediately within each trace processing: if the same endpoint appears multiple times on the critical path, we sum them as long as their call chains are exactly the same. For example, in the previous $A_1B_1C_1B_3A_5D_1A_7$ critical path example,

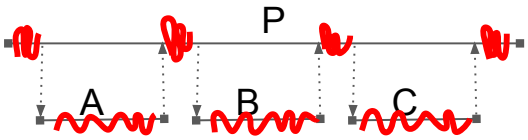


Figure 4.6: Ideal traces for a parent with three serialized children executions. Red lines show the critical path.

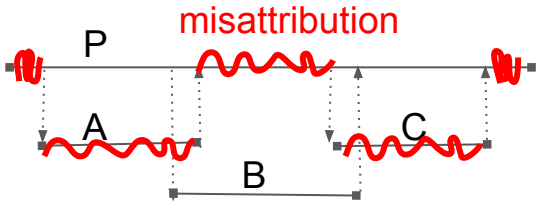


Figure 4.7: Actual traces due to clock drift. Red lines show the corresponding critical path.

we merge the multiple occurrences of call paths A_* and $A_* \rightarrow B_*$. This merger discards the ordering relationship between events, which we do not need for further analysis.

4.5.2 Challenges with the Clock Drift

The span start and end times recorded in Jaeger traces are both callee's local-machine time stamps converted to the standard UTC time. Machine clocks on two different physical machines drift [209, 210, 211] despite their periodic NTP-based synchronization. As a consequence of using local clocks, our critical path algorithm (if not corrected) can go wrong and sometimes lead to significant misattribution.

Span overlap problem: Figure 4.6 shows an ideal trace where the three spans A, B, and C are invoked one after another by the parent P. Most of the time should be attributed to the children. Figure 4.7 shows the trace for this example from our production, where the time recorded for the children spans have a small overlap; there is an overlap between the end of A and the start of B and the end of B and the start of C. In this case, the critical path is not attributed to span B and instead attributed to the parent. Due to the clock drift, more than 50% of our traces recorded this type of span overlaps causing misattribution in critical paths.

We conducted a detailed study on the impact of such clock drift. Figure 4.8 plots the time overlap recorded in Jaeger traces of two sequentially invoked RPCs sampled over

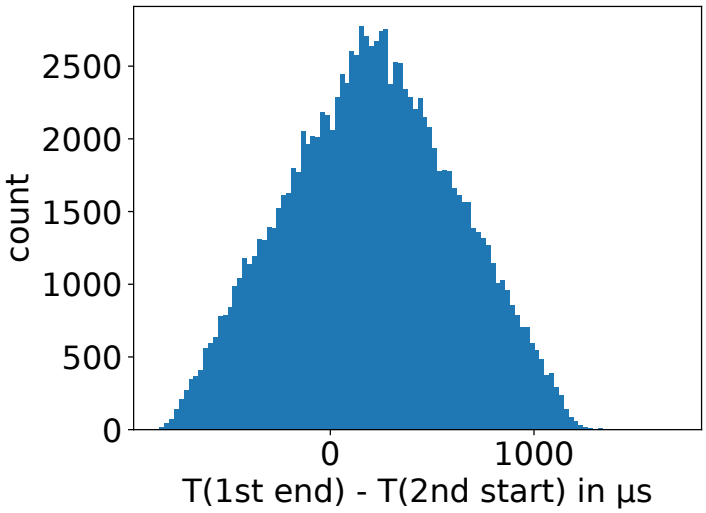


Figure 4.8: Distribution of time overlap recorded in Jaeger for two sequentially invoked RPCs. A positive value shows an overlap. The mean is 204.21μs and the max is 1696.00μs.

118K traces. A positive value shows overlap and a negative value shows non-overlap. More than 50% of samples show an overlap. The P50 overlap is 204μs and the maximum overlap is 1696μs.

Based on this empirical observation, we tuned the `happensBefore(A, B)` part of our CP algorithm with the following relaxation:

- $A_{end} - threshold < B_{start}$, and
- No other children of the parent of P of A can start or end in the overlapped time range

The first condition allows a small `threshold` amount of overlap between the end of the previous span with the start of the next span. The second condition ensures that in the region of the allowed overlap, there is no other spawn and sync event, which ensures the parent-child serialization. The threshold is set to $1ms$.

Span overflow and underflow problems: In addition to the overlap, there can be overflow and underflow of child spans due to the clock drift. We enumerate these problems along with our pragmatic solutions below:

- A child span C may start before the start of the parent span P. In such cases, we truncate the start time of C till the start time of P. This may involve the recursive truncation of C's descendants.
- A child span C may end after the end of the parent span P. In such cases, we truncate the end time of C to the end time of P. This may involve the recursive truncation of C's descendant.
- Although rare, a child span C may end before the start time of parent span P. Similarly, a child span C may start after the end time of the parent span P. In these cases, we completely drop the subtree formed by C for CPA.

This tailoring fixed our CP algorithm. The total time truncation over millions of traces was under 5% giving us the confidence that a significant part of the data was retained.

4.5.3 Aggregating Critical Paths

While one trace can be compressed into its essential critical path and represented as a CCCT, it may not be representative. Hence, we need to inspect numerous traces to derive a "typical" shape of the critical path. Distinct traces may exhibit different critical paths based on many things, such as calling parameters, scheduling decisions, system load, time of the day, and network delays, to name a few. Hence, a *summary* of *typical* components on the critical path is desired.

To this end, we merge all critical paths (represented as CCCTs) into a weighted, *aggregate CCCT*. We follow the tree merging process done in HPCToolkit [195]. The aggregate CCCT succinctly summarizes all call paths leading to critical path nodes in all traces; it captures the quantitative aspect by associating higher weights to those call paths that are often on the critical path. The weights of the nodes in such a tree would be the summation of the weights of the constituent call paths. Specifically, we provide

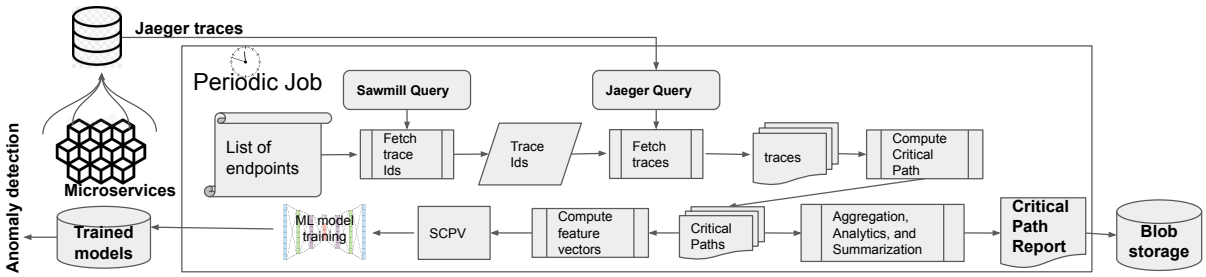


Figure 4.9: Schematic diagram of CPA over Jaeger traces.

different percentiles (e.g., P50, P95, P99) of the latency values, which are widely used for QoS purposes. Figure 4.5 exemplifies this process.

4.5.4 Workflow for Continuous CPA

Figure 4.9 depicts the workflow followed by CRISP for performing critical path analysis of microservice traces for all endpoints. The components belonging to CRISP are marked by the outermost rectangular box.

All services are instrumented to produce Jaeger traces during their RPCs. The instrumentation is enabled across languages such as Go, Java, Node.JS, and Python. The RPCs emit Jaeger spans into a common data store, which can be queried via SQL-style queries.

The CRISP workflow runs as a daily job. The workflow begins by collecting a list of endpoints. Each endpoint can be handled in parallel. Hence, we dedicate a handful of machines that shard the list of endpoints among them.

For each endpoint, CRISP queries the Jaeger data store (via `sawmill-query`) service to fetch a list of traceIDs. This query is set up to obtain the last two weeks' worth of traces. We then use these traceIDs to fetch the actual JSON traces (`jaeger-query`) service. We exploit IO parallelism here to fetch many traces concurrently. We compute the critical path over each trace in parallel using the map-reduce paradigm. The set of criti-

cal paths obtained is fed into an aggregating process that summarizes and produces the daily critical path report for each endpoint (top-down analysis) and also produces overall metrics aggregated over all endpoints (bottom-up analysis). The results are injected into blob storage that can be easily navigated by a varied set of users, including service owners, performance engineers, and capacity managers. An offline anomaly detection model is also trained per endpoint result.

4.6 CRISP Features

We have developed tools to inspect critical paths for top-down performance analysis of specific endpoints, bottom-up analysis over all endpoints, and automatic anomaly detection over traces. We describe these features in this section.

4.6.1 Top-Down Analysis

We store the results of our CPA for each endpoint into profiles for investigation by service owners. CRISP provides the following means of visualization of CPA over each endpoint.

Flame graph: Flame graph [191] is a powerful way to visualize hierarchical call paths arising from profiling. The interactive visualization is easier to digest and investigate. Since we maintain the summarized critical paths as aggregate CCCTs, which are formed of many weighted call paths, it naturally avails itself to be represented as a flame graph.

If we chose all traces to represent a single flame graph, the critical path found in P99 latencies may dominate the flame graph and mask the other common cases. For that reason, we show three different flame graphs for different percentiles of latency values (e.g., P50, P95, and P99). We also produce differential flame graphs [212] that show how

P50(E) ⬍	trace 1	trace 2	trace 3	trace 4	trace 5	trace 6	trace 7	trace 8	trace 9	trace 10
totalTime	3.58e+06	1.37e+07	1.31e+07	1.28e+07	1.21e+07	1.21e+07	1.21e+07	1.03e+07	9.39e+06	9.06e+06	8.99e+06
Service::Operation139	2.13e+06	0.00e+00	0.00e+00	0.00e+00	0.00e+00	9.86e+06	0.00e+00	8.10e+06	3.23e+06	4.36e+06	5.39e+06
Service::Operation2	1.96e+05	2.22e+05	1.34e+06	1.66e+06	1.44e+05	1.54e+05	5.20e+04	7.40e+04	4.02e+06	2.07e+05	1.30e+06
Service::Operation6	2.74e+05	1.43e+05	5.23e+04	2.67e+05	5.00e+04	5.38e+05	6.70e+05	5.48e+05	2.73e+05	2.72e+05	9.40e+04
Service::Operation3	2.33e+05	3.08e+05	1.69e+05	2.69e+05	2.01e+05	2.01e+05	2.54e+05	4.11e+05	2.77e+05	2.24e+05	1.91e+05
Service::Operation7	1.02e+05	1.44e+06	1.53e+04	1.41e+05	9.68e+04	1.30e+05	1.06e+05	9.25e+04	1.32e+05	5.09e+04	1.03e+05
Service::Operation1	7.37e+04	3.17e+05	0.00e+00	8.41e+04	2.49e+05	6.97e+04	8.18e+04	6.66e+04	8.09e+04	0.00e+00	3.97e+05
Service::Operation9	6.13e+04	1.35e+05	6.08e+04	5.10e+04	9.81e+04	6.63e+04	5.29e+04	7.77e+04	5.99e+04	6.94e+04	6.98e+04
Service::Operation142	6.51e+04	0.00e+00	0.00e+00	6.33e+04	8.90e+04	4.26e+04	8.19e+04	5.27e+04	5.28e+04	7.04e+04	5.69e+04
Service::Operation4	5.60e+04	1.09e+05	0.00e+00	5.63e+04	1.05e+05	5.71e+04	5.68e+04	5.29e+04	4.95e+04	0.00e+00	1.06e+05

Figure 4.10: Example heat map from 1000 traces. The result is sorted by the P50 percentile value of the exclusive time of each operation. Each cell is the accumulated time in μs.

the critical paths change between two percentile values.

Heat map: Flame graphs are useful for navigating call chains but developers sometimes need access to an actual Jaeger trace that represents a given data so that they can inspect it in further detail. For this reason, CRISP provides the heat map view (see Figure 4.10), where the rows are the endpoints and the columns represent individual traces. Each cell in the heat map represents the exclusive time on the critical path and each cell is gradient colored based on its contribution (exclusive time) to the total latency. In this view, we collapse the call paths and accumulate the metrics from all call paths, reaching the same endpoint in a single row. However, for exploration, the developers have access to the top 5 call chains (not shown) for each endpoint, which is available by hovering over any row. In this view, the user can also choose percentile values and inclusive or exclusive metrics to sort the rows. Each column is also sorted by a high to low contribution for a given chosen metric. Selecting any trace takes the user to the Jaeger-UI to inspect the trace.

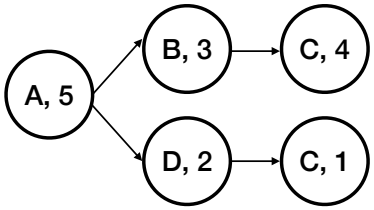
4.6.2 Bottom-Up Analysis

The objective of the bottom-up analysis is to derive insights from *all* endpoints and to bubble up those interior APIs improving which will improve many endpoints. The bottom-up analysis is a data-intensive process and needs access to critical paths from all endpoints. For this reason, we retain the aggregate CCCT computed for each endpoint from the top-down process, along with some additional statistics related to the overall graph structure. Once all endpoints are processed, the bottom-up analysis runs; it aggregates the statistics from each endpoint and quantifies the impact of each API over all other endpoints. The output of the bottom-up analysis is a descending priority list of top APIs that are often in many endpoints. Additionally, the bottom-up analysis produces various histograms over all traces taken together, which include the total number of times any API appears in any graph, the total number of times an API appears on the critical path, the number of unique APIs on the critical path, the critical path length, and the maximum degree of concurrency in a trace, among others. These graphs are intended to inform infrastructure and hardware engineers to better understand the current needs of our systems and aid capacity planning for the future.

4.6.3 Anomaly Detection

We also employ CRISP to pinpoint whether a new incoming trace (for a given endpoint) deviates from the normal execution behavior. For this purpose, we have trained a machine learning model and used it for inference.

During the offline training, we encode the critical path (CCCT) for each trace of an endpoint into feature vectors, which we call service critical path vectors (SCPV). We feed several SCPVs into an autoencoder to learn the normal execution pattern of the given service. During the online inference, the learned model will infer whether the given new



A	5
A→B	3
A→B→C	4
A→D	2
A→D→C	1

Figure 4.11: An example CCCT (left), the letters indicate name and the numbers indicate the exclusive time on the span. The corresponding SCPV (right).

trace is abnormal or not based on an anomaly score.

The architecture design, training, and inference of the autoencoder are derived from TraceAnomaly [192], which is the state-of-the-art framework for anomaly detection in microservices trace. The neural architectural details are described in the next subsection. The key difference between CRISP and TraceAnomaly is in the data encoding. TraceAnomaly uses a service trace vector (STV) which encodes every path in the trace and, in contrast, CRISP encodes only on the call paths for those spans that are on the critical path spans. **SCPV encoding:** Figure 4.11 exemplifies encoding the critical path present as a CCCT into an SCPV. For each node in CCCT, it assigns weights based on its exclusive execution time. Notice that endpoint C occurs twice on the critical path, thus it is also encoded twice in the SCPV, given the call chain is different. The training set is a 2D matrix where each column is a feature (call path) and each row is the feature values of a given trace. Using the call paths of spans only on the critical path, compared with the prior work that used all call paths in the entire graph, offers significant benefits. It reduces the feature dimensions; it reduces the training and inference time; and, most importantly, it improves the model accuracy. The impact of the CCCT-based encoding is substantial and evaluated in Section 4.7.3.

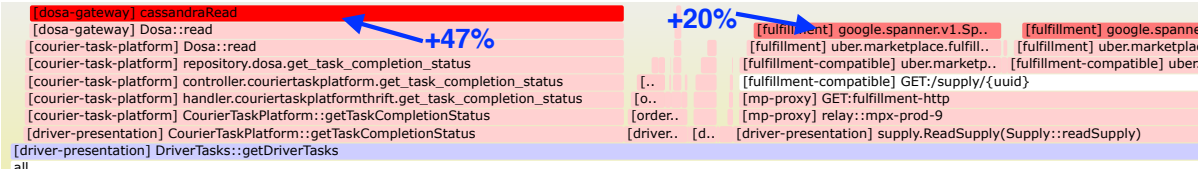


Figure 4.12: Differential flame graph for the `getDriverTask` endpoint. Red colors indicate the growth from P50 critical paths to P95 critical paths.

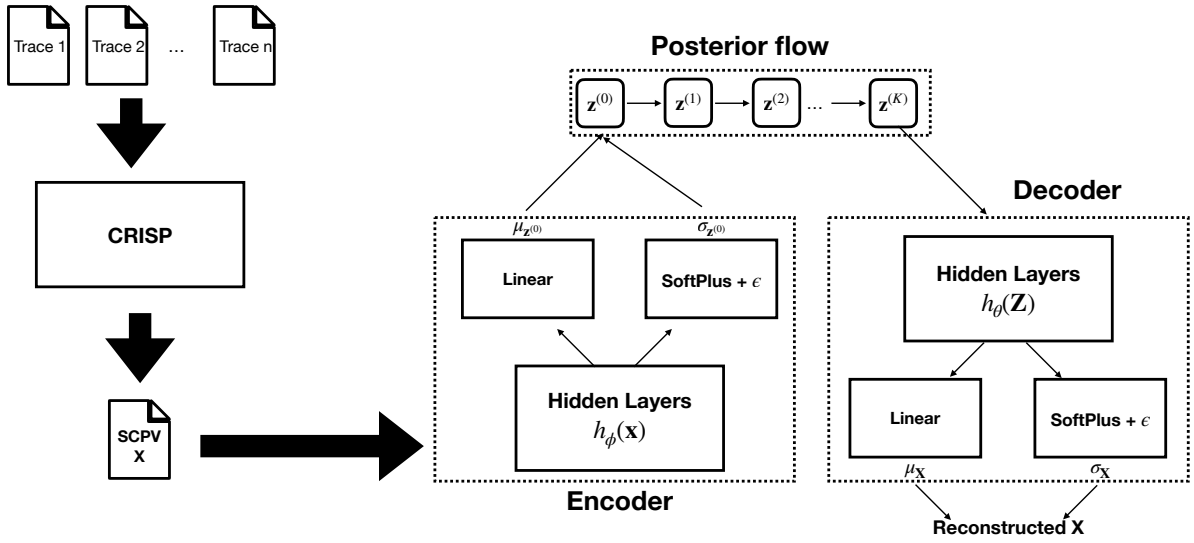


Figure 4.13: Architecture of neural network for anomaly detection.

4.6.4 Autoencoder Model Architecture

We choose the Deep Bayesian Network for anomaly detection given it is capable of learning complex patterns from the trace. We adopt the model from TraceAnomaly [192], which is the state-of-the-art framework for microservice trace based anomaly detection. Specifically, we adopt Variational Auto-Encoder (VAE) [213] to model the distribution pattern from the normal execution. VAE is an unsupervised learning that does not require a label, which can be expensive to obtain in our setting due to the volume of traces. Figure 4.13 depicts the architecture of VAE. It has three components: encoder, posterior flow, and decoder.

The encoder contains 1 hidden layer ($h_\phi(\mathbf{x})$) to learn the hidden features of SCPV.

The goal is to learn the mean $\mu_{\mathbf{z}^{(0)}}$ and the standard deviation $\sigma_{\mathbf{z}^{(0)}}$ of the SCPV. $\mathbf{z}^{(0)}$ is sampled from diagonal Gaussian $\mathcal{N}(\mu_{\mathbf{z}^{(0)}}, \sigma_{\mathbf{z}^{(0)}}\mathbf{I})$ and served as the latent variable to fit the distribution. ϵ is a small constant vector that has been introduced to avoid numerical issues during the training [192]. SoftPlus is defined as $\text{SoftPlus}(\mathbf{x}) = \log(1 + \exp(\mathbf{x}))$.

For the next step, posterior flow allows the network to learn more complex patterns of the trace. The input is $\mathbf{z}^{(0)}$ and after passing length of K flow it will become as $\mathbf{z}^{(K)}$.

Then, $\mathbf{z}^{(K)}$ will be passed into the decoder network to extract hidden features. Similarly, the purpose of those hidden features is to derive the mean $\mu_{\mathbf{x}}$ and standard deviation $\sigma_{\mathbf{x}}$ of the input trace vector. After that, the reconstructed $\mathbf{x}$ will be sampled from $\mathcal{N}(\mu_{\mathbf{x}}, \sigma_{\mathbf{x}}^{\mathbf{2}}\mathbf{I})$

4.6.5 Inference

When a new trace is given, the log-likelihood value will be computed against the model to detect whether the trace is abnormal or not. If the trace $\mathbf{x}$ is significantly different than the normal trace, the value of a trace $\log p_\theta(\mathbf{x})$ is noticeably smaller than the value of the normal traces. Instead of manually setting the threshold of anomaly, we follow the work from Liu et al. [192] and use Kernel Density Estimation (KDE) [214] to learn the distribution of the normal traces log-likelihood. Specifically, we adopt the p-value [215] approach and set the value as 0.001 to check if the probability of the log-likelihood value not following the learned distribution.

If the trace contains any unseen *call chain*, it will be regarded as abnormal. Training is a continuous process since the code evolves and the call paths keep changing over time. We use a sliding window of last 14 days of trace to keep our model up-to-date.

4.7 Experience and Evaluation

In Section 4.7.1, we describe one of our findings by applying the top-down analysis of CRISP at Uber, in Section 4.7.2 we show valuable characteristics of microservices at Uber by applying the bottom-up analysis of CRISP. In Section 4.7.3, we empirically evaluate the anomaly detection capability of CRISP and in Section 4.7.4 we describe how we employed CRISP in guiding future hardware selection to reduce tail latency in our services.

4.7.1 Tail Latency Investigation via Top-Down Analysis

`getDriverTasks` is a business-critical endpoint in the driver-presentation service responsible for returning the task plan that a driver needs to perform. A sample task plan could be: passenger mask check, pickup passenger, pickup food, drop off passenger, and drop off food. This endpoint assembles the task plan and enriches it by calling numerous other microservices such as courier-task-platform. Figure 4.12 shows the *differential* flame graph for the `getDriverTask` endpoint. The graph plots a difference between the critical paths seen in the traces with the P50 latency vs. P95 latency for the `getDriverTask` endpoint. The red-colored boxes show the growth in percentage time spent in P95 with regards to P50. The `getTaskCompletionStatus` API was absent in the P50 traces, whereas it occupies 47% of the total execution in P95 traces, contributing to the same amount of addition to the tail latency. This endpoint dependency makes a call to Cassandra—an expensive database read. Based on this insight from CRISP's differential flame graph views, we identified the root cause of performance variance and high tail latency. We recommend caching with timestamp filtering optimization as opposed to a database read to reduce the tail latency.

Trace processing overheads: Table 4.1 shows the overhead of analyzing the

`getDriverTasks` endpoint discussed in this section running on 16 cores of an Intel Xeon Skylake machine clocked at 2.4 GHz.

Table 4.1: Overhead of top-down analysis of `getDriverTasks`.

Num Traces	Trace size	Processing time	Memory usage
10k	6.8 GB	48 sec	2.1 GB
20k	14 GB	109 sec	4.2 GB
40K	28 GB	232 sec	8.5 GB
80K	56 GB	553 sec	17.6 GB

Sparse sampling vs. quality of CPA: We observed that the sampling rate does not qualitatively affect the aggregate critical path results. We conducted an experiment where we first produced an aggregate critical path from 1 million traces. We also produced critical paths from randomly sampled 100K and 10K traces from the same data set. We noticed that the attribution of the top 20 services on the critical path, whether for 10K or 100K samples, was essentially the same as the one produced from 1M traces.

4.7.2 Systemic Insights via Bottom-Up Analysis

In this section, we show the result of running CRISP with bottom-up analysis on the collected trace dataset and some insight associated with the data. The dataset includes more than 1 million traces, ∼4k services, and ∼40k endpoints. It takes around 4 hours on 32-cores of a Intel Xeon Skylake machine clocked at 2.4 GHz.

Total RPCs per request: Figure 4.14 is a histogram of the total number of RPCs made per request, which is same as the total number of spans in a trace. On average there are 112 spans in a trace. However, there exist several large ones with a maximum of 275K spans. Such scale brings significant challenges for the developer to debug without proper reduction of the graph size.

Total endpoints in a trace: Figure 4.15 is a histogram of the total number of

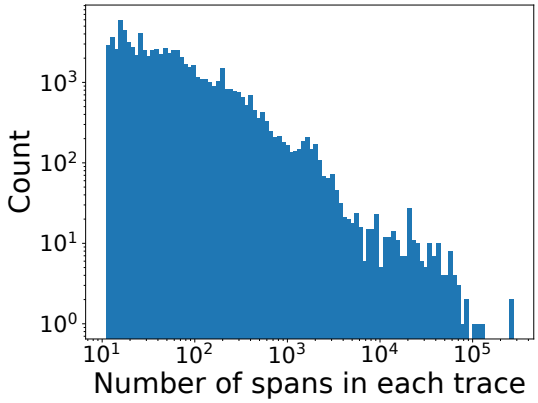


Figure 4.14: Histogram of the number of spans per trace.

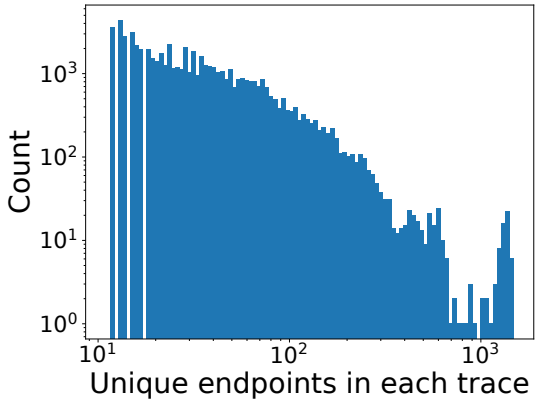


Figure 4.15: Histogram of number of unique endpoints per trace.

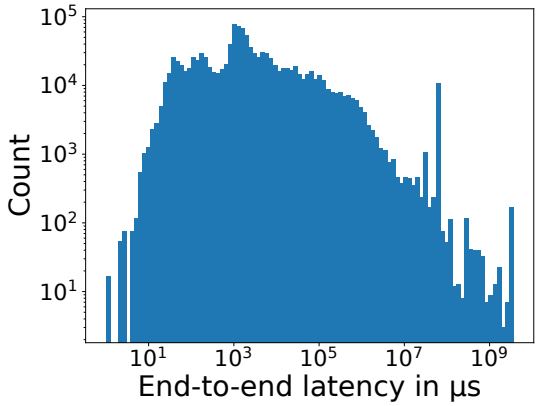


Figure 4.16: Distribution of latency among all traces.

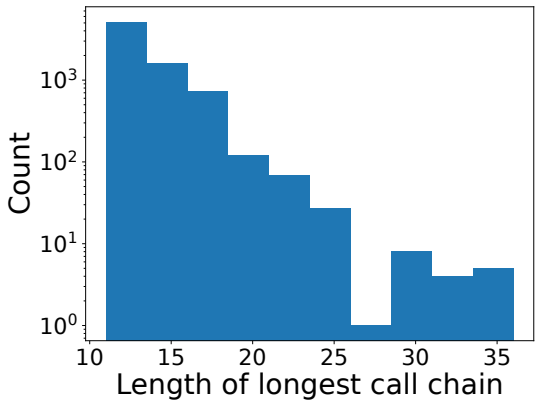


Figure 4.17: Histogram of longest call chain per trace.

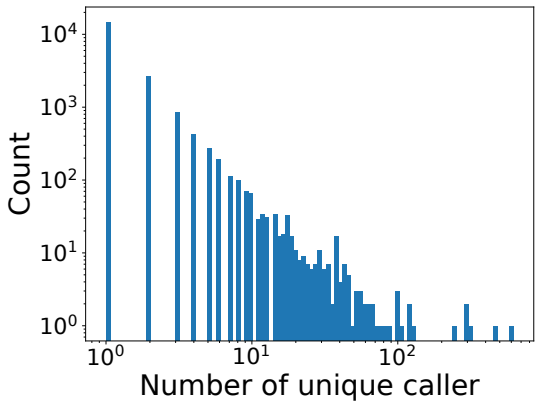


Figure 4.18: Histogram of the number of unique caller for each endpoint.

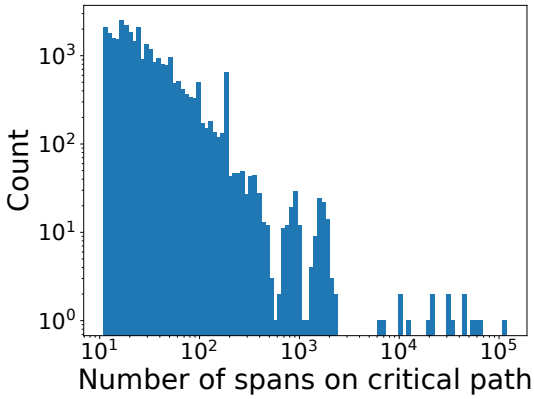


Figure 4.19: Histogram of the number of spans on the critical path per trace.

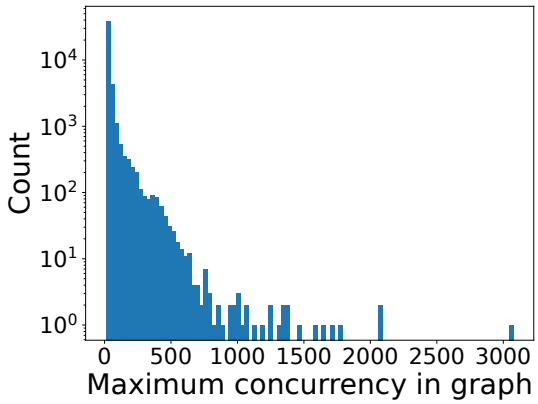


Figure 4.20: Histogram of the degree of the concurrency (max no. of overlapping spans) per trace.

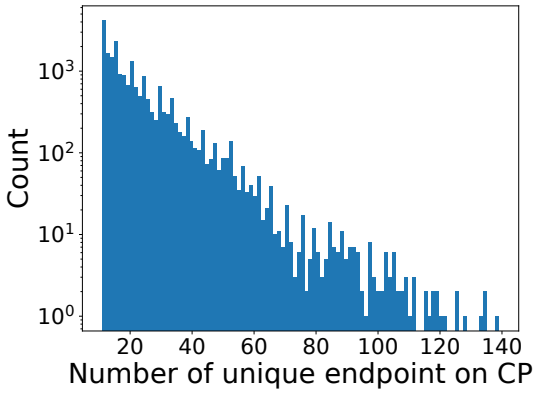


Figure 4.21: Histogram of the number of unique endpoints on the critical path per trace.

unique endpoints found in each trace. At most each trace has 1400 unique endpoints.

Latency distribution: Figure 4.16 plots the histogram of latencies observed in each of $\sim 1M$ traces. The tail is several orders of magnitudes longer than the mean or median.

RPC depth: Figure 4.17 is a histogram of the longest call chain found in each trace. The depth of the call chain is another measure of the complexity of traces. The average RPC depth is 8.5. The maximum observed depth is 36.

Unique caller: Figure 4.18 is a histogram of the number of the unique callers for each endpoint across one million traces. The number differs wildly as the mean value is just above 2 but the maximum value is 620.

Degree of concurrency: Figure 4.20 is a histogram of the maximum number of spans that overlap in time in each trace. This number gives the degree of concurrency (and hence a measure of the complexity) in our traces. Overall, the microservices show a high degree of concurrency. On average, the degree of concurrency is 21. The degree of concurrency often grows to 100s for more complicated services. The maximum degree of concurrency we observed in $\sim$ 1M traces was 3076.

Total RPCs on the critical path: Figure 4.19 is a histogram of the number of spans on the critical paths, which counts the number of RPCs made on the critical path. Besides a few outliers, the length of the critical path is short. On average, there are 33 RPCs on the critical path (in contrast, the entire graph in Figure 4.14 shows 112-275K RPCs in traces). The short critical path length allows the developer to investigate and debug easily.

Endpoints on the critical path: Figure 4.21 is a histogram of the unique endpoints on each critical path. Compared with the number of endpoints in the entire trace (Figure 4.15), the number of the endpoints on the critical path is an order of magnitude smaller (the maximums are 1400 vs. 140). The 10x size-reduction matches our observation of the 6 services we test for anomaly detection.

Table 4.2: Evaluation results for large online services. Inference time is measured with 1000 traces. (TA*=TraceAnomaly.)

	No. of Unique endpoints	Max no. of spans	No. of callpaths/ features		Training Time		CRISP training speedup	Inference Time		CRISP inference speedup	Precision		Recall	
			STV	SCPV	TA*	CRISP		TA*	CRISP		TA*	CRISP	TA*	CRISP
Service 1	214	1429	5117	1186	70m (GPU)	46m (GPU)	1.52X	2.24s (GPU)	1.21s (GPU)	1.85X	1.0	0.998	0.986	0.992
Service 2	969	1724	9725	1860	100m (GPU)	50m (GPU)	2.00X	3.54s (GPU)	1.40s (GPU)	2.54X	1.0	1.0	0.958	0.984
Service 3	734	5320	20321	2154	150m (GPU)	50m (GPU)	3.00X	5.64s (GPU)	1.36s (GPU)	4.15X	1.0	1.0	0.5	0.982
Service 4	912	20001	25347	2715	1184m (CPU)	56m (GPU) 219m (CPU)	21.14X (GPU) 5.41X (CPU)	56.67s (CPU)	1.56s (GPU) 9.26s (CPU)	36.33X (GPU) 6.12X (CPU)	1.0	1.0	0.928	0.978
Service 5	768	6562	26404	2336	811m (CPU)	51m (GPU) 177m (CPU)	15.90X (GPU) 4.58X (CPU)	42.90s (CPU)	1.36s (GPU) 5.81s (CPU)	31.54X (GPU) 7.38X (CPU)	1.0	0.998	0.5	0.982
Service 6	1477	10992	28968	1151	1305m (CPU)	46m (GPU) 148m (CPU)	27.77X (GPU) 8.82X (CPU)	78.88s (CPU)	1.18s (GPU) 4.48s (CPU)	66.85X (GPU) 17.61X (CPU)	1.0	1.0	0.912	0.977

4.7.3 Empirical Analysis of Anomaly Detection

Here, we will evaluate CRISP's anomaly detection on six critical endpoints.

Methodology: We collect traces for six microservices in real production over a 14-day period. The training data for each case includes 20,000 traces and the testing data has 500 unseen traces for normal and abnormal data. To generate abnormal inference data, we drop 20% of the nodes in the graph and randomly shuffle the duration of the nodes as described in [192, 216, 217]. We did not use real anomalous traces for evaluation since we do not have a large number of labelled anomalous traces (i.e., we have a lot of false negatives). Also, the labeled data contains false positives and coordinating with hundreds of developers to verify the veracity of labeling is non-trivial. We use TraceAnomaly [192] as the baseline against which we compare our results. We adopt the same architecture of the autoencoder and reuse their code. The main difference is that we use CRISP to preprocess the trace before feeding it into the autoencoder so that only paths appearing on the critical path information are included. A fundamental assumption is that any noticeable difference in the trace must impact the critical path.

Hardware: We use two machines in our evaluation: a CPU-only machine with 256 GB memory and a CPU+GPU machine with 128 GB memory. Most of the experiment is done on a machine with GPUs. It has 2 Quadro RTX 5000 GPUs and 2 socket Intel Xeon Gold 5218 CPU at 2.30GHz. The CPU machine has 2 sockets with Intel Xeon

Silver 4214 CPU at 2.20GHz. Both machines run on Linux 4.14. The reason to use two machines is that for some experiments, the training data for TraceAnomaly cannot fit the GPU memory, whereas CRISP's training data always fits on GPU memory. In such cases, for a fair comparison, we also run the experiment on the 256 GB CPU-only.

Table 4.2 shows the empirical evaluation results of anomaly detection on 6 large online services at Uber. It captures the essential features such as the number of RPCs, unique endpoints, and call path diversity in these services. It also shows the training and inference time with both STV (prior art from TraceAnomaly) and SCPV (our work) data. Finally, the last 4 columns present the model accuracy in terms of precision and recall. In summary, using critical path via CRISP reduces the training time and inference time and improves the recall performance on top of the state of the art. **Training speedup:** From the table, we can observe that CRISP offers up to 22× speedup for training compared with TraceAnomaly. Even the smallest speedup is more than 50%.

The reason for the speedup is that the training data from CRISP (SCPV) is one magnitude smaller than TraceAnomaly (STV) up to 25× for Service 6. The number of unique call paths on the critical path is significantly smaller than the total number of call paths in the entire graph (also see Figures 4.14-4.21). Furthermore, when the number of the trace and the dimension of the feature vector is large, the size of the training data of TraceAnomaly can easily exceed the memory of the GPU, which makes it unable to train. For such cases (Service 4, 5, and 6), we can still see more than 4× speedup even if we train both TraceAnomaly and CRISP on CPU machines. When CRISP is trained on the GPU machine, the speedup can easily exceed 15×. The faster training allows for more practical deployment. **Inference speedup:** Similar to training speedup, the reduction in inference data size leads to a faster inference of CRISP. The smallest speedup is more than 1.85× whereas the largest speedup is over 66×. This lower latency allows us to batch many inferences together to exploit GPU throughput.

Precision: From Table 4.2, we can see that both TraceAnomaly and CRISP are capable of detecting the abnormal trace accurately. Autoencoders are capable of capturing the complex pattern of the graph. TraceAnomaly works slightly better than CRISP on 2 services, but overall accuracy is very high for both methods.

Recall: The recall is the part that differentiates the quality of results between TraceAnomaly and CRISP. Recall measures how many of the actual positives the model captures through labeling it as positive, (i.e., $\frac{True\_Positive}{True\_Postive+False\_Positive}$). When the recall is closer to 1, it indicates that the model makes fewer false-positive predictions (an anomaly in this case). From Table 4.2, it is clear that CRISP outperforms TraceAnomaly by a noticeable margin. Particularly for Service 3 and 5, half of the positive prediction of the anomaly is false, meaning all normal traces for inference are labeled abnormal by TraceAnomaly. To make sure the prediction is actually incorrect, we asked the service owners and verified that the normal inference testing traces are not showing any abnormal behaviors. On the contrary, CRISP's recall is close to 1. For Service 1 and 2, the performance of CRISP is slightly better than TraceAnomaly, as both models make relatively accurate predictions. CRISP shows more than 5% improvement for Service 4 and 6.

CRISP produces superior results on services with a large number of call paths. For instance, there are 912 endpoints in Service 4 but the total call paths is 25,347. Since there is more diversity among the shapes of the call chains on the entire graph, the SCPV encoding fails to capture its full variety; consequently, unseen call paths easily trigger a false positive in TraceAnomlay. In contrast, the critical path remains fairly stable when trained over a large corpus of traces, and consequently CRISP has fewer false positives.

4.7.4 CPA in Hardware Selection

In addition to the parent-child transitive relationships and times, Jaeger traces also contain additional information, such as the hostname on which the span was executed. Uber's data center consists of diverse hardware CPU SKUs. Services can be installed on different hardware versions. Hence, an API may run on different hardware on different requests.

We collected the critical path for one of our important services using CRISP and identified that a downstream operation was on the critical path. We further clustered the samples from the profiles by the CPU versions on which they were running. The violin plot in Figure 4.22 shows how the latencies vary on 2 prominent CPU SKUs: Intel Xeon Silver 4212 running at 2.2 GHz (SKU-A) and Intel Xeon Silver 4212R running at 2.4 GHz (SKU-B).

The two SKUs are identical (same vendor, microarchitecture, cache size, etc.) with the only exception being that their CPU clock speeds are different. This mild (9%) difference in the clock speed has a profound impact on the behavior of the plotted service. The P50 value for SKU-A is 15% higher than that on SKU-B. Moreover, the tail latency on SKU-A is 1.5x higher than the one on SKU-A. To summarize, a slightly faster CPU clock proves to have a significant impact on reducing the tail latency and overall latency. This difference has a significant impact on the overall capacity allocation since tail latency (e.g., P95) is often used in capacity allocation. This observation demands further, systematic investigation into classifying critical path components as CPU SKU sensitive vs. insensitive; also, such categorization helps data center-wide microservice schedulers to favor SKU-sensitive services on the critical path onto the SKUs where they exhibit superior performance.

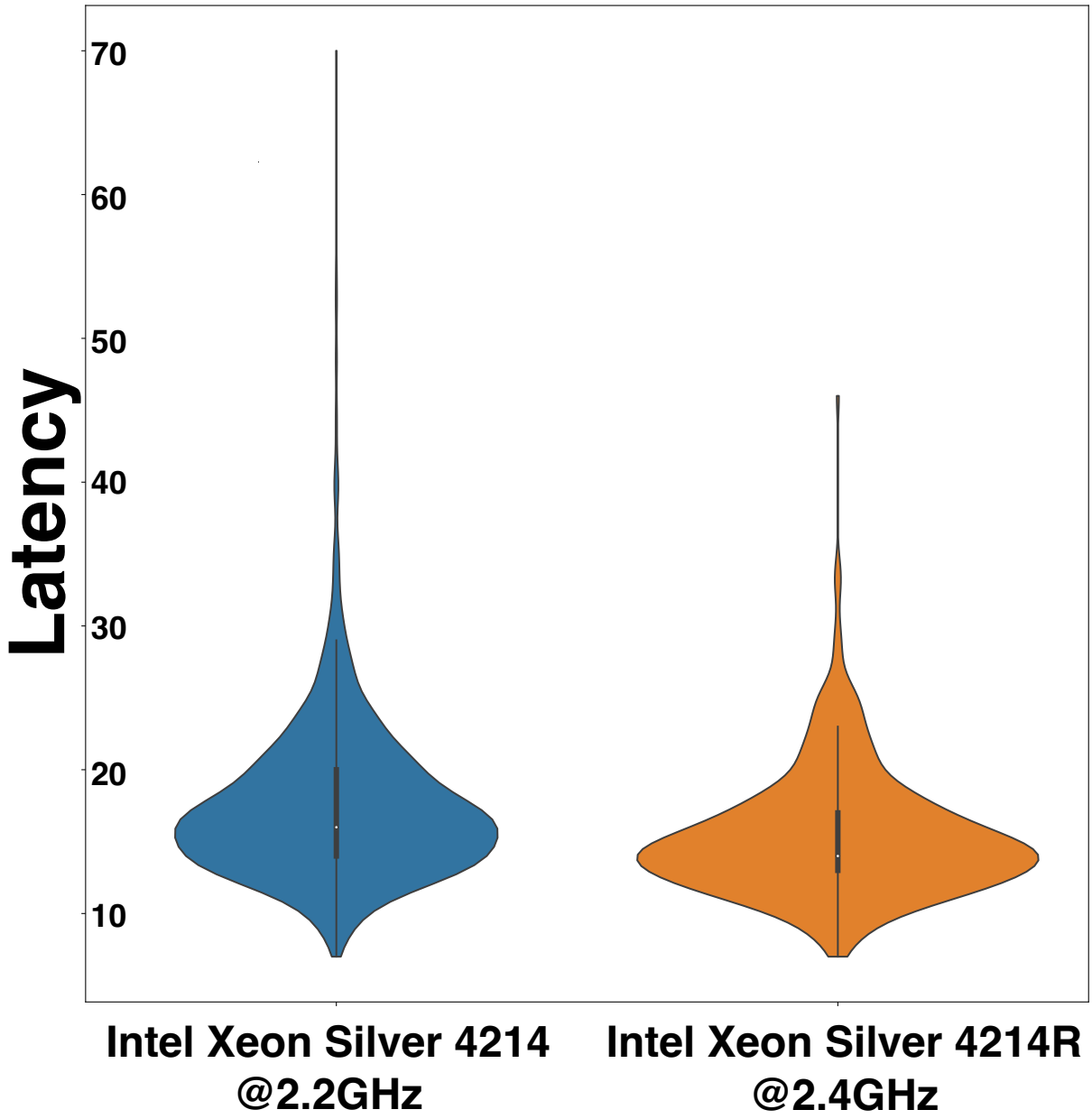


Figure 4.22: Violin plots of the exclusive execution time of a critical path operation with two different CPUs. The latency is in μs.

4.8 Related Work

Critical Path Analysis (CPA) has been extensively explored in the shared-memory parallel programming paradigm [188, 206, 218, 195, 219, 220, 221, 222, 223] but less explored in distributed parallel systems. Unlike shared-memory and structured parallel programs, microservices use distributed parallel computing environments and are unstructured in nature.

Barford and Crovella [224] utilize critical path analysis for profiling and understanding TCP transactions and improve data transfer latency in web applications; however their scale is significantly smaller than the 4K services deployed over millions of CPU cores that we handle. Bohem et al. [225] employ tracing and CPA for MPI programs in HPC environments; this approach has not been employed in microservice environments. Kaldor et al. [226] develop an end-to-end tracing system (Canopy) for tracking requests from web-browsers/mobile to backend services; it handles billions of traces. A distinguishing feature of CRISP compared with Canopy is the use of CPA, which significantly reduces the data needed for analysis.

Qiu et al. [217] propose a fine-grained resource management framework based on microservice traces using CPA. They employ the insights for scheduling and other resource management to reduce CPU utilization. However, their work does not cover industry-scale deployment; they also do not facilitate performance bug or anomaly detection and cannot provide bottom-up system-wide performance insights.

Fields et al. [207] explore a hardware predictor to analyze the criticality of instructions by using CPA and use it to guide dynamic instruction scheduling. Venkataramani et al. [223] propose Global Critical Path (GCP) to predict system-level performance and optimize the performance of highly concurrent self-timed circuits. These approaches rely on the precise last arriver information, which is readily available in these cases. Our

critical path computation in microservices also depends on knowing the last arriver. Unlike the aforementioned approaches, we do not have direct access to the last arriver in our distributed system. As a result, we need to use clock information from different hosts and adjust for clock skew to heuristically infer the last arriver.

Multiple tools have been developed to profile and debug large distributed and parallel systems. `lprof` [227] constructs request flow from logs and it is as good as the quality of logs; it has not been evaluated on microservices; it also does not provide CPA and hence suffers from a voluminous noisy data. Mace et al. [228] developed Pivot as a dynamic, extensible tracing system for inter-operating applications. Pivot employs a happen-before relationship between events to establish causality. Pivot does not build a critical path and hence pays equal attention to any causal relationship unlike CRISP. Chow et al. [229] build a system that utilizes a large number of request traces to validate hypotheses about causal relationships. Edgar [230] provides a summarized view of request traces, logs, and metadata in distributed systems. It does not employ sophisticated analyses or automated anomaly detection.

Several works have focused on microarchitectural aspects of microservices [231, 232, 233, 234, 36]. Most of these works are focused on how microservices utilize microarchitectural features, but ignore the end-to-end user request; in contrast, CRISP takes a higher-level approach and looks at the entire flow of requests through a chain of services.

Multiple works have studied anomaly detection in distributed systems. Liu et al. [192] use Deep Bayesian Network to detect the performance anomaly in an unsupervised manner. They utilize machine learning to learn the normal behavior pattern of the given dependency graph and try to detect the anomaly online. Gan et al. [216] propose a root cause analysis system for large-scale microservices using machine learning. The system uses Conditional Variational Autoencoders (CVAE) [235] to automatically generate the counterfactual training data. These approaches have used the entire call graph, leading

to significant training and inference time. In contrast, CRISP uses only the critical path(s), leading to dramatic speedups while producing higher quality results.

4.9 Conclusions and Future Work

Microservices are the preferred architecture choice in modern service-oriented software systems. Large-scale microservices have tens of thousands of endpoints with complex, nested, and asynchronous. Prior work in profiling microservices has either focused on tracing techniques, which produce a lot of data, but lack in delivering insights, or on micro-architectural optimization within a service, ignoring the full picture of the life of a request through myriad services. This work develops a tool, CRISP, which uses critical path analysis (CPA) over RPC traces to bubble-up interesting activities and discard noisy events. CRISP provides rich developer insights both for service owners and infrastructure engineers. In a short three-month deployment period, CRISP's analyses have sifted over 4,000+ services, 40,000+ endpoints, hundred of millions of traces, and tens of terabytes of data at Uber; as a result, CRISP has bubbled-up profiling results that helped developers understand and optimize important services. Employing the critical path, as opposed to the whole RPC trace, speeds up the training of models and on-the-fly inference for anomaly detection while also producing noticeably higher quality results.

Our future work involves enhancing CRISP to address other use cases such as setting the TTL values for downstream calls and bubbling up those downstream services that often return errors. We plan to expand our anomaly detection to include developers in the loop and improve traces with labelled data.

Chapter 5

Conclusion

In this thesis, we present how to utilize performance-driven analysis and optimizations to investigate communication paradigms and offer concrete improvement in three different scenarios.

For HLE, we present a source-to-source transformation framework, GOCC, that consumes lock-based pessimistic concurrency programs in the Go language and transforms them into optimistic concurrency programs that use Hardware Transactional Memory (HTM). GOCC performs rich inter-procedural program analysis to detect and filter lock-protected regions and performs AST-level code transformation of the surrounding locks when profitable. Profitability is driven by both static analyses of critical sections and dynamic analyses via execution profiles. A custom HTM library, using perceptron, learns concurrency behavior and dynamically decides whether to use HTM in the rewritten lock/unlock points. Given the rich history of transactional memory research but its lack of adoption in any industrial setting, we believe this workflow, which ultimately produces source-code patches, is more apt for industry-scale adoption. Results on widely adopted Go libraries and applications demonstrate significant (up to 10×) and scalable performance gains resulting from our automated transformation while avoiding major

performance regressions.

Inspired by GOCC and to better facilitate application performance programming and certain runtime optimization, we propose a software optimization strategy enabled by a novel low-latency Prediction System Service (PSS). Rather than relying on nuanced domain-specific knowledge or slapdash heuristics, a system service for prediction encourages programmers to spend their time uncovering new levers for optimization rather than worrying about the details of their control. The core idea is to write optimizations that improve performance in specific cases, or under specific tunings, and leave the decision of how and when exactly to apply those optimizations to the system to learn through feedback-directed learning. Such a prediction service can be implemented in any number of ways, including as a shared library that can be easily reused by software written in different programming languages, and opens the door to both new software optimization patterns and hardware design possibilities. As a demonstration of the utility of this approach, we show that three very different application-targeted optimization scenarios can each benefit from even a very straightforward perceptron-based implementation of the PSS as long as the service latency can be held low. First, we show that PSS can be used to more intelligently guide hardware lock elision with resulting speedups over a baseline implementation by 34% on average. Second, we show that a PSS can find good configuration parameters for PyPy's Just-In-Time (JIT) compiler resulting in 15% speedup on average. Last, we show PSS can guide the page reclamation task within a kernel memory management subsystem to reduce the average memory latency by 33% on average. In all three cases, this new optimization pattern with service support is able to meet or beat the best-known hand-crafted methods with a fraction of the complexity.

For microservices analysis at datacenter, we present CRISP — a tool to perform critical path analysis (CPA) over a large number of traces collected from RPCs in microservices environments. CRISP provides three critical performance analysis capabil-

ities: a) a *top-down* CPA of any specific endpoint, which is tailored for service owners to drill down the root causes of latency issues, b) a *bottom-up* CPA over all endpoints in the system — tailored for infrastructure and performance engineers — to bubble up those (interior) APIs that have a high impact across many endpoints, and c) an on-the-fly anomaly detection to alert potential problems. We have applied CRISP's capabilities on Uber's entire backend system composed of ∼40K endpoints that cater to real-time requests from more than 100 million active daily users worldwide. Using the critical path as the basis of performance analysis has a) helped us identify five performance issues and optimization opportunities across two business-critical microservices, b) guided us in our future hardware choice that reduces end-to-end latencies, and c) reduced the false positives in anomaly detection by up to 50% while speeding up the training and inference by up to 28× and up to 67×, respectively, over the state of the art.

The three chapters illustrate how we provide better performance and easier debugging and developing for the programmers by targeting the communication paradigms from different angles: synchronization mechanism for Gocc, user-kernel communication in PSS, and analysis of complex Remote Procedure Calls (RPCs) in CRISP.

In the future, we believe more opportunities can be explored if we focus on the communication paradigms with proper methods like performance-driven analysis and optimizations. For HLE, more evaluations can be done on real-world applications with precise lock elision and effective runtime control over Mutex and HTM. For PSS, more models and parameter types can be implemented as long they follow the abstract APIs. More runtime optimization scenarios can potentially benefit from machine learning capabilities provided by PSS. For microservices, more analysis and tools can be developed to facilitate the developer debugging and optimizing the microservice architecture. For instance, how to find out the common error message on business-critical microservices can be a valid research direction for the future.

Bibliography

[1] Google, "Effective Go - The Go Programming Language." `https://golang.org/doc/effective_go.html`.

[2] "Why Use the Go Language for Your Project?." `https://nix-united.com/blog/why-use-the-go-language-for-your-project/`.

[3] Google, "Kubernetes." `https://kubernetes.io/`.

[4] Uber, "Tally: A Go metrics interface with fast buffered metrics and third party reporters." `https://github.com/uber-go/tally`.

[5] "Package sync." `https://golang.org/pkg/sync/`.

[6] T. Tu, X. Liu, L. Song, and Y. Zhang, *Understanding Real-World Concurrency Bugs in Go*, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 865–878, Association for Computing Machinery, 2019.

[7] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*, in *Proceedings of the 20th annual international symposium on computer architecture*, pp. 289–300, 1993.

[8] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed., 2012.

[9] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee, *Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough*, in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, (New York, NY, USA), p. 265–274, Association for Computing Machinery, 2008.

[10] D. Dice, O. Shalev, and N. Shavit, *Transactional Locking II*, in *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, (Berlin, Heidelberg), p. 194–208, Springer-Verlag, 2006.

[11] M. F. Spear, M. M. Michael, and C. von Praun, *RingSTM: Scalable Transactions with a Single Atomic Instruction*, in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, (New York, NY, USA), p. 275–284, Association for Computing Machinery, 2008.

[12] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, *McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime*, in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, (New York, NY, USA), p. 187–197, Association for Computing Machinery, 2006.

[13] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, *Early Experience with a Commercial Hardware Transactional Memory Implementation*, *SIGARCH Comput. Archit. News* **37** (Mar., 2009) 157–168.

[14] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, *Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), Association for Computing Machinery, 2013.

[15] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, *Evaluation of Blue Gene/Q Hardware Support for Transactional Memories*, in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), p. 127–136, Association for Computing Machinery, 2012.

[16] R. Guerraoui, M. Kapalka, and J. Vitek, *STMBench7: A Benchmark for Software Transactional Memory*, in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), p. 315–324, Association for Computing Machinery, 2007.

[17] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero, *Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server*, vol. 44, pp. 25–34, 04, 2009.

[18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, *The case for learned index structures*, in *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, (New York, NY, USA), p. 489–504, Association for Computing Machinery, 2018.

[19] X. Chen, S. Li, H. Li, S. Jiang, Y. Qi, and L. Song, *Generative adversarial user model for reinforcement learning based recommendation system*, in *International Conference on Machine Learning*, pp. 1052–1061, PMLR, 2019.

[20] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, J. Ding, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan, *Sagedb: A learned database system*, 2019.

[21] D. T. Nguyen, Z. Lou, M. Klar, and T. Brox, *Anomaly detection with multiple-hypotheses predictions*, in *International Conference on Machine Learning*, pp. 4800–4809, PMLR, 2019.

[22] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, *Tvm: An automated end-to-end optimizing compiler for deep learning*, in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, (USA), p. 579–594, USENIX Association, 2018.

[23] S. Kramer, "The Biggest Thing Amazon Got Right: The Platform." `https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/`, October, 2011.

[24] T. Mauro, "Adopting Microservices at Netflix: Lessons for Architectural Design." `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`, Feb, 2015.

[25] Y. Goldberg, "Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture." `https://www.infoq.com/presentations/scale-gilt`.

[26] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. O'Reilly Media, Inc., 1st ed., 2016.

[27] 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 *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), p. 37–48, Association for Computing Machinery, 2012.

[28] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, *Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud*, in *2015 10th Computing Colombian Conference (10CCC)*, pp. 583–590, 2015.

[29] A. Gluck, "Introducing Domain-Oriented Microservice Architecture)." `https://eng.uber.com/microservice-architecture/`.

[30] Y. Zhang, D. Meisner, J. Mars, and L. Tang, *Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference*, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 456–468, 2016.

[31] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, *Exploiting Heterogeneity for Tail Latency and Energy Efficiency*, in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 625–638, Association for Computing Machinery, 2017.

[32] C. Delimitrou and C. Kozyrakis, *Amdahl's Law for Tail Latency*, *Commun. ACM* **61** (jul, 2018) 65–72.

[33] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, *Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency*, in *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, (New York, NY, USA), p. 1–14, Association for Computing Machinery, 2014.

[34] A. Sriraman, S. Liu, S. Gunbay, S. Su, and T. F. Wenisch, *Deconstructing the Tail at Scale Effect Across Network Protocols*, *The Annual Workshop on Duplicating, Deconstructing, and Debunking* (2016).

[35] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, *Enhancing Server Efficiency in the Face of Killer Microseconds*, in *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA '19)*, pp. 185–198, IEEE, 2019.

[36] A. Sriraman, A. Dhanotia, and T. F. Wenisch, *SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale*, in *Proceedings of the 46th International Symposium on Computer Architecture*, p. 513–526, Association for Computing Machinery, 2019.

[37] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha, *Enforcing Isolation and Ordering in STM*, in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), p. 78–88, Association for Computing Machinery, 2007.

[38] T. Harris and K. Fraser, *Language Support for Lightweight Transactions*, *SIGPLAN Not.* **38** (Oct., 2003) 388–402.

[39] T. Shpeisman, A.-R. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc, *Towards Transactional Memory Semantics for C++*, in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, (New York, NY, USA), p. 49–58, Association for Computing Machinery, 2009.

[40] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc, *Practical Weak-Atomicity Semantics for Java STM*, in *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, (New York, NY, USA), p. 314–325, Association for Computing Machinery, 2008.

[41] R. Rajwar and J. R. Goodman, *Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution*, in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, (USA), p. 294–305, IEEE Computer Society, 2001.

[42] D. A. Jiménez and C. Lin, *Dynamic branch prediction with perceptrons*, in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pp. 197–206, IEEE, 2001.

[43] D. Tarjan and K. Skadron, *Merging path and gshare indexing in perceptron branch prediction*, *ACM transactions on architecture and code optimization (TACO)* **2** (2005), no. 3 280–300.

[44] S. L. Graham, P. B. Kessler, and M. K. McKusick, *An execution profiler for modular programs*, *Software: Practice and Experience* **13** (1983), no. 8 671–685.

[45] "sync: Mutex performance collapses with high concurrency." `https://github.com/golang/go/issues/33747`.

[46] D. Melikyan, "Detecting Lock Contention in Go." `https://www.instana.com/blog/detecting-lock-contention-in-go/`, March, 2016.

[47] V. Blanchon, "Go: How to Reduce Lock Contention with the Atomic Package." `https://medium.com/a-journey-with-go/go-how-to-reduce-lock-contention-with-the-atomic-package-ba3b2664b549`, Aug, 2020.

[48] "Why Locking in Go much slower than Java?." `https://stackoverflow.com/questions/39815723/why-locking-in-go-much-slower-than-java-lots-of-time-spent-in-mutex-lock-mut`.

[49] V. Blanchon, "Go: Mutex and Starvation." `https://medium.com/a-journey-with-go/go-mutex-and-starvation-3f4f4e75ad50`, Sep, 2019.

[50] F. Valsorda, "Creative foot-shooting with Go RWMutex." `https://blog.cloudflare.com/creative-foot-shooting-with-go-rwmutex/`, Oct, 2015.

[51] Google, "The Go Programming Language Specification." https://golang.org/ref/spec#Defer_statements.

[52] "Solved: A low background number of - Intel Community." https://community.intel.com/t5/Software-Tuning-Performance/TSX-conflict-aborts-for-single-threaded-applications/m-p/983986#M3190.

[53] C. RITSON and F. BARNES, "An Evaluation of Intel's Restricted Transactional Memory for CPAs." https://core.ac.uk/download/pdf/18531106.pdf.

[54] D. Dice, Y. Lev, M. Moir, and D. Nussbaum, *Early experience with a commercial hardware transactional memory implementation*, in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pp. 157–168, 2009.

[55] A. Kleen, *Scaling Existing Lock-Based Applications with Lock Elision: Lock Elision Enables Existing Lock-Based Programs to Achieve the Performance Benefits of Nonblocking Synchronization and Fine-Grain Locking with Minor Software Engineering Effort.*, *Queue* **12** (Jan., 2014) 20–27.

[56] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum, *Applications of the adaptive transactional memory test platform*, in *3rd ACM SIGPLAN Workshop on Transactional Computing*, pp. 1–10, 2008.

[57] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, *Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, ZEnterprise EC12, Intel Core, and POWER8*, *SIGARCH Comput. Archit. News* **43** (June, 2015) 144–157.

[58] W. Ruan, T. Vyas, Y. Liu, and M. Spear, *Transactionalizing legacy code: An experience report using GCC and memcached*, *ACM SIGARCH Computer Architecture News* **42** (2014), no. 1 399–412.

[59] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner, *Improving in-memory database index performance with Intel® Transactional Synchronization Extensions*, in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 476–487, IEEE, 2014.

[60] "Intel 64 and IA-32 Architectures Optimization Reference Manual." https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[61] "GitHub - linux4life798/safetyfast: An Go library of synchronization primitives to help make use of hardware transactional memory (HTM)." `https://github.com/linux4life798/safetyfast`.

[62] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, *Hybrid NOrec: A Case Study in the Effectiveness of Best Effort Hardware Transactional Memory*, in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (New York, NY, USA), p. 39–52, Association for Computing Machinery, 2011.

[63] C. Jacobi, T. Slegel, and D. Greiner, *Transactional Memory Architecture and Implementation for IBM System Z*, in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, (USA), p. 25–36, IEEE Computer Society, 2012.

[64] D. Dice, A. Kogan, and Y. Lev, *Refined Transactional Lock Elision*, *SIGPLAN Not.* **51** (Feb., 2016).

[65] Y. Afek, A. Levy, and A. Morrison, *Software-Improved Hardware Lock Elision*, in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, (New York, NY, USA), p. 212–221, Association for Computing Machinery, 2014.

[66] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy, *Improved single global lock fallback for best-effort hardware transactional memory*, in *Transact 2014 Workshop. ACM*, p. 54, 2014.

[67] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir, *Adaptive Integration of Hardware and Software Lock Elision Techniques*, in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, (New York, NY, USA), p. 188–197, Association for Computing Machinery, 2014.

[68] N. Diegues and P. Romano, *Self-Tuning Intel Transactional Synchronization Extensions*, in *11th International Conference on Autonomic Computing (ICAC 14)*, (Philadelphia, PA), pp. 209–219, USENIX Association, June, 2014.

[69] N. Diegues, P. Romano, and L. Rodrigues, *Virtues and Limitations of Commodity Hardware Transactional Memory*, in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), p. 3–14, Association for Computing Machinery, 2014.

[70] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer, *Optimizing Hybrid Transactional Memory: The Importance of Nonspeculative Operations*, in *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in*

Algorithms and Architectures, SPAA '11, (New York, NY, USA), p. 53–64, Association for Computing Machinery, 2011.

[71] K. Chapman, A. L. Hosking, and J. E. B. Moss, *Hybrid STM/HTM for Nested Transactions on OpenJDK*, in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, (New York, NY, USA), p. 660–676, Association for Computing Machinery, 2016.

[72] J. Izraelevitz, A. Kogan, and Y. Lev, *Implicit acceleration of critical sections via unsuccessful speculation*, *11th ACM SIGPLAN Wkshp. on Transactional Computing, TRANSACT* **16** (2016).

[73] L. Zheng, X. Liao, H. Jin, and H. Liu, *Exploiting the Parallelism Between Conflicting Critical Sections with Partial Reversion*, *IEEE Transactions on Parallel and Distributed Systems* **28** (2017), no. 12 3443–3457.

[74] G. Sousa and A. Baldassin, *FGSCM: A fine-grained approach to transactional lock elision*, in *29th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2017, Campinas, Brazil, October 17-20, 2017*, pp. 113–120, IEEE Computer Society, 2017.

[75] Q. Wang, P. Su, M. Chabbi, and X. Liu, *Lightweight hardware transactional memory profiling*, in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pp. 186–200, 2019.

[76] A. Kleen, "Lock elision in the GNU C library." `https://lwn.net/Articles/534758/`, January, 2013.

[77] M. Pohlack and S. Diestelhorst, "From lightweight hardware transactional memory to lightweight lock elision.." `https://www.cs.purdue.edu/sss/projects/transact11/papers/Pohlack.pdf`, January, 2011.

[78] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc, *Single Global Lock Semantics in a Weakly Atomic STM*, *SIGPLAN Not.* **43** (May, 2008) 15–26.

[79] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan, *A Uniform Transactional Execution Environment for Java*, in *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, (Berlin, Heidelberg), p. 129–154, Springer-Verlag, 2008.

[80] W. Landi and B. G. Ryder, *Pointer-Induced Aliasing: A Problem Classification*, in *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '91, (New York, NY, USA), p. 93–103, Association for Computing Machinery, 1991.

[81] M. Hind, *Pointer Analysis: Haven't We Solved This Problem Yet?*, in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, (New York, NY, USA), p. 54–61, Association for Computing Machinery, 2001.

[82] M. Chabbi and J. Mellor-Crummey, *Contention-Conscious, Locality-Preserving Locks*, in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[83] Y. Afek, A. Matveev, O. R. Moll, and N. Shavit, *Amalgamated lock-elision*, in *Distributed Computing* (Y. Moses, ed.), (Berlin, Heidelberg), pp. 309–324, Springer Berlin Heidelberg, 2015.

[84] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir, *Adaptive Integration of Hardware and Software Lock Elision Techniques*, in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, (New York, NY, USA), p. 188–197, Association for Computing Machinery, 2014.

[85] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, *Taming Performance Variability*, in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, (USA), p. 409–425, USENIX Association, 2018.

[86] P. Su, S. Jiao, M. Chabbi, and X. Liu, *Pinpointing Performance Inefficiencies via Lightweight Variance Profiling*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[87] J. Huang, B. Mozafari, and T. F. Wenisch, *Statistical Analysis of Latency Through Semantic Profiling*, in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, (New York, NY, USA), p. 64–79, Association for Computing Machinery, 2017.

[88] "Package reflect." `https://golang.org/pkg/reflect/`.

[89] L. Torczon and K. Cooper, *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd ed., 2007.

[90] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, *ACM Trans. Program. Lang. Syst.* **13** (Oct., 1991) 451–490.

[91] R. Johnson, D. Pearson, and K. Pingali, *The Program Structure Tree: Computing Control Regions in Linear Time*, in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, (New York, NY, USA), p. 171–185, Association for Computing Machinery, 1994.

[92] L. O. Andersen, *Program Analysis and Specialization for the C Programming Language*, tech. rep., University of Copenhagen, 1994.

[93] "Reentrant Mutex." `https://en.wikipedia.org/wiki/Reentrant_mutex`.

[94] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, *Atlas: Leveraging Locks for Non-Volatile Memory Consistency*, in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, (New York, NY, USA), p. 433–452, Association for Computing Machinery, 2014.

[95] T. Kelly, "Programming Workbench Hand-Over-Hand Locking for Highly Concurrent Collections." `https://www.usenix.org/system/files/login/articles/login_fall20_14_kelly.pdf`, 2020.

[96] D. F. Bacon and P. F. Sweeney, *Fast Static Analysis of C++ Virtual Function Calls*, in *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, (New York, NY, USA), p. 324–341, Association for Computing Machinery, 1996.

[97] "Package callgraph." `https://pkg.go.dev/golang.org/x/tools/go/callgraph`.

[98] Google, "Profiling Go Programs." `https://blog.golang.org/pprof`.

[99] M. Chabbi, "pprof++: A Go Profiler with Hardware Performance Monitoring." `https://eng.uber.com/pprof-go-profiler/`, May, 2020.

[100] "Package format." `https://golang.org/pkg/go/format/`.

[101] "Package astutil." `https://godoc.org/golang.org/x/tools/go/ast/astutil`.

[102] "Package go/types." `https://golang.org/pkg/go/types/`.

[103] J. Lerch, J. Späth, E. Bodden, and M. Mezini, *Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths*, in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, p. 619–629, IEEE Press, 2015.

[104] "Go by Example: Closures." `https://gobyexample.com/closures`.

[105] M. Chabbi, A. Amer, S. Wen, and X. Liu, *An Efficient Abortable-locking Protocol for Multi-level NUMA Systems*, in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017* (V. Sarkar and L. Rauchwerger, eds.), pp. 61–74, ACM, 2017.

[106] M. L. Scott and W. N. S. III, *Scalable queue-based spin locks with timeout*, in *Proceedings of the 2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01), Snowbird, Utah, USA, June 18-20, 2001* (M. T. Heath and A. Lumsdaine, eds.), pp. 44–52, ACM, 2001.

[107] G. Ammons, T. Ball, and J. R. Larus, *Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling*, in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, (New York, NY, USA), p. 85–96, Association for Computing Machinery, 1997.

[108] D. M. Tullsen, S. J. Eggers, and H. M. Levy, *Simultaneous multithreading: Maximizing on-chip parallelism*, in *Proceedings 22nd Annual International Symposium on Computer Architecture*, pp. 392–403, 1995.

[109] "testing - The Go Programming Language." `https://golang.org/pkg/testing/`.

[110] "Golang benchmark RunParallel API." `https://golang.org/pkg/testing/#B.RunParallel`.

[111] "patrickmn/go-cache: An in-memory key:value store/cache (similar to Memcached) library for Go, suitable for single-machine applications.." `https://github.com/patrickmn/go-cache`.

[112] "Workiva/go-datastructures: A collection of useful, performant, and threadsafe Go datastructures.." `https://github.com/Workiva/go-datastructures`.

[113] "uber-go/zap: Blazing fast, structured, leveled logging in Go.." `https://github.com/uber-go/zap`.

[114] "VictoriaMetrics/fastcache: Fast thread-safe inmemory cache for big number of entries in Go. Minimizes GC overhead." `https://github.com/VictoriaMetrics/fastcache`.

[115] R. Rajwar and J. R. Goodman, *Speculative lock elision: Enabling highly concurrent multithreaded execution*, MICRO 34, (USA), p. 294–305, IEEE Computer Society, 2001.

[116] R. Karedla, J. S. Love, and B. G. Wherry, *Caching strategies to improve disk system performance*, *Computer* **27** (1994), no. 3 38–46.

[117] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. R. Karlin, and K. Li, *A trace-driven comparison of algorithms for parallel prefetching and caching*, in *OSDI*, pp. 19–34, 1996.

[118] R. Corp., "Automatic NUMA balancing." `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-auto_numa_balancing`.

[119] Dice Dave, "waiting policies for locks : spin-then-park ." `https://blogs.oracle.com/dave/waiting-policies-for-locks-:-spin-then-park`, April, 2015.

[120] S. Kashyap, C. Min, and T. Kim, *Scalable numa-aware blocking synchronization primitives*, in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 603–615, USENIX Association, July, 2017.

[121] W. N. Scherer and M. L. Scott, *Advanced contention management for dynamic software transactional memory*, PODC '05, (New York, NY, USA), p. 240–248, Association for Computing Machinery, 2005.

[122] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis, *Adaptive locks: Combining transactions and locks for efficient concurrency*, in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, (USA), p. 3–14, IEEE Computer Society, 2009.

[123] D. Dice, A. Kogan, Y. Lev, T. Merrifield, and M. Moir, *Adaptive integration of hardware and software lock elision techniques*, in *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, (New York, NY, USA), p. 188–197, Association for Computing Machinery, 2014.

[124] E. Teran, Z. Wang, and D. A. Jiménez, *Perceptron learning for reuse prediction*, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[125] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, *Perceptron-based prefetch filtering*, in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–13, IEEE, 2019.

[126] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abu-Ghazaleh, and D. A. Jiménez, *Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron*, in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1124–1137, IEEE, 2020.

[127] O. Z. Maimon and L. Rokach, *Data mining with decision trees: theory and applications*, vol. 81. World scientific, 2014.

[128] D. A. Freedman, *Statistical models: theory and practice*. cambridge university press, 2009.

[129] G. I. Webb, E. Keogh, and R. Miikkulainen, *Naïve bayes.*, *Encyclopedia of machine learning* **15** (2010) 713–714.

[130] T. Chen and C. Guestrin, *XGBoost: A scalable tree boosting system*, in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 785–794, ACM, 2016.

[131] E. Fix and J. L. Hodges, *Discriminatory analysis. nonparametric discrimination: Consistency properties*, *International Statistical Review/Revue Internationale de Statistique* **57** (1989), no. 3 238–247.

[132] J. J. Hopfield, *Neural networks and physical systems with emergent collective computational abilities.*, *Proceedings of the national academy of sciences* **79** (1982), no. 8 2554–2558.

[133] GitHub, "Github - nlynch-mentor/vdsotest: Utility for testing and benchmarking a linux vdso." `https://github.com/nlynch-mentor/vdsotest`, 2018. (Accessed on 04/08/2022).

[134] D. Dice, O. Shalev, and N. Shavit, *Transactional locking ii*, in *International Symposium on Distributed Computing*, pp. 194–208, Springer, 2006.

[135] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, *Mcrt-stm: a high performance software transactional memory system for a multi-core runtime*, in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 187–197, 2006.

[136] M. F. Spear, M. M. Michael, and C. Von Praun, *Ringstm: scalable transactions with a single atomic instruction*, in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pp. 275–284, 2008.

[137] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, *Performance evaluation of intel® transactional synchronization extensions for high-performance computing*, in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2013.

[138] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, *Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8*, *ACM SIGARCH Computer Architecture News* **43** (2015), no. 3S 144–157.

[139] LWN, "infrastruture for monitoring request queue congestion [lwn.net]." https://lwn.net/Articles/9519/, 2002. (Accessed on 04/05/2022).

[140] LWN, "Development kernel 2.5.39 released [lwn.net]." https://lwn.net/Articles/11130/, 2002. (Accessed on 04/05/2022).

[141] Linux, "backing-dev.c - mm/backing-dev.c - linux source code (v5.15-rc6) - bootlin." https://elixir.bootlin.com/linux/v5.15-rc6/source/mm/backing-dev.c, 2021. (Accessed on 04/05/2022).

[142] LWN, "Replacing congestion_wait() [lwn.net]." https://lwn.net/Articles/873672/, 2021. (Accessed on 04/05/2022).

[143] LWN, "[patch v5 0/8] remove dependency on congestion_wait in mm/ [lwn.net]." https://lwn.net/ml/linux-kernel/20211022144651.19914-1-mgorman@techsingularity.net/, 2021. (Accessed on 04/05/2022).

[144] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, *Tracing the meta-level: Pypy's tracing jit compiler*, in *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pp. 18–25, 2009.

[145] PyPy, "Jit help — pypy documentation." https://doc.pypy.org/en/latest/jit_help.html, 2021. (Accessed on 04/05/2022).

[146] Z. Yu, Z. Bei, and X. Qian, *Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing*, in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 564–577, 2018.

[147] Y. Li and Z. M. J. Jiang, *Assessing and optimizing the performance impact of the just-in-time configuration parameters-a case study on pypy*, *Empirical Software Engineering* **24** (2019), no. 4 2323–2363.

[148] D. Terpstra, H. Jagode, H. You, and J. Dongarra, *Collecting performance data with papi-c*, in *Tools for High Performance Computing 2009*, pp. 157–173. Springer, 2010.

[149] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, *Stamp: Stanford transactional applications for multi-processing*, in *2008 IEEE International Symposium on Workload Characterization*, pp. 35–46, IEEE, 2008.

[150] M. Gorman, "gormanm/mmtests: Mmtests: Benchmarking framework primarily aimed at linux kernel testing." https://github.com/gormanm/mmtests, 2011. (Accessed on 12/01/2021).

[151] M. Á. Abella-González, P. Carollo-Fernández, L.-N. Pouchet, F. Rastello, and G. Rodríguez, *Polybench/python: benchmarking python environments with polyhedral optimizations*, in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, pp. 59–70, 2021.

[152] "pyston/python-macrobenchmarks: A collection of macro benchmarks for the python programming language." `https://github.com/pyston/python-macrobenchmarks`. (Accessed on 06/17/2022).

[153] "Welcome to flask — flask documentation (2.1.x)." `https://flask.palletsprojects.com/en/2.1.x/`. (Accessed on 06/17/2022).

[154] "django-cms/django-cms: The easy-to-use and developer-friendly enterprise cms powered by django." `https://github.com/django-cms/django-cms`. (Accessed on 06/17/2022).

[155] "Gunicorn - python wsgi http server for unix." `https://gunicorn.org/`. (Accessed on 06/17/2022).

[156] V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, *Measuring energy and power with papi*, in *2012 41st international conference on parallel processing workshops*, pp. 262–268, IEEE, 2012.

[157] LWN, "[patch v5 0/8] remove dependency on congestion_wait in mm/ [lwn.net]." `https://lwn.net/ml/linux-kernel/20211022144651.19914-1-mgorman@techsingularity.net/`, 2021. (Accessed on 12/01/2021).

[158] M. Gorman, "kernel/git/mel/linux.git - candidate patch series by mel gorman." `https://git.kernel.org/pub/scm/linux/kernel/git/mel/linux.git/commit/?h=mm-reclaimcongest-v5r4&id=a2f8f6191574311e28d0c3609394937533ec490c`, 2021. (Accessed on 12/01/2021).

[159] T. Kraska, *Towards instance-optimized data systems*, *Proc. VLDB Endow.* **14** (jul, 2021) 3222–3232.

[160] M. Mitzenmacher and S. Vassilvitskii, *Algorithms with Predictions*, p. 646–662. Cambridge University Press, 2021.

[161] V. Carbune, T. Coppey, A. Daryin, T. Deselaers, N. Sarda, and J. Yagnik, *Smartchoices: hybridizing programming and machine learning*, *Reinforcement Learning for Real Life (RL4RealLife) Workshop in the 36th International Conference on Machine Learning (ICML)* (2019).

[162] L. Lamport, *A fast mutual exclusion algorithm*, *ACM Trans. Comput. Syst.* **5** (Jan., 1987) 1–11.

[163] J.-H. Yang and J. H. Anderson, *A fast, scalable mutual exclusion algorithm*, *Distrib. Comput.* **9** (Mar., 1995) 51–60.

[164] M. Chabbi and J. Mellor-Crummey, *Contention-conscious, locality-preserving locks*, in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[165] B.-H. Lim and A. Agarwal, *Reactive synchronization algorithms for multiprocessors*, in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, (New York, NY, USA), p. 25–35, Association for Computing Machinery, 1994.

[166] M. Zhang, H. Chen, L. Cheng, F. C. M. Lau, and C. Wang, *Scalable adaptive numa-aware lock*, *IEEE Transactions on Parallel and Distributed Systems* **28** (2017), no. 6 1754–1769.

[167] Z. Zhang, M. Chabbi, A. Welc, and T. Sherwood, *Optimistic concurrency control for real-world go programs*, in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 939–955, 2021.

[168] B. Rountree, D. K. Lowenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, *Adagio: Making dvs practical for complex hpc applications*, in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), p. 460–469, Association for Computing Machinery, 2009.

[169] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, *A run-time system for power-constrained hpc applications*, in *High Performance Computing* (J. M. Kunkel and T. Ludwig, eds.), (Cham), pp. 394–408, Springer International Publishing, 2015.

[170] N. Gholkar, F. Mueller, and B. Rountree, *Uncore power scavenger: A runtime for uncore power conservation on hpc systems*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[171] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, *A regression-based approach to scalability prediction*, in *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, (New York, NY, USA), p. 368–377, Association for Computing Machinery, 2008.

[172] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, *Neural acceleration for general-purpose approximate programs*, in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, IEEE, 2012.

[173] F. Salfner, M. Lenk, and M. Malek, *A survey of online failure prediction methods*, *ACM Computing Surveys (CSUR)* **42** (2010), no. 3 1–42.

[174] Y. Watanabe, H. Otsuka, M. Sonoda, S. Kikuchi, and Y. Matsumoto, *Online failure prediction in cloud datacenters by real-time message pattern learning*, in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, pp. 504–511, IEEE, 2012.

[175] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, *Failures in large scale systems: long-term measurement, analysis, and implications*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.

[176] S. Fu and C.-Z. Xu, *Exploring event correlation for failure prediction in coalitions of clusters*, in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pp. 1–12, 2007.

[177] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske, *Hora: Architecture-aware online failure prediction*, *Journal of Systems and Software* **137** (2018) 669–685.

[178] A. Gainaru, F. Cappello, M. Snir, and W. Kramer, *Failure prediction for hpc systems and applications: Current situation and open issues*, *The International journal of high performance computing applications* **27** (2013), no. 3 273–282.

[179] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, *Rapidly selecting good compiler optimizations using performance counters*, in *International Symposium on Code Generation and Optimization (CGO'07)*, pp. 185–197, 2007.

[180] K. D. Cooper, P. J. Schielke, and D. Subramanian, *Optimizing for reduced code space using genetic algorithms*, in *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, (New York, NY, USA), p. 1–9, Association for Computing Machinery, 1999.

[181] M. M. Chabbi, J. M. Mellor-Crummey, and K. D. Cooper, *Efficiently exploring compiler optimization sequences with pairwise pruning*, in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pp. 34–45, 2011.

[182] R. Vuduc, J. W. Demmel, and J. A. Bilmes, *Statistical models for empirical search-based performance tuning*, *The International Journal of High Performance Computing Applications* **18** (2004), no. 1 65–94.

[183] Z. Pan and R. Eigenmann, *Fast and effective orchestration of compiler optimizations for automatic performance tuning*, in *International Symposium on Code Generation and Optimization (CGO'06)*, pp. 12–pp, IEEE, 2006.

[184] K. Wang, X. Lin, and W. Tang, *Predator—an experience guided configuration optimizer for hadoop mapreduce*, in *4Th IEEE international conference on cloud computing technology and science proceedings*, pp. 419–426, IEEE, 2012.

[185] P. Lengauer and H. Mössenböck, *The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors*, in *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pp. 111–122, 2014.

[186] R. Thonangi, V. Thummala, and S. Babu, *Finding good configurations in high-dimensional spaces: Doing more with less*, in *2008 IEEE international symposium on modeling, analysis and simulation of computers and telecommunication systems*, pp. 1–10, IEEE, 2008.

[187] T. Osogami and S. Kato, *Optimizing system configurations quickly by guessing at the performance*, in *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 145–156, 2007.

[188] C.-Q. Yang and B. Miller, *Critical path analysis for the execution of parallel and distributed programs*, in *[1988] Proceedings. The 8th International Conference on Distributed*, pp. 366–373, 1988.

[189] "Jaeger: open source, end-to-end distributed tracing." `https://www.jaegertracing.io/`. (Accessed on 12/01/2021).

[190] "Heat Map." `https://en.wikipedia.org/wiki/Heat_map`. (Accessed on 01/11/2022).

[191] B. Gregg, *The flame graph*, *Communications of the ACM* **59** (2016), no. 6 48–57.

[192] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, *et. al.*, *Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks*, in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 48–58, IEEE, 2020.

[193] "Uber's Fulfillment Platform: Ground-up Re-architecture to Accelerate Uber's Go/Get Strategy." `https://eng.uber.com/fulfillment-platform-rearchitecture/`. (Accessed on 01/12/2022).

[194] S. L. Graham, P. B. Kessler, and M. K. Mckusick, *Gprof: A call graph execution profiler*, in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, (New York, NY, USA), p. 120–126, Association for Computing Machinery, 1982.

[195] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, *HPCToolkit: Tools for performance analysis of optimized parallel programs*, *Concurrency and Computation: Practice and Experience* **22** (2010), no. 6 685–701.

[196] "perf: Linux profiling with performance counters." `https://perf.wiki.kernel.org/index.php/Main_Page`, 2013.

[197] "perf (linux) - wikipedia." `https://en.wikipedia.org/wiki/Perf_(Linux)`. (Accessed on 01/12/2022).

[198] OpenTracing Developers, "OpenTracing: What is Distributed Tracing?." `https://opentracing.io/docs/overview/what-is-tracing/`.

[199] OpenTracing Developers, "OpenTracing: Span." `https://opentracing.io/docs/overview/spans/`.

[200] "Head-based and tail-based sampling, rate-limiting." `https://opentelemetry.uptrace.dev/guide/sampling.html#introduction`, April, 2022.

[201] "User Datagram Protocol." `https://en.wikipedia.org/wiki/User_Datagram_Protocol`. (Accessed on 01/11/2022).

[202] J. Kreps, N. Narkhede, J. Rao, *et. al.*, *Kafka: A distributed messaging system for log processing*, in *Proceedings of the NetDB*, vol. 11, pp. 1–7, 2011.

[203] "Evolving Schemaless into a Distributed SQL Database." `https://eng.uber.com/schemaless-sql-database/`. (Accessed on 01/12/2022).

[204] "Fast and Reliable Schema-Agnostic Log Analytics Platform." `https://eng.uber.com/logging/`. (Accessed on 01/12/2022).

[205] Apache Flink Team, "Apache Flink: Stateful Computations over Data Streams." `https://flink.apache.org/`.

[206] M. Frigo, C. E. Leiserson, and K. H. Randall, *The Implementation of the Cilk-5 Multithreaded Language*, in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, (New York, NY, USA), p. 212–223, Association for Computing Machinery, 1998.

[207] B. Fields, S. Rubin, and R. Bodik, *Focusing processor policies via critical-path prediction*, in *Proceedings 28th Annual International Symposium on Computer Architecture*, pp. 74–85, IEEE, 2001.

[208] G. Ammons, T. Ball, and J. R. Larus, *Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling*, in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, (New York, NY, USA), p. 85–96, Association for Computing Machinery, 1997.

[209] Wikipedia, "Clock drift." `https://en.wikipedia.org/wiki/Clock_drift`.

[210] S. S V, "Time drift monitoring: Troubles of unsynchronized servers - site24x7 blog." `https://www.site24x7.com/blog/time-drift-monitoring-troubles-of-unsynchronized-servers`. (Accessed on 06/02/2022).

[211] bluematador, "Time Drift (NTP)." `https://www.bluematador.com/docs/troubleshooting/time-drift-ntp`.

[212] B. Gregg, "Differential Flame Graphs." `https://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html`.

[213] D. P. Kingma and M. Welling, *Auto-encoding variational bayes*, *arXiv preprint arXiv:1312.6114* (2013).

[214] V. A. Epanechnikov, *Non-parametric estimation of a multivariate probability density*, *Theory of Probability & Its Applications* **14** (1969), no. 1 153–158.

[215] P. H. Westfall and S. S. Young, *Resampling-based multiple testing: Examples and methods for p-value adjustment*, vol. 279. John Wiley & Sons, 1993.

[216] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, *Sage: practical and scalable ML-driven performance debugging in microservices*, in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 135–151, 2021.

[217] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, *FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices*, in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 805–825, 2020.

[218] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson, *The Cilkprof Scalability Profiler*, in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, (New York, NY, USA), p. 89–100, Association for Computing Machinery, 2015.

[219] A. Yoga and S. Nagarakatte, *Parallelism-centric what-if and differential analyses*, in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 485–501, 2019.

[220] N. R. Tallent and J. M. Mellor-Crummey, *Effective performance measurement and analysis of multithreaded applications*, in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 229–240, 2009.

[221] C. Curtsinger and E. D. Berger, *Coz: Finding code that counts with causal profiling*, in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 184–197, 2015.

[222] X. Liu, J. Mellor-Crummey, and M. Fagan, *A New Approach for Performance Analysis of OpenMP Programs*, in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, (New York, NY, USA), p. 69–80, Association for Computing Machinery, 2013.

[223] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein, *Global critical path: A tool for system-level timing analysis*, in *Proceedings of the 44th annual Design Automation Conference*, pp. 783–786, 2007.

[224] P. Barford and M. Crovella, *Critical path analysis of TCP transactions*, *ACM SIGCOMM Computer Communication Review* **30** (2000), no. 4 127–138.

[225] D. Böhme, M. Geimer, L. Arnold, F. Voigtlaender, and F. Wolf, *Identifying the Root Causes of Wait States in Large-Scale Parallel Applications*, *ACM Trans. Parallel Comput.* **3** (jul, 2016).

[226] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, *et. al.*, *Canopy: An end-to-end performance tracing and analysis system*, in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 34–50, 2017.

[227] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, *lprof: A non-intrusive request flow profiler for distributed systems*, in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 629–644, 2014.

[228] J. Mace, R. Roelke, and R. Fonseca, *Pivot tracing: Dynamic causal monitoring for distributed systems*, in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 378–393, 2015.

[229] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, *The mystery machine: End-to-end performance analysis of large-scale internet services*, in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 217–231, 2014.

[230] "Edgar: Solving Mysteries Faster with Observability." `https://netflixtechblog.com/edgar-solving-mysteries-faster-with-observability-e1a76302c71f`. (Accessed on 11/30/2021).

[231] A. Yasin, *A top-down method for performance analysis and counters architecture*, in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 35–44, IEEE, 2014.

[232] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, *Profiling a warehouse-scale computer*, in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 158–169, 2015.

[233] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), p. 248–259, Association for Computing Machinery, 2011.

[234] H. Yang, A. Breslow, J. Mars, and L. Tang, *Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers*, in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, (New York, NY, USA), p. 607–618, Association for Computing Machinery, 2013.

[235] K. Sohn, H. Lee, and X. Yan, *Learning structured output representation using deep conditional generative models*, *Advances in neural information processing systems* **28** (2015) 3483–3491.