Thèse n° 9028

Latency Interfaces for Systems Code

Présentée le 8 septembre 2023

Faculté informatique et communications
Laboratoire des systemes fiables
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Rishabh Ramesh IYER

Acceptée sur proposition du jury

Prof. E. Bugnion, président du jury
Prof. G. Candea, Prof. A. Argyraki, directeurs de thèse
Prof. J. Hoffman, rapporteur
Prof. M. Swift, rapporteur
Dr J. Mogul, rapporteur
Prof. T. Bourgeat, rapporteur

École
polytechnique
fédérale
de Lausanne

2023

While today’s world is one of immediacy, life’s greatest goals
still do not materialize overnight. Rather, they materialize through
resilience, perseverance and the lessons of failure.

— Rafael Nadal

To my family…

Acknowledgements

I look back upon the last six years fondly. While there were many challenges and several ups and downs, I found great friends and inspiring mentors who have celebrated my victories and helped me persevere through the bad times. While this thesis—out of necessity—lists only my name, it would not have existed without these amazing people, many of whom I list below.

My advisors, George Candea and Katerina Argyraki: George and Katerina have had an indelible impact on not only this thesis but also on my approach to research. George has instilled in me the importance of thinking big and focusing on research that will endure the test of time (*your paper should be relevant in 10 years!*), while simultaneously emphasizing the importance of being meticulous and getting every minuscule detail right (*what will your children think of you if they discover a typo in your work?*). Throughout my Ph.D, George set uncompromising standards and pushed me to improve as a researcher every single day, but constantly led by example and demonstrated that those standards could indeed be met.

Katerina, whom I consider a master of simplicity, has taught me the ability to distill and precisely communicate the key ideas underlying a research project. She often had to do this the hard way, by carefully unraveling the core ideas from my tangled thoughts, but remained patient with me throughout, which is something I will always be thankful for. Katerina's ability to focus on what matters goes beyond research; I have often found myself in her office, venting about either paper rejections or why my research is pointless, but she always managed to miraculously tell me precisely what I needed to hear in order to soldier on.

Both George and Katerina have gone well beyond what is typically expected of an advisor in terms of looking out for me, both personally and professionally. In particular, both of them had unwavering faith in me and the topic of my thesis, even when I had lost faith in both. I sincerely hope that the end of graduate school does not mean the end of having them as advisors, since I plan to rely on their advice for the foreseeable future. Thank you, George and Katerina, for everything. This thesis belongs to the two of you as much as it does to me.

My thesis committee and mentors at EPFL: I am grateful to Jan Hoffman, Mike Swift, Jeff Mogul, Thomas Bourgeat, and Ed Bugnion for serving on my thesis committee. Jeff and Thomas in particular, read the entire document in great detail and provided feedback that significantly improved the thesis. Ed, apart from being an amazing committee president, has been a great mentor over the past few years and has always taken the time to provide valuable feedback on my research ideas.

I would also like to thank three other professors at EPFL: Babak Falsafi, James Larus, and Sanidhya Kashyap. Babak hired me as an intern at EPFL when I was still an undergrad at IIT Bombay. This was my first foray into research and without this opportunity (and his subsequent letter of recommendation), I would never have pursued a Ph.D. Jim, like Ed, always made time to give me feedback on my research ideas. What I particularly appreciated about Jim was his brutally honest feedback, which often laid bare flaws that I had hoped others would ignore. Finally, Sanidhya (whom I first met outside as a fellow student at a conference), has been halfway between a friend and a mentor and someone who has always made time for me.

DSLAB: Next, I would like to thank my fellow members of the Dependable System Lab (DSLAB), which has been my home for the last six years.

Arseniy Zaostrovnykh was my first friend in the lab and remains one of my closest friends to this day. Apart from being one of the sharpest people I know and an invaluable sounding board for anything verification-related, Arseniy has been the one person who not only put up with but also shared my craziness and lack of self-preservation when it came to sporting activities. Arseniy and I have spent countless hours together biking, rafting, climbing, competing in the gym, axe-throwing, and even wrestling all over the office floor. This has led to some incredibly memorable incidents that I would not trade for all the world: from jumping into Lake Geneva together after an 8-hour bike ride circumnavigating it, to me tearing a shoulder ligament when racing down a ski slope and having to take a 12-hour flight to Canada while being unable to move my neck after a particularly intense wrestling match with Arseniy.

Solal Pirelli and I joined the Ph.D program at the same time and have been through many of the ups and downs of a Ph.D in DSLAB together. Solal is one of the most unique people I have met and talking to him always provides a perspective that I will not get from anyone else. This unique perspective has also significantly contributed to this thesis; it was a stray comment by Solal that actually provided an escape from one of the deepest ruts I found myself in during the Ph.D and provided the spark for what turned into Chapter 4 of this thesis.

Lei, Can, Yugesh, and Jiacheng represent the new generation of DSLAB. While our Ph.Ds did not overlap for as long as I would have liked, I am grateful to them for breathing new life into DSLAB, particularly after Arseniy had left. Can in particular, has become a great friend and I will miss our constant banter, his constant support during SOSP/OSDI deadline scrambles, and most of all, our incredibly competitive board game battles. I am also grateful to Yugesh for our multi-hour discussions on history, spirituality, and religion, and to Jaicheng for expanding upon my thesis work in a way that I never could have done.

I would also like to thank the interns I have worked with over the past 6 years—Musa, Daneshvar, Yugesh, Vlad, Ayoub, Beyazit, and Mihai—many of whom made small, but critical contributions to the work presented in the thesis. Musa, in particular, played a pivotal role in getting Concord (Chapter 3) off the ground, while Ayoub was responsible for extending my ideas to cryptographic code (§6.4.2). What has been a particular source of pride is that nearly all of my past interns are now successful Ph.D students in great graduate programs around the world.

Finally, I am incredibly grateful to our lab admin Celine Brzak, and her three canine assistants Beary, River, and Soho. Celine is not only the best possible lab admin—who ensured that I never had to worry about dealing with any EPFL bureaucracy—but also an endless source of positivity. It is virtually impossible to leave Celine's office without a big smile on one's face and Celine, Beary, River, and Soho were usually the ones who managed to cheer me up even when I was having the worst of days.

NAL: I would also like to thank my fellow members of the Network Architecture Lab (NAL), Luis, Pavlos, Mia, Zeinab, and Catalina.

Luis Pedrosa was a postdoc in NAL when I began my Ph.D and the primary collaborator on the first two papers I submitted as a Ph.D student. Luis and I spent most of the summer of 2018 in front of a whiteboard—he lost an entire summer's worth of time, while I gained an understanding of compilers, program analysis, and network functions. I will always be grateful to Luis for being so generous with his time and for helping me find my footing in a field of research that was vastly different from what I studied as an undergrad.

I am grateful to Pavlos for his sagacious advice on how to navigate the trials and tribulations of a Ph.D, Mia for introducing me to the sport of spikeball, and Zeinab and Catalina for our incredibly interesting lunch and dinner conversations that revolved around the peculiarities of India, Lebanon, and Chile, respectively.

Sylvia Ratnasamy, Scott Shenker and the Netsys Lab at UC Berkeley: I am grateful to Sylvia and Scott for hosting me at Berkeley for 6 months of my Ph.D. Both Sylvia and Scott are researchers whose work I have admired for the longest time, and being able to collaborate with the two of them was among the highlights of my Ph.D. I am also grateful to the members of 419 Soda: Narek, Vivian, Tenzin (honorary member), Wen, and Akshay who were incredibly welcoming and helped me settle in at Netsys remarkably quickly.

Siddharth Gupta and Pakshal Bohra: Sid and Pakshal have been my two closest friends during my Ph.D, they are the ones who know me best and the ones I rely on above everyone else during my worst days.

Sid (who is often been referred to as my work-wife) and I have navigated nearly every up and down in the Ph.D together. Over the past six years, Sid was not only a constant source of inspiration with his uniquely creative approach to research but also my companion on hundreds of walks along the shores of Lake Geneva where we discussed work, life, and everything in between, my ski buddy, my fellow gourmet, the one who effectively filed my tax returns (since I simply copy-pasted what he painstakingly filed and changed the name), and the one who frequently rescued me from both embarrassing and life-threatening situations. It is proving surprisingly difficult for me to summarize Sid's role over the past 6 years in just a few lines, so I will instead say this: I hope everyone finds a friend like Sid to share their Ph.D experience with, it is among the best things that can happen to you.

Pakshal has been one of my closest friends since we first shared a dorm together back at IIT Bombay over 10 years ago. While we initially bonded over our shared love for Man Utd and Nadal, I have come to rely on Pakshal to tell me the things I do not like to admit to myself. Pakshal is incredibly patient and has put up with several of my idiosyncracies over the past 10 years; from the times when I would make him walk several kilometers out of the way at IIT so that I could catch my favorite Pokemon in Pokemon Go, to my insistence that we always walk back after eating out at a restaurant no matter how long the walk may be. As I prepare to leave Switzerland (while Pakshal plans to stay on), one of the things I am most sad about is the fact that my streak of playing hundreds of games with Pakshal on every single version of FIFA since 2013 will finally come to an end.

Mario Drumond and Marios Kogias: Mario and Marios have been akin to two older brothers throughout my Ph.D journey.

Mario was my supervisor when I was an intern at EPFL more than 7 years ago, and while he failed to show up and left me hanging on my very first day (he was too drunk from the night before :P), it was primarily due to him, and how much he seemed to be enjoying his Ph.D, that I chose to pursue a Ph.D myself. Mario is one of the most entertaining people I have met; there is never a dull moment when he is around and my Ph.D journey would have been much less enjoyable without him constantly mocking my all-too-frequent bouts of negativity, my obsession with elegant research, and my need to pursue unnecessary "feats of strength". That said, Mario has also been a constant source of support and guidance and is one of the most large-hearted people I have ever met, always being the first to celebrate my victories.

Marios is the best systems Ph.D student I have ever met and (outside of George and Katerina) has contributed the most to both the technical content of this thesis and my growth as a systems researcher. He is always the first person I go to whenever I need feedback on a new research idea and I will always cherish the hundreds of hours we spent discussing random research ideas. What I find particularly heartwarming is that our relationship seems to have been symbiotic: while Marios started out working exclusively on low-latency datacenter systems, and I was more focused on program analysis and hardware, Marios has now begun publishing in conferences focused exclusively on programming languages research [155] while my latest publication is targeted at improving the efficiency of low-latency datacenter systems [126]. Marios has also been a great friend outside of work and was not only my first driving instructor, but also the one who taught me to vacation as the Greeks (apparently) do, which involves doing nothing other than swimming, sleeping, eating, and lazing on the beach.

Sahand Kashani and Mahyar Emami: I am also grateful to my fellow members of FUBAR, Sahand, and Mahyar, for being two of the most dependable friends that one could ever ask for.

Sahand, in particular, exemplifies the qualities of discipline and perseverance; if I had to pick one person in the world to complete an assigned task in the allotted time, I would pick Sahand in a heartbeat. Sahand's discipline and perseverance have been an inspiration throughout my Ph.D and I will be glad if I am able to inculcate even a small fraction of his qualities. Outside of work, Sahand has also been just as enthusiastic as Arseniy when it comes to sporting activities and has been a frequent companion on long bike rides, hikes, and bouldering workouts but only if we agreed to pursue said activity either indoors or before sunrise, given his pathological fear of being sunburnt. He has also been our resident evangelist for Italian food, with his insistence on visiting an Italian restaurant at least once a week.

Mahyar is a man of few words, but when he speaks he is always worth listening to. Mahyar possesses a keen understanding of people and has often given me invaluable advice on how to deal with tumultuous situations. TU, Sahand and Mahyar for always keeping the door to your office open for me.

Other friends at EPFL and in Lausanne: There are several others at EPFL whom I have been fortunate enough to call friends and who have greatly enriched my Ph.D journey. I am grateful to Venu for all the cricket games and bouldering sessions, and Aditya for the football games, constant banter, being my fellow chess fanatic, and introducing me to Brandon Sanderson. I am also grateful to Arash for teaching me how to enjoy the little things in life, Adrien for providing a unique mixture of craziness and empathy, Mark for always being encouraging, and Oggy, Negar and Tesca for our fun dinners together.

I am also thankful to the members of Lausanne's Ultimate Frisbee Pickup social team, in particular Lee Sze Chuin, Marc Checkly, and Enrico Eberhard. Our biweekly frisbee games were something I actively looked forward to, and provided a welcome respite from all things EPFL.

There are almost certainly several other friends whom I have missed; please know that this was done inadvertently and that all of you played a huge role in making my PhD journey worthwhile.

Family: Finally, I am eternally grateful to my parents and my sister Rithvika for their unconditional love and support throughout the past six years. Trying to find the right words to thank them is a futile endeavor. Amma, Appa, and Rithu, you know how much you mean to me. This thesis is dedicated to the three of you.

Lausanne, December 8, 2023 Rishabh Iyer

Abstract

This thesis demonstrates that it is feasible for systems code to expose a latency interface that describes its latency and related side effects for all inputs, just like the code's semantic interface describes its functionality and related side effects.

Semantic interfaces, such as code documentation, header files, and specifications, are indispensable. By providing a succinct summary of a system's functionality, they make it possible for developers to efficiently reason about, use and deploy code they did not write themselves. In contrast, there is no equivalent construct that describes latency behavior in a way that is simultaneously succinct, precise, and complete. Widely-used representations such as envelopes (e.g., probabilistic upper bounds or asymptotic time complexity) or benchmarks (e.g., SPEC or TPC-C results) provide an incomplete understanding of latency, leading to hiccups and meltdowns in production when the workload or runtime environment changes in unpredicted ways.

We take a three-part approach to realize latency interfaces for systems code.
First, we show how to design datacenter systems that provide predictable latency behavior while sustaining high throughput. We present Concord, an efficient runtime for datacenter applications that demonstrates how the careful approximation (as opposed to canonical implementation) of theoretically optimal scheduling policies enables datacenter systems to sustain significantly higher throughput while continuing to meet the same latency targets.

Second, we propose that the latency interface of a system be a program that accepts the same input(s) as the system and outputs its processing latency. We contribute three key ideas that help summarize latency in a succinct, precise, and complete manner: latency-critical variables, which provide succinct abstractions of how the system interacts with its environment, the latency resolution, which provides readers of the interface with explicit control over the trade-off between succinctness and precision, and deployment-specific interfaces which enable users of the system to reason precisely about its latency behavior in their distinct deployment environments. We concretize this representation in the domain of network functions (NFs) and present LINX, a program analysis tool that automatically extracts latency interfaces from NF implementations. We demonstrate that the LINX-extracted interfaces are succinct, precise, and complete and show how they can be used to identify latency regressions, diagnose and fix performance bugs, as well as identify the latency impact of NIC offloads.

Third, we present CFAR, a technique, and tool that allows developers to reason precisely about micro-architectural side effects (specifically CPU cache usage) of systems code. CFAR introduces memory distillates, an intermediate representation that contains all information relevant to how a program accesses memory and discards everything else. CFAR automatically extracts memory distillates from systems code and allows developers to query the distillate to answer specific questions about the code's cache usage. We demonstrate that CFAR enables developers to not only identify inputs that lead to inefficient cache usage and security vulnerabilities in their own code, but also reason about the performance impact of using third-party code.

Résumé

Cette thèse démontre qu'il est possible pour le code système d'exposer une interface de latence qui décrit la latence et les effets secondaires associés pour toutes les entrées, tout comme une interface sémantique décrit les fonctionnalités et les effets secondaires.

Les interfaces sémantiques, telles que la documentation, les fichiers d'en-tête et les spécifications, sont indispensables. En fournissant un résumé succinct des fonctionnalités d'un système, elles permettent aux développeurs de raisonner, d'utiliser et de déployer efficacement du code qu'ils n'ont pas écrit eux-mêmes. En revanche, il n'existe pas d'équivalent décrivant la latence de manière succincte, précise et complète. Des représentations largement utilisées telles que les enveloppes (par exemple, limites supérieures probabilistes ou complexité temporelle asymptotique) ou les benchmarks (par exemple, résultats SPEC ou TPC-C) fournissent une description incomplète de la latence, conduisant à des ratés et des effondrements en production lorsque la charge de travail ou l'environnement d'exécution changent de manière imprévue.

Nous adoptons une approche en trois parties pour réaliser des interfaces de latence pour le code système.
Tout d'abord, nous montrons comment concevoir des applications pour centre de données offrant un comportement de latence prévisible tout en maintenant un débit élevé. Nous présentons Concord, un environnement d'exécution efficace pour ces applications, qui démontre comment l'approximation minutieuse (par opposition à l'implémentation exacte) de techniques théoriquement optimales permet de maintenir un débit nettement plus élevé tout en respectant les mêmes objectifs de latence.

Deuxièmement, nous proposons que l'interface de latence d'un système soit un programme qui accepte les mêmes entrées que le système et retourne sa latence. Nous apportons trois idées clés qui aident à résumer la latence de manière succincte, précise et complète : les variables critiques en matière de latence, qui décrivent succinctement la manière dont le système interagit avec son environnement, la résolution de latence, qui fournit aux lecteurs de l'interface un contrôle explicite entre concision et précision, et les interfaces spécifiques au déploiement, qui permettent aux utilisateurs du système de raisonner précisément sur son comportement de latence dans leurs environnements spécifiques. Nous concrétisons cette représentation dans le domaine des fonctions réseau et présentons LINX, un outil d'analyse qui extrait automatiquement les interfaces de latence des implémentations de ces fonctions. Nous démontrons que les interfaces extraites par LINX sont succinctes, précises et complètes et montrons comment elles peuvent être utilisées pour identifier les régressions de latence, diagnostiquer et corriger les bugs de performances, ainsi qu'identifier l'impact sur la latence des fonctionnalités de déchargement.

Troisièmement, nous présentons CFAR, une technique et outil qui permet aux développeurs de raisonner précisément sur les effets secondaires micro-architecturaux (en particulier l'utilisation du cache CPU) du code système. CFAR introduit les distillats de mémoire, une représentation intermédiaire qui contient toutes les informations pertinentes sur la manière dont un programme accède à la mémoire et rien d'autre. CFAR extrait automatiquement les distillats de mémoire depuis du code système, et permet aux développeurs d'utiliser un distillat pour répondre à des questions spécifiques sur l'utilisation du cache. Nous démontrons que CFAR permet non seulement aux développeurs d'identifier les entrées qui conduisent à une utilisation inefficace du cache et à des vulnérabilités de sécurité dans leur propre code, mais également de raisonner sur l'impact de l'utilisation de code tiers sur la performance.

Contents

List of Figures

List of Tables

Setting the stage Part I

1 Introduction

Large and complex computer systems such as those that power Google's web search, Amazon's e-commerce, or Netflix's video streaming are an integral part of everyday life for billions of people. More than 5.1 billion people [177] are active internet users today, which results in more than 3.5 billion Google search queries being made [176], 15 million e-commerce packages being shipped [174], 1 billion hours of video streaming content being watched [249], and 350 billion emails being exchanged on a daily basis [175].

To build such complex systems *correctly*, i.e., ensure that they provide the desired functionality, developers need *semantic interfaces* (e.g., header files, documentation, and/or specifications). Such complex systems typically consist of numerous components, each of which is built by a different team of developers. Since semantic interfaces provide succinct, human-readable descriptions of the functionality that a piece of code or system component provides, developers can quickly use, build upon, and deploy code that was written by others while being confident that the system as a whole correctly implements the desired functionality.

However, given how integral such systems are to everyday life, it is no longer sufficient for them to be correct; they are also expected to be *interactive*, i.e., deliver consistently low latencies to provide a seamless user experience. Failure to provide consistently low latencies can directly impact revenue: for example, Amazon is known to lose 1% of sales for every additional $100ms$ in latency [239], while a $500ms$ delay in Google's search results causes a 20% drop in traffic and thus advertisement revenue [225]. Similarly, brokers can lose up to $4 million per millisecond if their platform is $5ms$ behind the competition [239] and a 2017 Akamai study showed that a $100ms$ delay results in a 7% decrease in the number of customers that complete transactions [4].

There exists no equivalent *latency interface* that developers can use to reason precisely about the expected latency behavior of code. Engineers today reason about latency in terms of envelopes (e.g., "runs in $O(n)$ time") and benchmarks, which implies that they deploy their system without understanding the entire spectrum of latency it can exhibit. As a result, systems frequently exhibit unexpected performance behavior [83, 103, 108] in production when the input workload or runtime environment changes in unpredicted ways which results in missed performance targets and a perpetual need to fix performance bugs [101, 130]. To give a sense of how much work this is, half the configuration-related patches in open-source cloud systems are needed to fix performance issues, while Mozilla developers have had to fix from 5 to 60 performance bugs every month over the past 10 years [130].

So in this thesis, we answer the question(s): Can there exist a useful latency interface for systems code? i.e., an interface that summarizes the latency behavior of systems code, just like a semantic interface summarizes its functionality? What should such an interface look like to help developers reason precisely about the expected latency of not just their own, but also third-party code?

1.1 Reasoning about System Latency Today

To help developers reason about expected latency behavior, latency interfaces need to provide a balance between two, typically conflicting properties: *accuracy* and *readability* [147]. By accuracy, we mean the ability to describe latency *completely* (for every possible input and runtime environment) and *precisely* (with a small error). By readability, we mean that the representation should be *smaller* than the system implementation and as *abstract* as possible, i.e., summarize latency in terms of primitives appropriate for a semantic interface of the system, and reveal implementation details only when necessary. Accuracy and readability are conflicting requirements because improving accuracy typically involves adding more detail, which makes the representation harder to read.

Widely-used representations such as asymptotic complexity bounds [26], upper bounds on worst-case on execution time [241] and probabilistic service level objectives (SLOs) [211] typically sacrifice accuracy for readability, i.e., there are many inputs for which they do not accurately describe latency.

Asymptotic bounds on time complexity (e.g., “runs in $O(n)$ time”) [26] describe the limiting behavior of the number of computational steps the program must perform as a function of the size of its input (n). While such bounds are easy to read (and hence widely used), they cannot provide developers with a precise understanding of latency in terms of wall clock time. This is because they ignore both the constants in the latency expression and factors such as the underlying hardware. Constants and hardware often end up being the dominant factor in wall clock time; for instance, it is common to trade off algorithmic complexity for improved locality (and thus lower constants) in memory and storage devices [72, 73, 242].

Upper bounds on worst-case execution time (WCET) [241] are widely used in the domain of real-time and safety-critical systems (e.g., control systems in airplanes [216, 236], cars [234] and industrial manufacturing [235]) where the timeliness of executing an operation is part of its semantic correctness. While such bounds are both easy to read and expressed in terms of wall-clock time, they are not very useful beyond real-time and safety-critical systems. This is because most systems are designed to be fast in the common case and make progress in the worst [147], and so worst-case latencies may be orders of magnitude higher than typical or median latencies and cannot be used to make informed development decisions.

Finally, latency SLOs [211] provide a target value or range for the statistical execution latency of the system (e.g., X percentage of requests take < Y time). In theory, SLOs overcome the primary limitations of both of the above representations since they reason about wall clock time and can be used to describe latency in scenarios other than the worst case. However, in practice, SLOs typically deteriorate to worst-case upper bounds. For instance, a recent paper from Google states that “*SLOs are aimed at bad outcomes, they are often far from the expected outcomes, and few customers would be happy if a system only met its contractual SLOs*” and calls for SLOs to take the expected workload into account [165].

In the absence of a readable and accurate representation, most developers resort to benchmarking a system in order to reason about its latency behavior, i.e., they treat the system implementation as its own interface. Benchmarking is not only a tedious process but is also error-prone, particularly when used to understand the latency behavior of code that the developer did not write themselves [115].

Fig. 1.1 summarizes the quandary that developers face when trying to reason about the expected latency behavior of systems code. They must either divine the expected latency behavior of systems code from representations that provide little information about latency for realistic workloads or download, build, write tests for, and run the code themselves to discover the expected latency. In contrast, they can reason about expected functionality just by reading the succinct semantic interface.

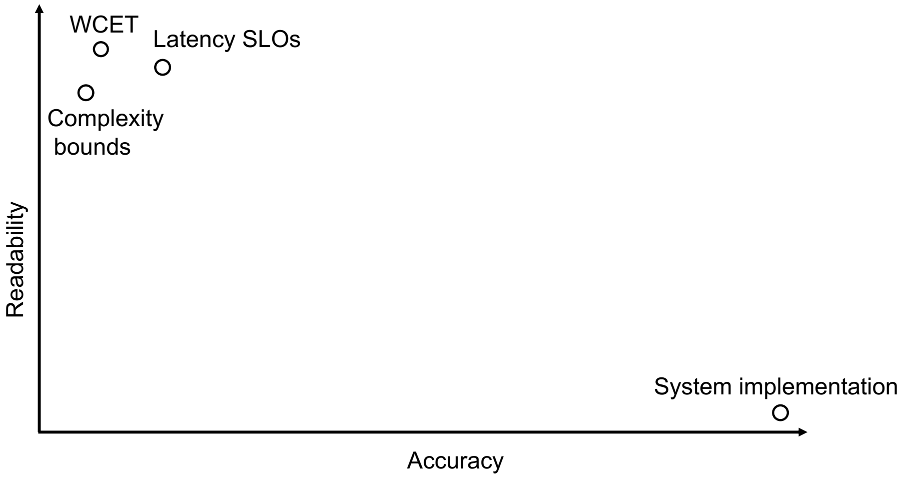


Figure 1.1: Existing representations of system latency. WCET refers to Worst-Case Execution Time. SLOs refer to Service Level Objectives.

1.2 Thesis Goals

The goal of this thesis is to develop representations and techniques that enable developers to reason about the expected latency behavior of systems code as easily as they reason about its expected functionality today.

By systems code, we refer to the software that bridges applications to the hardware and infrastructure that executes it. Systems code typically performs low-level operations (e.g., DMA), and the quintessential examples of systems code are operating systems, hypervisors, device drivers, and variants thereof (such as network functions running on kernel-bypass frameworks). Systems code plays a foundational role in the software stack; as a result, upper layers rely on it being optimized for speed and efficient resource usage. Languages traditionally associated with systems programming are C, C++, and Rust, languages that provide facilities for low-level memory manipulation and hardware access, while also allowing for higher-level abstractions when needed.

We advocate that achieving our goal requires systems code to have a latency interface that describes its expected latency behavior and related side effects in an accurate and readable manner, just like the code's semantic interface describes its expected functionality and related side effects.

We take a three-part approach to realize latency interfaces for systems code.
In the first part of the thesis (§1.2.1), we show how to design operating system schedulers that enable datacenter applications (such as the ones underlying web search and e-commerce) to provide predictable latency behavior, i.e., latency that conforms to a well-defined interface and is hence easy to reason about. Here, we work with existing representations of latency, and so define the latency interface using tail latency SLOs, which is the standard practice in datacenters today. We contribute Concord, an efficient scheduling runtime for datacenter applications that carefully eliminates sources of overhead that plague state-of-the-art schedulers and lead to worse tail latency. While Concord achieves its goals, our experiences in this part of the thesis motivated us to develop interfaces and techniques that provide a more precise understanding of latency behavior and related micro-architectural effects.

In the second part of the thesis (§1.2.2), we propose a new representation for latency interfaces—simple, executable programs that accepts the same input(s) as the system and output its processing latency—that we believe is best suited to describing the expected latency behavior of systems code in an accurate and readable manner. We concretize this representation in the domain of network functions (NFs) and present LINX, a program analysis tool that automatically extracts latency interfaces from NF implementations.

Finally, since an interface should describe related side effects [210], we present CFAR, a technique and tool that allows developers to reason precisely about micro-architectural side effects (specifically CPU cache usage) of systems code (§1.2.3). Our work on CFAR demonstrates that simple, executable programs can summarize not only the processing latency but also any related side effects of systems code in an accurate yet readable form.

1.2.1 Ensuring that Datacenter Applications Meet Their Latency Objectives

Datacenter applications are expected to meet strict microsecond-scale tail latency SLOs (e.g., 99^{th} percentile latency should be $< X\mu$s) to remain interactive for end users [134, 185, 193]. Bounding tail latency (typically 99^{th} percentile or higher) is necessary because of the "tail-at-scale" problem [65]: given that such applications distribute each user request across thousands of servers with the end-to-end response time determined by the slowest individual response, it becomes highly likely that at least one server will incur a high percentile latency that will end up determining the end-to-end response time. Additionally, for the application as a whole to remain interactive and respond to user requests within tens of milliseconds, each individual server should process requests within ten to a few hundred microseconds [22, 134, 193].

We focus on ensuring that such applications meet their tail latency SLOs in a cost-effective manner, i.e., while sustaining high throughput per server. Taking cost into account is necessary since the easiest way to ensure that tail latency SLOs are met is to overprovision the number of servers required and have each server serve a smaller fraction of the incoming load [66, 154]. However, this overprovisioning wastes precious CPU cycles and is not economical for services that need to scale to billions of users [65, 134].

Since tail latency is dominated by queueing delay (not service time), we advocate for rethinking microsecond-scale scheduling and present Concord, an efficient scheduling runtime for microsecond-scale datacenter applications. Concord demonstrates that careful *approximation* (as opposed to canonical implementation) of theoretically optimal scheduling policies enables new microsecond-scale mechanisms that provide significant throughput benefits while ensuring the applications continue to meet the same tail latency SLO. Concord introduces new mechanisms that carefully approximate two scheduling policies to reduce their implementation overhead and thus improve application throughput: (1) Preemptive scheduling, which is necessary to prevent long-running requests from starving short-running ones but introduces the overhead of having to context-switch between requests, and (2) Single-queue scheduling, which is necessary to ensure optimal load balancing of incoming requests but introduces cache coherence overheads due to requests being passed from the CPU core maintaining the single queue to the one that processes the request.

Designing and implementing Concord made us acutely aware of the two main challenges that system developers face when trying to reason about the latency behavior of systems code. First, reasoning precisely about the differing latency behavior of different execution paths through the code (e.g., processing of different request types) is hard; a direct consequence of this is that developers are forced to rely on high-overhead, black box techniques such as interrupts to preempt long-running requests. Second, developers possess little visibility into the micro-architectural (specifically CPU cache) behavior of their code. As a result, cache issues (such as the ones associated with single queue scheduling) often go unaddressed even in state-of-the-art systems focused on maximizing performance [66, 134].

While it is feasible to overcome these challenges (as we did) using the traditional systems approach—carefully measure, analyze, optimize, repeat—it is a painstaking process that needs to be repeated for every system. Hence, in the next two parts of the thesis, we focussed on developing interfaces and techniques that provide a more precise understanding of latency behavior and related micro-architectural effects.

1.2.2 An Accurate, Readable Representation for System Latency

Here, our goal is to answer the question: *Is it feasible to summarize the processing latency of systems code for all possible inputs in an interface that is simultaneously accurate and readable?*[1]

Summarizing processing latency in an accurate yet readable interface is more challenging than doing the same for semantics because the system's deployment environment (e.g., the hardware it runs on) typically has a greater impact on its latency than on its semantics. This is because systems—both hardware and software—typically employ strong semantic modularity but close to no latency modularity. For instance, a `mov (%ebx),%eax` instruction has the same semantics on all `x86` machines but when it comes to latency, the modularity is much weaker: the time to execute `mov (%ebx),%eax` can vary by orders of magnitude depending on several factors such as the micro-architectural specifics of the machine and other processes executing on the same machine.

We propose that the *latency interface* of a system be a *program* that accepts the same inputs as the system and outputs how long the system would take to process the given input. System developers are familiar with code, allowing them to quickly read such interfaces and understand the latency behavior of the system without having to run it. Accepting the same input(s) enables the interface to describe performance for different expected workloads, something that upper/lower bounds cannot do.

We introduce three key ideas that enable latency interfaces to summarize latency in a manner that is simultaneously accurate and readable: First, the interfaces describe latency not as concrete numbers but as formulae of random variables that we call *latency-critical variables (LCVs)*. LCVs summarize the impact of latency of all factors other than the current input (e.g., prior inputs, system state, configuration, and runtime environment) on latency. Representing latency as formulae containing LCVs enables the interface to succinctly summarize the latency for arbitrary workloads. Second, we introduce the concept of a *latency resolution* which specifies the smallest change in latency that the interface captures. At a given resolution the interface only reveals those implementation details that cause latency variability greater than the resolution, thus eliminating unnecessary details and giving developers who do not care about cycle-accurate latency predictions (whom we expect to be the majority) explicit control over the trade-off between accuracy and readability. Finally, we introduce the concept of *deployment-specific* interfaces, i.e., interfaces tailored to a particular deployment. Deployment-specific interfaces enable developers who merely want to use the system in a particular deployment environment to maximize readability while retaining all information relevant to latency behavior in that environment.

To realize latency interfaces in practice, we concretize our proposal in the context of software network functions (NFs)—in-network packet processing applications such as load balancers, firewalls, and NATs— and present LINX (Latency INterface eXtractor), a program analysis tool

[1] We focus on systems code that provably terminates to avoid running into the Halting problem [224]

that automatically extracts latency interfaces from NF implementations. LINX takes as input NF code written in C and outputs latency interfaces in the form of simple Python programs. We demonstrate that LINX-extracted interfaces are 2-3 orders of magnitude simpler than the corresponding NF implementations, can predict NF latency across deployments with an average error of < 8% and enable system engineers to identify performance regressions, diagnose and fix performance bugs, and identify the latency impact of NIC offloads.

1.2.3 Reasoning About Latency Side Effects

Since a program's semantic interface describes not only its expected output(s) but also any expected side effects [210], i.e., modifications to shared state that may lead to differences in externally observed behavior, an equivalent latency interface should also describe any expected latency side effects in addition to the processing latency described above.

Latency side effects arise due to shared micro-architectural state. Since all programs running on the same CPU core (e.g., caller and callee, application and operating system) share core-local micro-architectural resources (e.g., data and instruction caches, TLB, branch predictor, etc.) calling into a piece of code has not only a direct impact on latency (via the execution latency of callee) but also an indirect cost that depends on how the callee perturbs shared micro-architectural state. This indirect cost is a frequently observed source of latency variability; for example, FlexSC [215] showed how a system call can take up to 3× longer depending on the invoking program's micro-architectural resource usage, while the invoking program may run up to 4× slower after the system call, depending on the system call's micro-architectural resource usage.

We focus on a dominant source of micro-architectural side effects, namely the CPU cache. Our goal is to enable developers to answer frequently asked questions about how a piece of systems code interacts with the cache, such as: How does the code's cache usage vary with workload (e.g., as a function of the number of network connections)? Which workloads make the working set exceed the cache size? Existing performance-analysis tools such as profilers [35, 153, 189, 230] cannot directly answer the questions listed above, because they do not understand what the code does to the micro-architecture as a function of workload.

We present CFAR (Cache Footprint AnalyzeR), a tool that processes a piece P of systems code into answers to developers' questions about how that code uses the cache. CFAR's processing consists of two phases: In the former, CFAR takes as input the code and outputs an intermediate representation (a "distillate") that contains all the information on how the code accesses memory. In the latter, developers can write simple programs ("projectors") that use the distillate to compute answers ("projections") to specific questions about P's cache usage.

CFAR reinforces our belief that simple, executable programs are best suited to summarizing latency information in a manner that is simultaneously readable and accurate. Like our proposed latency interfaces, CFAR distillates and projections are represented as programs that take the same inputs as P and return the metric of interest. In particular, projections can be seen as interfaces that describe the latency behavior of P in terms of the metric defined by the user-given projector (e.g., the number of unique cache lines touched per network connection).

1.3 Thesis Statement

It is feasible for systems code to expose a latency interface that describes its latency and related side effects for all possible inputs, just like the code's semantic interface describes its functionality and related side effects. Simple, executable programs can summarize system latency in a manner that is simultaneously precise, complete, and human-readable, and thus useful to system engineers.

1.4 Thesis Contributions

This thesis makes the following contributions:

Efficient microsecond-scale scheduling in the datacenter

- We demonstrate that *approximating* (as opposed to canonically implementing) theoretically optimal scheduling policies lead to significant throughput benefits at microsecond-scale at negligible tail latency costs.
- We introduce *compiler-enforced cooperation*, a technique that enables preemptive scheduling at 4× lower overhead than the state of the art at microsecond-scale.
- We show how to design a *work-conserving dispatcher*, a thread that not only dispatches incoming requests to other (worker) threads but also contributes to application goodput.
- We design and implement Concord, a scheduling runtime for microsecond-scale datacenter applications that implements the above techniques. In comparison to the state of the art, Concord improves throughput by up to 52% for microbenchmarks and up to 83% for Google's LevelDB key-value store while meeting the same tail latency SLO.

Latency interfaces as succinct, executable programs

- We propose that the *latency interface* of a system be a program that accepts the same inputs as the system and returns its processing latency.
- We introduce *Latency-Critical Variables (LCVs)*, random variables that summarize the impact of prior inputs, system state, configuration, and runtime environment on latency. Representing latency as formulae containing LCVs (as opposed to concrete numbers) enables the interface to succinctly summarize the latency for arbitrary workloads.
- We introduce the *latency resolution* which specifies the granularity at which the interface describes latency and provides readers of the interface with explicit control over the trade-off between readability and accuracy.

- We introduce *deployment-specific* latency interfaces which enable engineers who did not write the code for, but merely want to use, a system to reason precisely about its latency in their specific deployment environments.
- We design and implement LINX, a program analysis tool that automatically extracts latency interfaces from software network function (NF) implementations. LINX-extracted interfaces are 2-3 orders of magnitude simpler than the corresponding NF implementations and can predict NF latency across deployments with an average error of < 8% while only taking minutes to extract. We demonstrate how LINX-extracted interfaces can be used to identify performance regressions, diagnose and fix performance bugs, and identify the latency impact of NIC offloads.

Automatically reasoning about how systems code uses the CPU cache

- We propose *memory distillates*; an intermediate representation for programs that contains all information relevant to how the program accesses memory, and discards everything else. Developers can query the distillate using simple programs that we call *projectors* to compute the answers to specific questions they have about the program's cache usage.
- We design and implement CFAR, a program analysis tool that automatically extracts memory distillates from system implementations and provides support for developers to write projections. The combination of distillation and projection ensures that CFAR provides engineers with succinct, yet accurate information about CPU cache usage which enables them to identify inefficient paths and security vulnerabilities in their own code and reason about the latency impact of incorporating third-party code into their systems.

1.5 Thesis Organization

The rest of the thesis is organized as follows:

- In Chapter 2 of this introductory part of the thesis, we provide the necessary background on existing representations for system latency and techniques to infer latency properties from system implementations.
- In Part II of the thesis, we describe Concord, our scheduling runtime that enables datacenter applications to meet microsecond-scale latency objectives (as defined today) in a cost-effective manner (Chapter 3).
- In Part III, we first define our proposal for latency interfaces (Chapter 4), before presenting LINX, a tool that automatically extracts such interfaces from software network function implementations (Chapter 5), and CFAR a technique and tool that enables developers to automatically reason about the performance side effects of systems code (Chapter 6).
- Finally, in Part IV, we describe future research directions (Chapter 7) and conclude (Chapter 8).

1.6 Bibliographic notes

This thesis primarily builds upon the ideas presented in the following publications:

- Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. "Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling" In: *Symposium on Operating Systems Principles*. 2023 [126]
- Rishabh Iyer, Katerina Argyraki, and George Candea. "Performance Interfaces for Network Functions" In: *Symposium on Networked Systems Design and Implementation*. 2022 [123]
- Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. "Performance Contracts for Software Network Functions" In: *Symposium on Networked Systems Design and Implementation*. 2019 [124]
- Rishabh Iyer, Katerina Argyraki, George Candea, and Sylvia Ratnasamy. "Automatically Reasoning about how Systems Code uses the CPU Cache" *Currently in submission*.
- Rishabh Iyer, Jiacheng Ma, Katerina Argyraki, George Candea, and Sylvia Ratnasamy. "The Case for Performance Interfaces for Hardware Accelerators" In: *Workshop on Hot Topics in Operating Systems*. 2023 [125]

2 Background

In this chapter, we provide the necessary background on existing representations for system latency (§2.1) and techniques to infer latency properties from system implementations (§2.2).

2.1 Representing System Latency Today

In this section, we detail existing representations for system latency from both academia and industry and show how each representation provides a different balance between *completeness* and *precision*. Recall that we define completeness as the ability to describe latency for *every possible* input, system state built up by prior inputs and hardware platform that the code is executed on, and define precision as the ability to describe latency for a *particular* input, system state, hardware with small error. Table 2.1 summarizes the balance that each representation provides that we explain throughout the rest of this section.

2.1.1 Asymptotic Bounds on Time Complexity

Asymptotic bounds on time complexity (e.g., this code runs in $O(n^2)$ time) are mathematical descriptions of how the number of computational steps that a program must perform grows with the size of its input (n). While such descriptions are typically used to provide upper or lower bounds—using the Big-O and Big Omega notations, respectively [26]—they can also be used to describe the average time complexity for a given probability distribution of inputs and the amortized complexity for a sequence of program invocations.

Asymptotic bounds were designed, and continue to be the gold standard, for comparing the growth rate of *algorithms* (and not *implementations*) as the size of the input grows arbitrarily large. As a result, they ignore both constants and all non-dominant terms in the latency expression (e.g., $O(n^2 + 10n + 100) = O(n^2)$), since both become irrelevant when the size of the input approaches infinity.

Ignoring constants and non-dominant terms causes such bounds to be imprecise with respect to metrics such as machine instructions or wall clock time, which are the metrics that systems developers ultimately care about. Constants, in particular, often end up being the dominant factor in wall clock time especially for realistic input sizes; for instance, it is common to trade off algorithmic complexity for the lower constants provided by improved locality in memory and storage devices [72, 73, 242].

2.1.2 Worst Case Execution Time

Worst case execution time (WCET) [241] provides an upper bound on a program's execution latency in terms of wall clock time and is widely used in real-time and safety-critical systems. For example, in order to comply with aviation regulations, airplane manufacturers must verify tight WCET bounds on components such as the flight control and avionics systems to ensure timely responses to external events and timely communication between the airplane and air traffic control, respectively [236]. Similarly, automotive vehicles must have verified WCET bounds for the anti-lock braking system (ABS) to ensure that it can respond in time to prevent skidding [234].

However, WCET bounds are typically only useful in real-time or safety-critical systems where the timeliness of executing an operation is part of its semantic correctness. This is because such systems are designed to meet a single latency objective, namely a bounded worst case, and the observed latency in other scenarios is of little interest. In contrast, most systems code has a very different design philosophy and is designed to be fast in the common case at the cost of being slow(er) in the worst case [147]. A classic example of this is branch prediction in modern processors, where instead of using the predicted value of the branch as only a hint and incurring similar latencies for both correctly and incorrectly predicted branches, modern processors are optimized for the common case in which branches are predicted correctly at the cost of being 20× slower in the worst case where the prediction is incorrect [120]. Given that for most systems, the worst-case latency can be significantly higher than typical or median latencies, developers cannot make informed decisions based solely on a worst-case bound.

So in summary, while WCET bounds can be precise for the absolute worst case, they sacrifice completeness and provide little/no information for realistic workloads.

2.1.3 Benchmark Scores

Benchmark scores represent performance as a set of numerical metrics (e.g., number of transactions per minute in TPC-C [227], end-to-end training time in Dawnbench [63], etc). These numbers are typically obtained through a standardized testing process, for instance, by running the program/system on a predefined set of inputs. Benchmark scores were primarily designed to compare multiple implementations of the same functionality (e.g., different relational databases, different CPUs that implement the x86 ISA, etc) and hence represent performance as easy-to-compare numbers.

While benchmark scores are widely used to quantify the performance of software in many domains (e.g., TPC-C,YCSB for databases [52, 227], Dawnbench for machine learning training [63], Octane, Jetstream [129, 178] for Javascript engines), they have several shortcomings. First, they cannot help developers understand the expected performance for *their* workload since they do not describe performance as a function of the workload, but only as opaque numbers. As a result, developers must either reason about how similar their workload is to the ones in the benchmark and extrapolate, or benchmark the program themselves which defeats the purpose of the benchmark score. Second, benchmark scores are typically only used when the functionality being implemented has gained significant maturity to have a representative set of workloads (e.g., transaction processing systems or the x86 ISA). This is not the case for most systems code that developers write on a day-to-day basis. Finally, benchmark scores themselves are not always reliable, since they depend on the environment in which the code runs. This has led to "benchmark wars" where different organizations claim superiority based on differing benchmark results for the same software and same inputs [214].

So, while benchmark scores are more complete than WCET estimates since they describe latency for all inputs in the benchmark, their completeness is restricted to just the benchmark since they provide no predictive capabilities.

2.1.4 Service Level Objectives

A latency service level objective (SLO) consists of two parts: (1) An easily measurable latency metric such as median or 99^{th} request latency; this is referred to as the latency service level indicator (SLI), and (2) a target value or range. So, latency SLOs take the form: "SLI <= target" or "lower bound <= SLI <= upper bound" [211].

The above format allows SLOs to represent latency in a flexible manner since SLIs can be used to distinguish between different percentile latencies (e.g., median vs tail) and different inputs. For instance, one can envision a detailed SLO that takes the form: "50% of reads to key-value store X will take < $1\mu s$, 99% of reads will take < $100\mu s$, and 99% of writes will take < $1ms$."

However, one key drawback of how SLOs are typically formulated arises from the preference for externally-visible and easily-measurable SLIs. Such SLIs are preferred because many SLOs are a part of service level agreements (SLAs) which consist of an SLO and a consequence (e.g., "if 50% of reads take > $1\mu s$, then X will pay Y an amount Z"), and in such cases, the SLI needs to be easily measurable so that it can hold water in a court of law. However, this results in SLIs typically being functions of only the system's inputs and outputs and not the harder-to-measure internal state, which makes SLOs incomplete with respect to system state. To ensure that the SLOs are not violated due to this incomplete understanding of state, the latency targets typically deteriorate to near-worst-case values in practice to account for all possible system states.

Nevertheless, we draw significant inspiration from SLOs in this thesis. This is evident both in how we chose them as a starting point for latency interfaces in Chapter 3 and how LINX's deployment-specific interfaces resemble the detailed SLO from above when they concretize the expected system state in a particular deployment environment (§5.4).

2.1.5 Performance Annotations

Performance annotations [201] describe a method's latency as a a set of ⟨input/global-variable constraints, latency formula⟩ tuples, where each formula is a mathematical function of the method's input and/or global variables.

Performance annotations are more complete than typical SLOs since the tuples include constraints about all system state that is maintained as explicit global variables. However, this is still insufficient since latency often depends not only on explicit global variables but also on implicit/ghost variables [106, 107].

As an example, consider a simple lookup operation implemented as a `while` loop that iterates over nodes in a linked list until the `next` pointer is null (e.g., `while(node.next){...}`). Here, the number of iterations (and hence latency) depends on the number of nodes traversed which is not stored as an explicit global variable and so will not be captured by performance annotations. Finally, since the constraints in each tuple do not take into account the underlying hardware, performance annotations possess no predictive power across different hardware platforms either.

We perform an in-depth evaluation of performance annotations in §5.7 and show how they fail to capture the impact of most system state and hardware on latency.

2.1.6 Summary

Table 2.1 summarizes the above representations and shows the balance between completeness and precision that each provides. In contrast, our proposed representation for latency interfaces as simple, executable programs achieve both. In a nutshell, this is because our programs include constraints in terms of not only the current input and global variables (like performance annotations) but also arbitrary functions of system state and hardware to achieve completeness. Finally, just like benchmark scores, latency interfaces are precise because they rely on targeted ground truth measurements derived from running the program. We elaborate on how our proposed representation is simultaneously precise and complete in Chapter 4.

Representation	**Complete**			**Precise**
	Input parameters	**System state**	**Hardware**	
Time complexity bounds	Yes	Yes	No	No
WCET	No	No	No	Yes
Benchmark scores	No	No	No	Yes
SLOs	Yes	No	Yes	No
Performance annotations	Yes	Partially	No	No
Latency interfaces	Yes	Yes	Yes	Yes

Table 2.1: Comparison of the precision and completeness of existing approaches to representing system latency. System state refers to the state maintained by the system implementation in software. Performance annotations are partially complete with respect to system state since they can take into account state maintained as explicit, global variables.

2.2 Inferring Latency Properties from Code

Having discussed existing representations for latency behavior, we now provide background on techniques that infer latency properties from system implementations and can thus be used to populate one's representation of choice. We focus on techniques that infer latency properties *before* the system is deployed in production, and do not discuss techniques that rely on runtime monitoring and verification of the system [86, 139, 167, 202, 243].

We start by discussing black box techniques (§2.2.1, §2.2.2) i.e., techniques that do not analyze the internal workings of the system, before moving on to white box ones (§2.2.3, §2.2.4). We conclude by describing how this thesis builds upon each of these techniques (§2.2.5).

2.2.1 Traditional Profiling

The traditional approach to inferring the latency properties from code is to run it using a profiler [35, 153, 230]. Profilers typically take as input the code (as a binary), a workload, and a list of metrics of interest (e.g., CPU cycles, instructions executed, cache misses, etc). They then run the binary on the workload and output a detailed breakdown of the metrics of interest.

Profilers are widely used for three reasons: First, because they run the code directly on the hardware and do not rely on modeling of any kind, they provide ground truth latency information for the given inputs. Second, since they make no assumptions about the provided code, they can be used for almost any systems code. Finally, modern profilers make it easy for developers to scrutinize and reason about latency in terms of not only wall clock time but also more nuanced micro-architectural metrics such as the number of cache misses.

However, the information provided by profilers is limited to the inputs provided, and they offer no predictive capabilities for other possible inputs. So, the utility of the profiler effectively hinges on the developer's ability to provide a representative set of inputs. Providing a representative set of inputs to understand latency (or more generally performance) is hard because performance suffers from the "large input problem" [157, 180], i.e., the fact that unexpected performance behavior often manifests only when input size exceeds some limit that may seem arbitrary to those who are not intimately familiar with the code. So, designing a test suite that completely covers a system's performance behaviors is hard, and developers don't even have well-defined coverage metrics. For example, line coverage is used as a proxy for coverage of semantic behaviors; performance profiling does not even benefit from such an approximate metric.

2.2.2 Trend Profiling

Trend profiling [100], algorithmic profiling [253], input-sensitive profiling [53,54] and Freud [201] improve upon traditional profilers by inferring trends (in terms of the inputs and global variables) based on the results of executing the provided workload, to provide predictive capabilities for unseen workloads. An example trend can be: the latency of function f is $n^2+10n+100$ where n is the length of the input array.

These approaches work as follows. First, they automatically instrument the provided program binary to record the values of all input features and global variables. Note, they only record features stored as explicit program variables, they do not add instrumentation to compute implicit ones. Then, they run the program on the provided workload, record the instrumented features and rely on machine learning techniques (typically linear regression [100, 201]) to automatically infer trends from the recorded data.

While trend profiling provides greater visibility into the expected latency behavior of code when compared to traditional profiling, it suffers from three main limitations. First, just like traditional profiling, the trends are inferred based entirely on the provided workload, so the onus is once again on the developer to provide such a workload. Second, since trend profilers do not record implicit variables, they often cannot capture the impact on latency of most system state (e.g., the implicit length of a linked list that is calculated as the number of nodes traversed until the `next` pointer is null). Finally, since trend profiling is a black box technique and does not analyze what the code does, it often cannot differentiate between correlation and causation. For instance, when we evaluated our work against a state-of-the-art trend profiler [201] and tried to extract a trend for a hash table lookup using a workload where every key was mapped to the same hash value, the trend profiler concluded that the runtime was determined by the occupancy of the map, as opposed to the number of collisions encountered by the input key.

We perform a detailed evaluation of state-of-the-art trend profilers in §5.7 and demonstrate that the above limitations prevent them from accurately capturing the impact of most system state on latency.

2.2.3 Discovering Adversarial Workloads

There exists a large body of work that focuses on discovering "adversarial inputs", i.e., inputs that incur high latency, to help developers find and fix potential performance bugs and scrutinize the performance of their code when under attack.

The first work on this topic discovered such adversarial workloads by manually scrutinizing the code. Crosby and Wallach were the first to demonstrate that adversarial inputs that exploit algorithmic complexity vulnerabilities can lead to denial-of-service (DoS) attacks [58]. Subsequent work [3, 212] went one step further by manually generating adversarial inputs that lead to not only worse algorithmic complexity but also exhaustion of resources such as heap memory.

More recent work focuses on discovering adversarial inputs *automatically* and relies on automated program analysis techniques such as fuzzing or symbolic execution to search for adversarial inputs.

SlowFuzz [190] and PerfFuzz [149] demonstrate how fuzzing can be used to automatically discover adversarial inputs to individual methods and data structures. Fuzzing is a program analysis technique that involves running the program on a diverse set of inputs, called seeds, to discover properties of interest. In fuzzing, the program is first run on a set of seeds, with the program's behavior for those seeds used as feedback to inform the next set of inputs. This process is repeated until the developer is satisfied, or a time budget is reached. The key to effective fuzzing is deciding the right aspects of program behavior to use as feedback. For example, SlowFuzz used the "total number of instructions executed" as its feedback metric, and PerfFuzz showed that using the "number of previously unseen branch instructions executed" as the metric leads to better results.

WISE [34], PerfPlotter [42], Violet [112], and Castan [188] demonstrate how symbolic execution can lead to the discovery of adversarial inputs; since we also rely on symbolic execution in this thesis, we describe it in detail here. Symbolic execution (SE) [140] is a program analysis technique that automatically explores feasible execution paths of a given program. It uses a special interpreter, called a symbolic execution engine (SEE). The SEE can make any input or variable (including a pointer) symbolic, i.e., assign to it a symbol representing many possible concrete values. As the symbolic inputs propagate through the program, the SEE keeps track of the resulting symbolic expressions. For example, suppose a program takes as input an integer `in`, the SEE can make `in` symbolic, assigning to it a symbol α that represents all possible integer values; if the program at some point assigns to an integer variable `x` the value `in+1`, then `x` also becomes symbolic with value $\alpha + 1$. If the program reaches a conditional branch predicated on a symbolic value, the SEE explores both paths and keeps track of the *constraints* that led down each path, such as $\alpha < 0$. The SEE uses a constraint solver [64, 95] to ensure that it explores only feasible paths and to identify the class of inputs that triggers each one. However, SE typically cannot *exhaustively* analyze programs (identify all feasible paths) due to path explosion [30], i.e., scenarios in which the SEE must explore a large, potentially unbounded number of paths. Path explosion typically occurs in the presence of loops and/or due to symbolic pointers in code that maintains significant state. Overcoming path explosion is an active area of research with recent SEEs having introduced various techniques such as state merging [146], loop-extended symbolic execution [203], loop summaries [99, 247], loop invariants [127], and symbolic abstract transformers [143].

WISE [34] uses symbolic execution to find adversarial inputs as follows: First, WISE exhaustively explores all program paths for small inputs to find worst-case paths, with the small input sizes ensuring that path explosion is limited. Then, it uses these worst-case paths as a heuristic to guide an incomplete search over inputs of larger sizes. While WISE's idea of splitting the search space based on input size is clever, in practice, it often runs into the "large input problem" [157, 180] where unexpected performance behavior often manifests only when input size exceeds some limit. This results in WISE's search heuristics often being insufficient to detect poor performance for large inputs.

Violet [112] uses symbolic execution to automatically detect combinations of configuration options that lead to poor latency. Focusing only on the latency impact of configurations (and not the input) allows Violet to sidestep many path-explosion-related challenges.

Finally, Castan [188] automatically generates adversarial inputs for network functions (NFs).

Castan's key contribution is taking into account the behavior of not only the software but also the underlying hardware to direct SE's search process. Given that NF latency is significantly impacted by last-level cache (LLC) misses [68, 156], Castan leverages an empirically-derived model of the LLC to guide its search for adversarial inputs.

All the above techniques and tools are useful since they help developers find and fix performance bugs in their code that can stem from both algorithmic complexity and micro-architectural resource usage. However, they have two main shortcomings: First, they provide no guarantees, i.e., they only extract inputs with "bad" performance but cannot prove that these inputs are the "worst" performing ones. Second, the cost models that guide their search are typically highly domain-specific and rarely generalize. For instance, Castan's cost model is intrinsically tied to how network functions interact with the LLC and does not generalize.

2.2.4 Verifying Latency Behavior

Here, we discuss two prior approaches to formally verifying the latency behavior of an implementation before it is deployed. The key difference here is that these techniques (unlike the ones above) provide provably correct guarantees about latency behavior.

Verifying Upper Bounds on Worst Case Execution Time (WCET)

The need for formally verified upper bounds on WCET in safety-critical and real-time systems such as airplanes [236] and cars [234] has motivated a plethora of prior work on computing increasingly precise, verified upper bounds on WCET (summarized in [241]).

In this section, we focus on *static* approaches to computing upper bounds for WCET. These approaches do not run the code on real hardware or in a simulator but instead rely on detailed timing models of the hardware. Approaches that eschew such models and instead rely on empirical observations also exist, but since empirical observations cannot compute guaranteed upper bounds that will never be violated, measurement-based approaches are rarely used in safety-critical or real-time systems [241].

Static approaches typically rely on a three-step process to compute an upper bound for WCET[1]: (1) First, the "flow analysis" step extracts a precise control flow graph (CFG) for the program, computes loop bounds for all loops in the code, and determines any infeasible paths through the CFG. All three tasks are performed in a conservative manner (no feasible execution paths are eliminated), and when necessary, additional annotations from the developer are required (e.g., to resolve indirect function calls and compute upper bounds on hard-to-analyze loops). (2) Then, the "low-level analysis" step computes the worst-case execution latency for individual basic blocks using a precise timing model of the target hardware, and (3) Finally, the "calculation" step models the program as an Integer Linear Program (ILP) and constructs an objective function

[1] The terms used for each of these steps are those used in existing literature on WCET analysis

with variables representing the number of times each basic block is executed, weights for each variable corresponding to the WCET for each basic block and constraints representing feasible execution paths through the CFG. This formulation is fed to an ILP solver which computes an upper bound on the WCET of the entire program by maximizing the objective function.

However, static WCET computation techniques are crippled by their reliance on a precise, worst-case timing model of the underlying hardware (needed for Step 2 above). Since such models must be painstakingly derived for each processor, state-of-the-art WCET tools only provide support for a handful of simple embedded processors that were released at least 15 years ago [241]. Further, since hardware is designed to make the common case fast and not the worst case predictable [147], the bounds computed can be an order of magnitude larger than the observed WCET [29]. In §5.5.2, we evaluate the techniques we propose in this thesis against state-of-the-art static WCET ones and demonstrate that this is indeed the case.

Verifying Bounds on Resource Usage

There exists considerable prior work on verifying bounds on a program's resource usage (e.g., computational steps, heap usage).

This line of work was pioneered by Wegbreit in 1975 [237] who proposed to represent such bounds as recurrence relations. Wegbreit proposed to extract such bounds in two steps: first extract recurrence relations from the program and then compute closed-form expressions for each extracted recurrence. Wegbreit implemented his analysis for LISP programs but mentions that it "can only handle simple programs". The most complicated examples that he provides are a function that reverses lists and a function that computes the union for sets represented as lists.

The COSTA project [5, 6] revisited this idea of computing bounds as recurrence relations for Java bytecode and use abstract interpretation [57] to the compute recurrence relations. However, they only focus on recurrence relations defined on program inputs, and so cannot compute precise bounds for stateful code such as data structures.

The SPEED project [106, 107] demonstrates how to compute such bounds for C++ programs (including snippets of data structures from the C++ standard template library) with loops and recursion. SPEED computes bounds in terms of not only inputs to the program but also "user-defined quantitative functions" that can characterize implicit program variables such as the length of a linked list and the depth of a tree. To compute these bounds, SPEED instruments the program with counters for each user-defined quantitative function and then uses off-the-shelf abstract interpretation-based linear invariant-generation tools to infer invariants on these counters automatically.

Finally, an alternative approach focuses on automatically computing bounds on the amortized resource usage (i.e., resource usage across a sequence of inputs). This approach, known as automatic amortized resource analysis (AARA) [111] typically computes such bounds at compile time using type inference instead of an abstract-interpretation-based analysis. AARA relies on the potential method [221] and assigns a potential function to each data structure that maps the state of the data structure to a non-negative number. It then computes the amortized cost of an operation as the sum of its execution cost and the change in potential of the data structure caused by it.

While much of the prior work on AARA (summarized in [111]) has focused on computing amortized heap usage for first-order functional programs, there exists work that extends the technique to imperative programs as well, including reasoning about other metrics such as WCET. For example, Jost et. al [131] demonstrated how AARA can be used for bounding the WCET in terms of CPU cycles by augmenting AARA with an analysis of the WCET for each basic block in the binary. That said, a key limitation of AARA is that it requires the metric of resource usage to be *context-independent*; i.e., the cost of each instruction must be statically computable and must not depend on instructions that come before or after it. While this assumption does not hinder WCET analysis since one can assume the worst-case cost for each instruction, it prevents AARA from being applied to reason about other percentile latencies since the latency of executing each instruction on modern hardware depends significantly on the instructions that came before it [120].

2.2.5 Takeways

The techniques and tools presented in this thesis draw significant inspiration from those presented above, particularly the SPEED project, symbolic-execution-based tools, and profilers. The latency-critical variables (LCVs) that we propose in Chapter 4 are similar to SPEED's "user-defined quantitative functions", although we generalize the idea to include functions of not just software state but also hardware state. Like WISE, Castan, and Violet, we rely on symbolic execution (SE) in both LINX (Chapter 5) and CFAR (Chapter 6), although we do not rely on domain-specific cost models to direct SE since we explore all execution paths through the program[2]. Finally, like profilers, LINX relies on (partially) running the program to discover ground truth latencies; this is because our goal is not to compute guaranteed upper bounds on execution latency for safety-critical code, but rather to provide *useful* predictions for the expected latency of more general systems code.

[2] Naturally, this introduces scalability limitations, we mention them in §5.2

Part II

Building Systems with Predictable Latency Behavior

3 Meeting Microsecond-Scale, Tail Latency Objectives while Sustaining High Throughput

As a first step towards enabling developers to efficiently reason about the latency behavior of systems code, we study the problem of ensuring that datacenter applications provide *predictable* latency, i.e., latency that meets a well-defined target and is hence easy to reason about. In this chapter, we work with existing representations of latency, and so define latency targets in terms of tail latency SLOs, which is the standard practice in datacenters today [65].

We focus on ensuring that datacenter applications meet their tail latency SLOs in a cost-effective manner, i.e., while sustaining high throughput per server. Taking throughput into account is necessary while reasoning about tail latency SLOs since the easiest way to ensure that the latter are met is to overprovision the number of servers required and have each server serve a smaller fraction of the incoming load [66, 154]. However, this overprovisioning wastes precious CPU cycles and is not economical for services that need to scale to billions of users [65, 134].

In the rest of the chapter, we first define the problem and show why it mandates efficient microsecond-scale scheduling (§3.1), before performing a systematic analysis of the throughput overheads introduced by state-of-the-art microsecond-scale schedulers (§3.2). We then present the design (§3.3), implementation (§3.4) and evaluation (§3.5) of Concord, our proposed scheduling runtime that addresses each source of overhead identified by our analysis. Finally, we discuss Concord's limitations and broader applicability (§3.6), summarize related work (§3.7), and conclude with lessons learned that serve as motivation for the rest of the thesis (§3.8).

3.1 Problem Definition

Datacenter applications such as web search and e-commerce are architected in a scale-out manner, with each user request being distributed across thousands of individual servers [21, 65]. These applications aggregate the responses from individual servers, and so the end-to-end response time is determined by the slowest individual response.

This scale-out architecture imposes stringent, microsecond-scale, tail latency service level objectives (SLOs) on the code running on individual servers [22, 65]. Bounding tail (typically 99^{th} percentile or higher) latency is critical since the large number of servers involved in processing a user request makes it likely that at least one will incur a high-percentile latency that will determine the end-to-end response time. Additionally, for the service as a whole to remain interactive and respond to user requests within tens of milliseconds, each individual server must process requests within ten to a few hundred microseconds [22, 134, 193]. These strict tail latency SLOs are only expected to get tighter over time [117, 135] as applications are being modularized into increasingly finer microservices [1, 94, 160, 199] and communication stacks are being offloaded to specialized hardware [12, 136, 222, 254].

Since the tail latency of a request at individual servers is dominated by its queueing delay (and not service time), state-of-the-art schedulers are optimized based on queueing theory results. Weirman and Zwart [240] show that there is no single scheduling policy that minimizes tail latency across all possible workloads; the First Come, First Served (FCFS) policy is optimal for light-tailed workloads, and Processor Sharing (PS) is optimal for heavy-tailed workloads. Additionally, single-queue scheduling improves tail latency when compared to multi-queue scheduling for both FCFS and PS policies. Since both light- and heavy-tailed workloads are common in production [66, 134], state-of-the-art microsecond-scale schedulers need to support both (1) preemptive scheduling to implement PS for heavy-tailed workloads, and (2) a single queue.

However, optimizing systems for tail latency inevitably sacrifices the maximum throughput they can sustain, with the sacrificed throughput only increasing as request service times grow shorter. For example, single-queue systems such as ZygOS [193] and Shinjuku [134] achieve lower maximum throughput than IX [24], an earlier system that had no tail-latency optimizations. Similarly, preemptive scheduling in Shinjuku imposes a 20% throughput penalty at a scheduling quantum of 5µs, and a 50% throughput penalty at a quantum of 2µs. Fig. 3.1 conceptualizes the trade-off faced by tail-optimized microsecond-scale systems: chasing tight bounds on tail latency makes such systems move from the blue to the orange curve, which results in them saturating sooner than their non-tail-optimized counterparts.

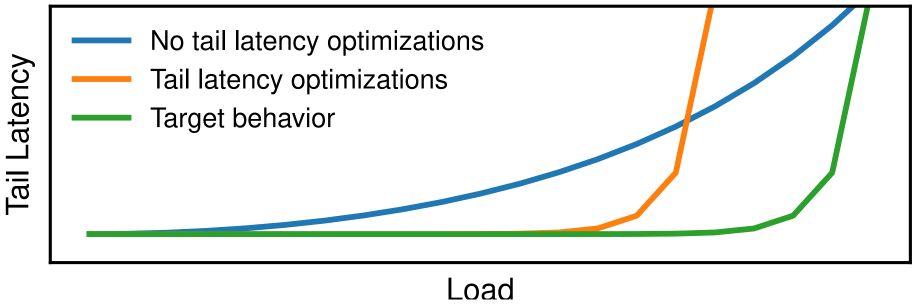


Figure 3.1: Abstract visualization of throughput overhead in state-of-the-art datacenter systems.

Systems optimized for tail latency also frequently sacrifice deployability and generic support for applications. For example, Shinjuku relies on non-standard use of virtualization hardware to achieve microsecond-scale preemption, but this precludes its deployment on VMs in public clouds. Similarly, Persephone [66], another state-of-the-art system, relies on non-blind scheduling, i.e., it requires prior knowledge of the application's service time distribution and is restricted to applications with request classes that have disjoint service-time distributions known a priori. However, this makes it ill-suited for the datacenter where blind policies are required to deal with heterogeneous applications [114].

So in this chapter, our goal is to build a microsecond-scale scheduler that (1) preserves the tail latency properties of the state-of-the-art by supporting both preemptive and single queue scheduling while (2) improving application throughput and (3) eschewing reliance on custom hardware or application-level assumptions. To do so, we do not introduce new scheduling *policies*, but instead introduce new *mechanisms* that implement existing policies with lower overhead. Since the saturation point of tail-optimal systems is determined exclusively by the amount of overhead in the system, reducing system overheads is both necessary and sufficient to reach ideal system behavior (green curve in Fig. 3.1).

3.2 Throughput Overheads at Microsecond-Scale

We now analyze the throughput overheads in state-of-the-art microsecond-scale schedulers that arise from implementing preemption and single queue scheduling, and show how they scale as timescales grow shorter. We use an analytical model to describe the sources of throughput overhead. This enables us to look beyond particular prior systems and instead reason abstractly about the trade-offs and optimality of multiple systems. We first introduce our system model (§3.2.1), and then use it to show how existing systems suffer from double-digit overheads at today's 5µs timescales and triple-digit overheads at tomorrow's 1µs timescales (§3.2.2).

Schedulers that implement a single queue fall in two categories: those that maintain a *physical* and *logical* single queue respectively. In the former [66, 134], one thread (the dispatcher) is dedicated to maintaining the queue and sending requests to the others, while in the latter [185, 193] there is no dedicated thread and idle threads *steal* requests from other threads, mitigating load imbalance. We focus (in this section) on single *physical* queue systems for two reasons: (1) The only prior work [134] that implements both a single queue and preemption—and is thus the most relevant baseline—employs such a queue, and (2) having a dedicated thread with global visibility of the entire system provides the flexibility to implement arbitrary queueing policies. We provide a detailed discussion of systems that maintain a single *logical* queue in §3.6.

3.2.1 System Model

We consider a system with 1 dedicated dispatcher thread and n worker threads, all of which are pinned to individual CPU cores. The dedicated dispatcher maintains the single queue. This model reflects how many state-of-the-art microsecond-scale systems are built [61, 66, 134, 137, 159].

We define system throughput overhead ($Overhead_{sys}$) as the fraction of CPU cycles that do not contribute towards application goodput. Eq. 3.1 describes the overall overhead on a system with n workers and 1 dispatcher. We separate the overhead based on the types of threads, i.e. $Overhead_w$, $Overhead_d$ denote the per-worker and dispatcher overheads respectively. Since the dispatcher does not run application logic, $Overhead_d = 1$.

To define $Overhead_w$, we consider the CPU cycles wasted during the lifetime of a request with a service time of S CPU cycles, and summarize it in Eq. 3.2. Intuitively, for every request there are some cycles lost during processing (c_{proc}) beyond the application logic. These include overheads of the underlying runtime, such as for logging, and are proportional to the service time. Further, in a system that supports preemptive scheduling, there are also lost cycles associated with each preemption (c_{pre}) that include context switch and inter-thread communication costs. Finally, after the completion of a request, the worker will need to communicate with the dispatcher and wait for the next request, which will incur further wasted cycles (c_{fin}).

We now further break down these costs: c_{proc} is a fixed fraction of the service time (S), and depends on the implementation of the underlying runtime. c_{pre} is a cost paid on every preemption event, so $\lfloor S/q \rfloor$ times for every request, where q is the scheduling quantum; it includes the cost of receiving the preemption notification (c_{notif}), the context-switch cost (c_{switch}), and the cost to wait for the next request (c_{next}), as seen in Eq. 3.3. Finally, c_{fin} consists of the context switch cost and the cost to fetch the next request (shown in Eq. 3.4).

$$Overhead_{sys} = \frac{n \times Overhead_w + Overhead_d}{n+1} \tag{3.1}$$

$$Overhead_w = \frac{c_{proc} + c_{pre} + c_{fin}}{S} \tag{3.2}$$

$$c_{pre} = \left\lfloor \frac{S}{q} \right\rfloor \times (c_{notif} + c_{switch} + c_{next}) \tag{3.3}$$

$$c_{fin} = c_{switch} + c_{next} \tag{3.4}$$

3.2.2 Sources of Throughput Overhead

We now use this model to analyze the overheads in state-of-the-art systems. Later in §3.3, we introduce new mechanisms that address each of these overheads.

Preemptive scheduling (c_{notif} or c_{proc})

Today, there exist two mechanisms for implementing preemption at microsecond-scale—interrupts and code instrumentation—that introduce significant overheads via the components c_{notif} and c_{proc}, respectively. We describe each approach using the corresponding state of the art—Shinjuku [134] and Compiler Interrupts [23], respectively—as canonical examples.

In interrupt-based systems, the dispatcher sends an inter-processor interrupt (IPI) to a worker whenever it has reached the desired scheduling quantum. The benefit is that the preemption is precise; the worker promptly stops processing the current request and moves on. The drawback is the large cost of receiving IPIs (c_{notif}). Since this cost results in an overhead that is inversely proportional to the quantum size q, namely $Overhead \propto \frac{c_{notif}}{q}$ (Eq. 3.3), interrupt-based approaches lead to prohibitive throughput overheads at microsecond timescales. For instance, receiving an IPI in Shinjuku costs $\approx$1200 cycles which results in an $\approx$12% overhead for $q = 5\mu s$, and an $\approx$30%

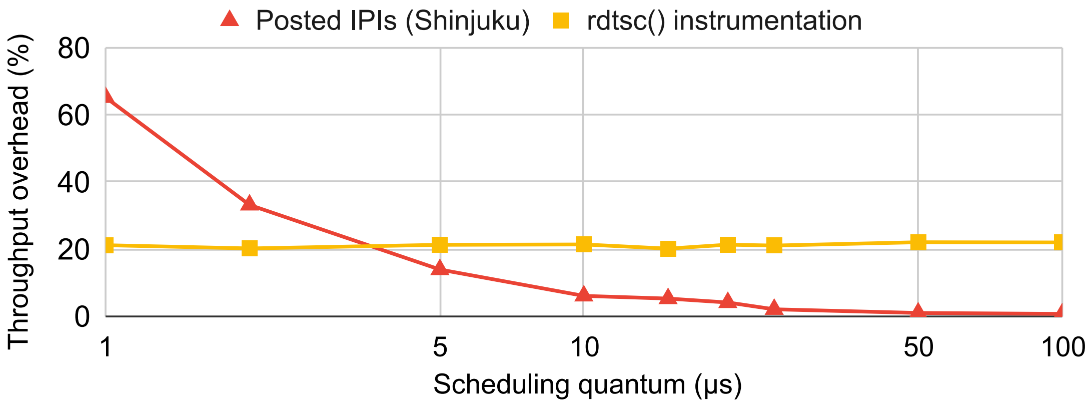


Figure 3.2: Overhead of state-of-the-art preemption mechanisms as a function of the scheduling quantum. This overhead excludes the time required to context switch and receive a new request.

overhead for $q = 2\mu s$, assuming a 2GHz clock. Note, Shinjuku's IPIs rely on non-standard use of virtualization hardware and cannot be deployed in the public cloud. The corresponding overhead for Linux's easily-deployable IPIs is double [31, 134].

Instrumentation-based approaches forgo the dispatcher and rely solely on compile-time instrumentation of the code. The compiler inserts bookkeeping probes (e.g., `rdtsc` calls) at regular intervals in the application code, enabling the worker to track how long it has been executing for and yield the CPU when the quantum has elapsed. This approach offers the benefit of avoiding IPIs, thus eliminating c_{notif}. However, bookkeeping probes are expensive (e.g., calling `rdtsc()` costs $\approx$30 cycles) and so inserting them frequently leads to prohibitively high overheads, while inserting them infrequently leads to poor preemption timeliness. In our model, the cost of bookkeeping is represented by c_{proc}. Since the bookkeeping is typically performed at significantly smaller granularities than microsecond-scale scheduling quanta [23], c_{proc} is a fixed fraction of the service time and leads to a throughput overhead independent of the scheduling quantum.

Fig. 3.2 provides empirical evidence for our model's predictions w.r.t the two preemption mechanisms. We measure the time it takes Shinjuku and Compiler Interrupts to service $1M$ requests, each running for 500µs, while handling preemption notifications with scheduler quanta from 1µs to 100µs. We compare to a baseline where each request runs to completion, without interruption. To isolate the preemption overhead, we run both systems with no-op preemption handlers. As predicted by our model, IPIs in Shinjuku lead to an overhead that grows linearly with decreasing scheduling quanta: 33% at 2µs and 6% at 10µs. On the other hand, the `rdtsc()` probes used by compiler interrupts lead to a uniform $\approx$ 21% overhead across all scheduling quanta, since the probes are inserted approximately every 200 instructions, which is substantially smaller than 1µs.

Synchronous inter-thread communication (c_{next})

Maintaining a single *physical* queue mandates synchronous communication between the dispatcher and worker threads. To ensure optimal load balancing in such systems, each worker

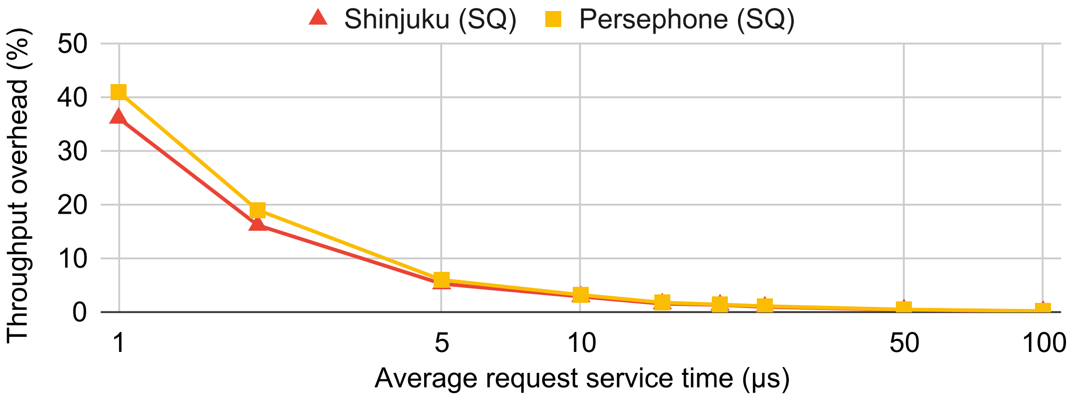


Figure 3.3: Time spent idle by a worker thread awaiting the next request in state-of-the-art Single Queue (SQ) systems.

thread must first finish processing the current request before it can pull the next request from the dispatcher. To avoid concurrency issues due to multiple workers pulling from the dispatcher, state-of-the-art systems [66, 134] implement a single queue as follows: (1) workers set a flag upon finishing a request and then poll a dedicated cache line for a new request; (2) the dispatcher continuously polls the workers' flags and sends a new request as soon as a flag is set.

This synchronous communication directly results in wasted CPU cycles (c_{next}), since workers sit idle until the dispatcher sends them a new request. In particular, c_{next} subsumes at least two cache coherence misses, which add up to $\approx$400 cycles in total [62]. These misses occur when (1) the dispatcher reads the flag previously written by the worker (Read after Write miss) and (2) the dispatcher writes into the worker's request queue that was last read by the worker when processing the previous request (Write after Read miss). Note, 400 cycles provides a lower bound on c_{next} since this assumes that the dispatcher sends a new request to the worker instantly. In practice, the dispatcher may be busy preempting or dispatching requests to any of the other n cores, so in the worst case, the worker thread might have to wait as long as $400 \times n$ cycles. For short requests, this idle time can lead to significant throughput overheads since the component of system overhead induced by c_{next} is inversely proportional to the service time: $Overhead \propto \frac{c_{next}}{S}$.

Fig. 3.3 illustrates the measured median overhead due to c_{next} for Shinjuku and Persephone when running with 8 cores. As predicted by the model, the overhead is inversely proportional to service time. However, the overhead increases slightly faster than $\frac{1}{S}$ because, with shorter request times, it becomes more likely that multiple workers finish while the dispatcher is busy sending a request to another worker.

Dedicated dispatcher ($Overhead_d$)

Since the dedicated dispatcher does not run application logic even when idle, $Overhead_d = 1$.

While dedicating 1 core does not significantly impact throughput when running on a large server, it does have a serious impact for smaller VMs in the cloud. For example, consider a 16-core server with 1 dispatcher and 15 worker threads, where the dispatcher runs at full capacity to feed the 15 workers. When serving the same workload from a 4-vCPU VM in the cloud, the dedicated dispatcher only serves 3 workers (20% of its capacity), and thus ends up being idle 80% of the time. As a result, in this particular deployment, the system as a whole sacrifices $\frac{80}{4\times 100} = 20\%$ of its potential maximum throughput.

To summarize, state-of-the-art microsecond-scale schedulers suffer from three main sources of throughput overhead: preemptive scheduling, synchronous inter-thread communication between the dispatcher and workers to maintain the single queue, and the dedicated dispatcher that runs no application logic. In the next section, we describe how Concord, our proposed microsecond-scale scheduling runtime, addresses each source of overhead.

3.3 Efficient Microsecond-Scale Scheduling

We now describe the design of Concord, an efficient scheduling runtime for microsecond-scale applications.

Concord's design is driven by the insight that careful *approximation* (as opposed to canonical implementation) of optimal scheduling policies enables efficient, low-overhead microsecond-scale *mechanisms* that lead to significant throughput improvements at negligible tail-latency costs. Thus, Concord preserves the tail latency properties of the state of the art, while increasing application throughput by introducing new mechanisms that carefully approximate existing policies.

Concord relies on three key mechanisms to efficiently approximate the optimal single queue and precise preemption policies and mitigate the sources of throughput overhead described in §3.2. First, compiler-enforced cooperation (§3.3.1) approximates precise preemption using asynchronous communication between the dispatcher and worker threads. While this asynchrony leads to slightly imprecise scheduling quanta, it does not significantly impact tail latency. Instead, it enables Concord to reduce the preemption overhead by 4× by minimizing c_{notif} while keeping c_{proc} low. Second, Join-Bounded Shortest Queue (JBSQ) scheduling (§3.3.2) adds bounded core-local queues to approximate a single queue; this enables Concord to reduce cache-coherence stalls in worker threads by $9-13\times$ by nearly eliminating c_{next}. Finally, the Concord-dispatcher (§3.3.3) is work-conserving and steals work from the global single queue when all worker threads are busy. This approximates both the single queue and precise preemption—the dispatcher sends preemption notifications late when busy—but ensures $Overhead_d < 1$ which significantly improves application throughput at low core counts. Fig. 3.4 provides an architectural diagram of Concord that we gradually explain throughout the section.

Concord uses an asymmetric threading model with a dispatcher thread D and worker threads

$W_1, ..., W_n$; in §3.3.1 we describe how this design enables Concord to achieve low-overhead preemption while retaining the flexibility to support arbitrary scheduling policies. Each worker thread is pinned to a CPU core, to ensure maximum locality. This architecture is consistent with §3.2.1 and how state-of-the-art microsecond-scale systems are built today [61, 66, 134, 137, 159, 185].

3.3.1 Compiler-Enforced Cooperation

We now describe how Concord's compiler-enforced cooperation provides an alternative to enforcing latency quanta using IPIs that: (1) enables preemption at lower overhead for microsecond-scale tasks, (2) does not require non-standard use of hardware and can be deployed in the public cloud, and (3) makes it easier to port applications and preempt safely.

In Concord, scheduling decisions are communicated between workers W_i and the dispatcher D via a per-core dedicated cache line L_i, instead of IPIs. The Concord runtime enforces this communication for arbitrary applications using automated compiler instrumentation. The dispatcher monitors how long each request has been executing and writes to L_i when the request has reached the end of its scheduling quantum. Concord's compiler instrumentation ensures application code running on W_i periodically checks cache line L_i for a preemption signal from the dispatcher. When the signal is received, the worker thread writes to L_i indicating to D that preemption has taken place, and yields. Yielding consists of saving the context corresponding to the current request, and then switching to the default worker context, which awaits the next request. The dispatcher re-places the preempted request on the main queue. Thus, Concord *automatically* converts worker threads from being "interrupt-driven CPU drivers" to "poll-mode CPU drivers". This is consistent with how the majority of low-latency systems today eschew interrupts in favor of polling due to the associated overheads [70, 123, 124, 134, 142, 186, 193, 251].

Concord deliberately separates scheduling concerns between D, in charge of signalling the end of a quantum, and W_i, in charge of yielding. D has global visibility of the system, and so it is best positioned to decide when W_i should stop processing a request and which request it should begin processing instead. On the other hand, cooperative yielding allows worker threads to switch between requests within $\approx 100ns$, and avoids expensive preemptive context switches. Delegating the preemption notifications to the dispatcher ensures that Concord can support scheduling algorithms beyond First Come, First Served (FCFS) and Processor Sharing (PS). For instance, Concord can easily be extended to support algorithms such as Shortest Remaining Processing Time [209] or ones that takes locality into account and prioritize scheduling preempted requests back on to the core they were last processed by. Implementing such algorithms in single *logical* queue systems is hard, since they do not have a dispatcher, and thus have no core that possesses visibility of all the requests in the system.

Communicating scheduling decisions via shared cache lines enables Concord to *minimize* c_{notif} (cost of preemption notification), while keeping c_{proc} (instrumentation overhead) low. c_{notif}

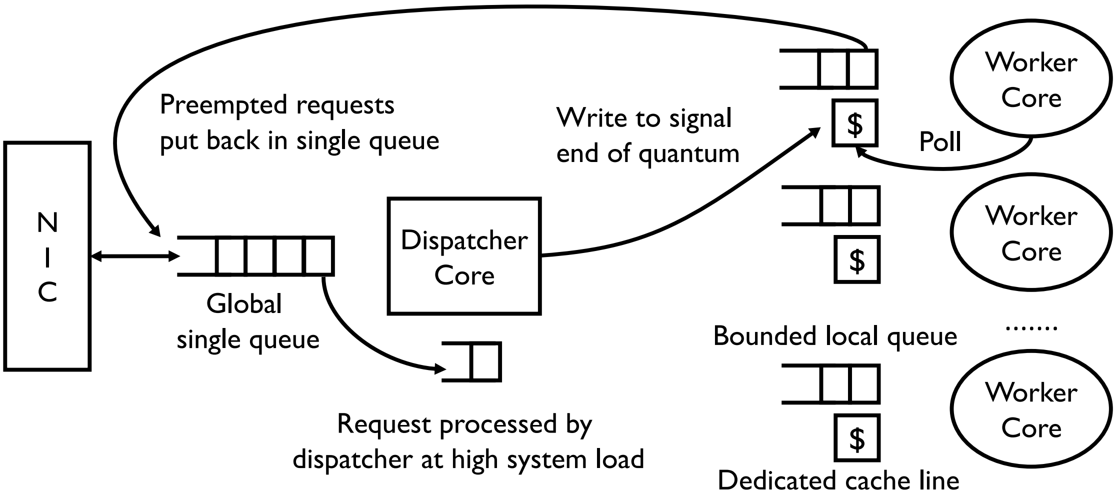


Figure 3.4: The Concord architecture. Compiler-enforced cooperation relies on communicating via a shared cache line, JBSQ(k) employs bounded core-local queues to eliminate coherence stalls, and the dispatcher's steals work at high load.

is minimized since a shared cache line is the fastest way for two cores to communicate in commodity shared-memory processors. This minimization does not significantly increase c_{proc} because, unlike an `rdtsc` call, which always costs $\approx$30 cycles, the cache line L_i is in the L1 cache of worker W_i for all but the final check, so most checks consist of an L1 cache hit plus a compare, i.e., 2 cycles. The final check (after which the request yields), incurs a Read after Write cache-coherence miss since it is the first check after the dispatcher writes to L_i. However, this miss only costs $\approx$150 cycles leading to a c_{notif} that is $\frac{1}{8}^{\text{th}}$ the cost of a Shinjuku IPI (which costs $\approx$1200 cycles), while eschewing reliance on the non-standard use of virtualization hardware.

Fig. 3.5 shows this overhead, for scheduling quanta from 1-100µs. We see that Concord's overhead is near-constant at around 1-1.5%, mainly coming from the instrumentation and not the notification itself, which is consistently 16× cheaper than invoking `rdtsc()`. Concord's overhead is also 12× lower than that of Shinjuku's IPIs at a scheduling quantum of 2µs and 10× lower at a quantum of 5µs. As the quantum increases further, the percentage overhead of an IPI decreases until the two become roughly equal ($\approx$ 0.7%) at around 25µs. Note, 25µs refers to the scheduling quantum and not the service time, so even datacenter applications that have some long requests (e.g., 100µs to 10ms) will benefit from Concord, as long as there are also many short requests (1-10µs) for which we would like to preempt the long-running requests. Many real-world applications have such distributions e.g., search engines, microservices and function-as-a-service (FaaS) frameworks, and in-memory stores or databases such as RocksDB, LevelDB, and Redis that support both point and range queries [13, 40, 52].

Compiler-enforced cooperation approximates precise preemption since workers do not yield instantaneously: the application code must first reach the cache-line check to see the preemption notification. In practice though, we observed that as long as preemption occurs within a "small" interval around the desired quantum, tail latency is not significantly affected. This ensures that

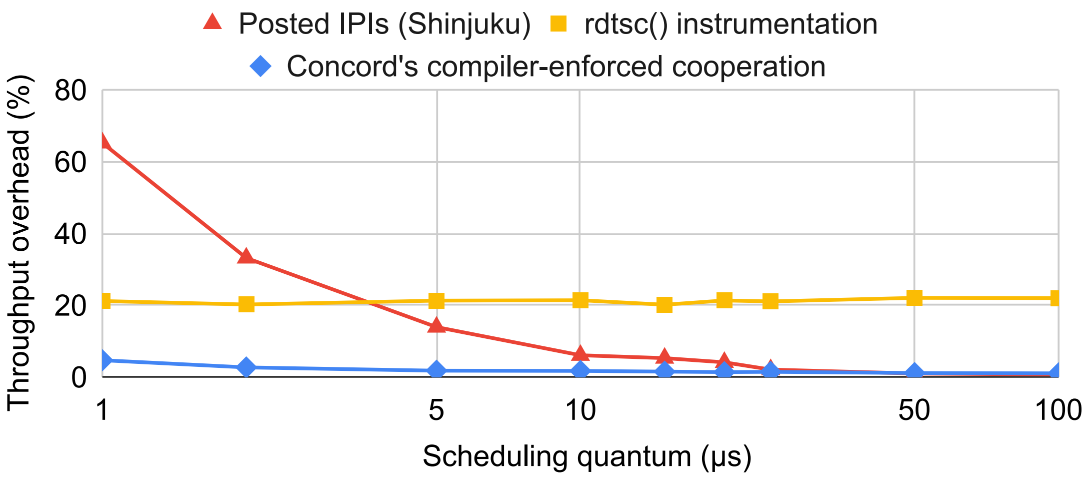


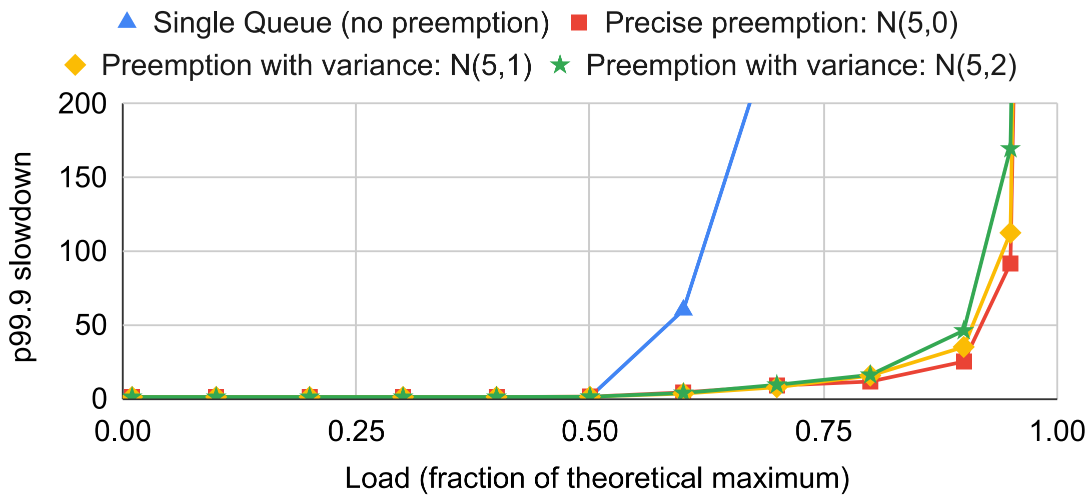
Figure 3.5: Comparing the throughput overhead introduced by Concord's compiler-enforced cooperation to that of state-of-the-art preemption mechanisms. This overhead excludes the time required to context switch and receive a new request, which is identical across all systems.

compiler-enforced cooperation outperforms IPIs by achieving greater throughput while meeting the same tail latency SLO.

Fig. 3.6 illustrates the impact of non-instantaneous preemption using a queueing simulation for two service time distributions from prior work [66,134]. In the first (Bimodal(99.5 : 0.5, 0.5 : 500)), 99.5% of the requests have a 0.5μs service time and 0.5%, 500 μs. In the second (Bimodal(50 : 1, 50 : 100)), 50% of the requests have a 1μs service time and the other 50%, 100 μs. For both distributions, we model Concord's preemption as a one-sided Normal random variable[1] with a mean of 5μs and different standard deviations and compare this non-instantaneous preemption with precise preemption (red line), which is the optimal behavior and no preemption (blue line), which serves as a lower bound. We observe that for small standard deviations, the latency behavior of the system is almost identical to the optimal precise preemption curve, indicating that approximating the preemption quantum does not significantly affect tail latency. In §3.5.4, we show that for a 5μs quantum, Concord's instrumentation keeps the standard deviation within 2μs across 25 benchmarks from standard benchmark suites.

Finally, compiler-enforced cooperation also reduces the overhead of sending the preemption notification at the dedicated dispatcher by 2.5×, thus reducing the amount of work that the dispatcher needs to do. This is because sending an IPI costs approximately 300 cycles on today's hardware, but writing to a shared cache line costs approximately 120 cycles in comparison. This enables the upcoming mechanisms (JBSQ and dispatcher work-conservation) to be more effective since both require the dispatcher to perform additional work.

[1] The distribution is one-sided because we never preempt before the quantum



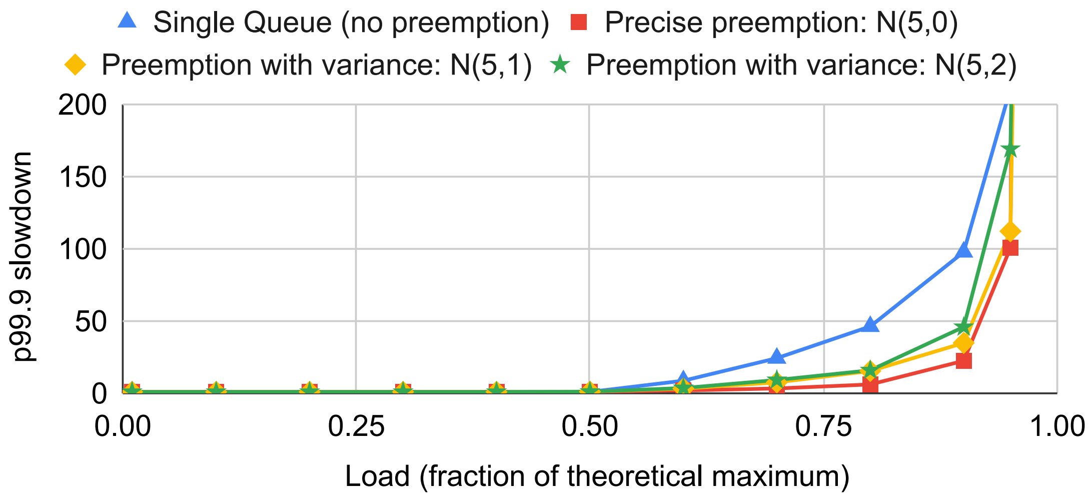


Figure 3.6: The impact of non-instantaneous preemption on 99.9^{th} percentile request slowdown for Bimodal($99.5:0.5, 0.5:500$) (top) and Bimodal($50:1, 50:100$) (bottom) service time distributions. $N(x,y)$ represents a normal variable with mean x and standard-deviation y

Safety-first preemption

Concord takes a safety-first approach to preemption that we believe is particularly suited to microsecond-scale applications. Concord ensures safety by not preempting worker threads when they are either performing external calls that might acquire locks (e.g., system calls) or holding a lock in the application code. While such an approach can (in theory) lead to tail-latency spikes due to long-running critical sections or system calls, in our experience this is rarely the case in practice because such calls are infrequent in microsecond-scale application code.

Concord guarantees that preemption is avoided within external calls by construction, since the compiler has full control over the portions of code it instruments. This has the added benefit of ensuring that widely-used libraries (e.g., `libc`) can be used in Concord without modification. In contrast, such libraries must be modified to ensure safety in systems that rely on IPIs (e.g., Shinjuku) since the worker thread has no control over what code it will be executing when it receives an IPI from the dispatcher.

To avoid preemption while holding application locks developers must modify their code; however, in our experience, this takes negligible effort. For example, to achieve safety in LevelDB we only had to add a total of 4 lines of code that incremented/decremented a counter whenever a mutex was locked/unlocked in the application code. By only preempting if the counter was zero, Concord ensured that it would never preempt a worker thread while it held a lock. On the other hand, the Shinjuku prototype avoids this issue by disabling preemption during entire LevelDB API calls. However, this approach can lead to significant tail-latency spikes since entire LevelDB API calls can run for significantly longer than just their critical sections. It was easy for us to create a microbenchmark where the worker thread in Shinjuku was not preempted until 100µs due to a long-running LevelDB GET API call. For this microbenchmark, Concord improved throughput by 4× in comparison to Shinjuku while meeting the same tail-latency SLO.

3.3.2 Eliminating Cache Coherence Stalls in Worker Threads

To eliminate the overhead due to worker threads being idle, Concord carefully trades the optimal single queue policy in favor of the Join-Bounded-Shortest-Queue policy [142], abbreviated JBSQ(k). JBSQ(k) approximates an ideal, work-conserving single queue by combining a single, central queue with short, bounded per-worker queues, each with a maximum depth of k messages. JBSQ(1) is therefore equivalent to a single queue.

JBSQ enables Concord to forego the purely pull-based single queue and adopt a controlled *push-based* policy: Whenever there is a pending request in the main queue, and one or more per-worker queues have empty slots, the dispatcher pushes the request to the shortest per-worker queue. This ensures that, upon completing a request, worker threads can immediately begin processing a new request from their local queue, thus eliminating the idle time spent waiting for the next request.

To ensure that the per-worker queues do not significantly impair load balancing (and hence tail latency), k must be just large enough to ensure that a worker is never idle during the dispatcher-worker communication. Any larger value of k only hurts tail latency without improving throughput. While the exact communication delay is a complex function of the number of workers and the service time distribution, we found $k = 2$ to be sufficient for service times above 1µs. Approximately, a value of $k = \lceil \frac{c_{next}}{S} \rceil + 1$, where S is the service time, should ensure zero idle time. Prior work [142] has shown that $k = 2$ imposes a negligible tail-latency penalty over the optimal single queue.

Fig. 3.7 compares the throughput overhead due to idling in state-of-the-art systems implementing a single queue and Concord, which uses JBSQ(2). We observe that using JBSQ(2) results in an

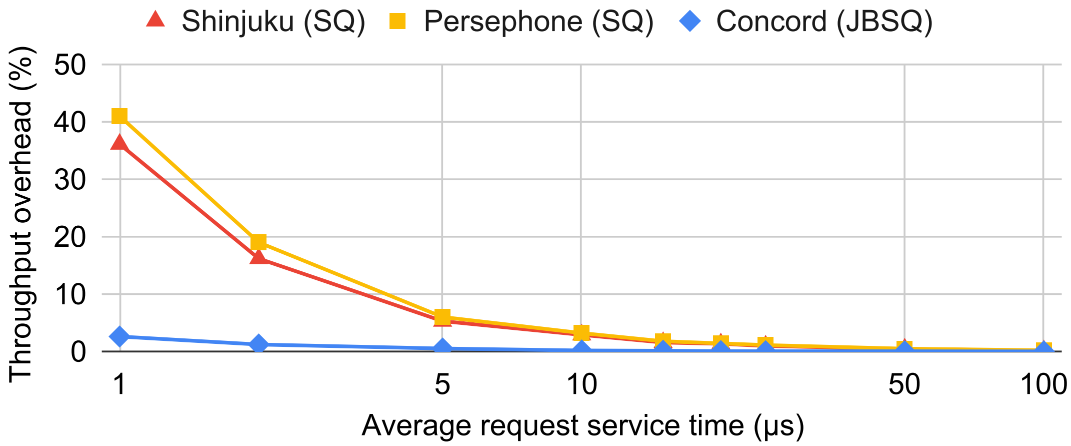


Figure 3.7: Time spent idle by a worker thread awaiting the next request in Single Queue (SQ) and JBSQ systems.

overhead that is 9−13× lower. Of course, JBSQ(2) does not make c_{next} zero. This is because the asynchronous dispatching and processing of requests requires the worker to start a timer denoting the scheduling quantum; in a synchronous single queue, this can be done by the dispatcher.

This work is the first to make the observation that JBSQ(k) is necessary to mask cache-coherence latencies during inter-thread communication. In fact, JBSQ(k) is the accepted policy when the communication delay between the dispatcher and workers approaches the average service time [60,113,117,142,220]. This is because it enables explicit control of the trade-off between tail latency and throughput, through the choice of k, thus making it possible to pick an optimal queue depth. Prior work has already explored the use of JBSQ(k) in scenarios where the dispatcher was located on either a programmable switch [142] or a smartNIC [113, 117].

3.3.3 A Work-Conserving Dispatcher

To reduce $Overhead_d$ (=1 for a dedicated dispatcher) the Concord dispatcher contributes to application goodput while retaining the ability to respond to network and worker events in a timely manner. Whenever the dispatcher notices that all per-worker queues are full, it begins processing user requests, i.e., it runs application logic instead of dispatcher logic for one quantum.

To ensure the dispatcher provides timely responses to network and worker events, Concord employs `rdtsc`-based instrumentation for the dispatcher. This is because there is no external agent to send preemption signals to the dispatcher and so it must self-preempt. The automatically inserted `rdtsc` probes check periodically whether it is time for the dispatcher to switch from application requests to dispatching.

As a result, Concord has two differently instrumented versions of the application code. The expensive, `rdtsc`-based instrumentation is only used for the dispatcher thread, while the cache-line-polling is used on the worker threads. Since all threads are pinned to CPU cores, the second version does not cause I-cache pressure at the workers; these instructions are limited to the private I-cache of the dispatcher.

Having two versions of the code requires a slight modification of the single queue: This is because requests that have started processing on a worker cannot be processed by the dispatcher and vice versa, since the instruction pointers are different due to different instrumentation. Therefore, the dispatcher can only pick up non-started requests from the central queue, and, once it starts processing a request, it is solely responsible for completing that request. So, whenever D is not dispatching and not processing a request, it picks the first non-started request from the central queue. If it needs to preempt itself before completing the request, it saves the context to a dedicated buffer. The next time D is idle, it picks this request up from the buffer and continues processing it.

This approximation does not significantly impact tail latency. We present an intuitive argument now, and demonstrate it empirically in §3.5. First, at low loads it is unlikely that all per-worker queues will be full, hence the dispatcher will never have a request to pick up from the queue. To understand the impact at high loads, assume that the dispatcher is idle for 50% of each time quantum, and the `rdtsc` instrumentation induces 20% overhead. This makes the dispatcher only $50\% - 50 \times 20\% = 40\%$ as effective as a typical worker, causing the request to take 2.5× the usual service time. In practice, this overhead turns out to be far less than the time the request would spend queueing or bouncing around different workers if the dispatcher had not taken it over for processing. Typical tail slowdown targets at high load are $20-50\times$ according to [66, 117], so we believe that the 2.5× is negligible.

To summarize, Concord efficiently approximates the optimal single queue and precise preemption to mitigate the throughput overheads that plague state-of-the-art microsecond-scale schedulers. To do so, Concord leverages three mechanisms—compiler-enforced cooperation, JBSQ(k) scheduling and a work-conserving dispatcher—all of which eschew the non-standard use of hardware and application-level assumptions.

3.4 Concord Prototype

In this section, we describe the key implementation details of our Concord prototype.

3.4.1 API

Concord's API comprises three callbacks:

- `setup()` initializes global application state
- `setup_worker(int core_num)` initializes application state for each worker thread, such as local variables or configuration options

- `response_t* handle_request(request_t*)` processes a single application request and returns a pointer to the response. At any particular point in time, a request is only processed by a single thread, although preemption might cause it to be served by multiple threads over its entire service time.

This simple event-driven API hides all of Concord's underlying complexity from application developers and enables Concord to be easily integrated into existing dataplane OSes; we now describe two integrations.

3.4.2 Concord Runtime

We integrate the Concord runtime into two state-of-the-art microsecond-scale operating systems, Shinjuku and Persephone.

The Concord-shinjuku implementation was straightforward, since Shinjuku's dispatcher already implements a preemptive scheduling policy and there exists a userlevel threading mechanism. We only had to change the preemption signal, add per-core queues and add support for dispatcher work-stealing. Concord-shinjuku only required adding 847 LOC to Shinjuku's initial codebase.

The Concord-persephone implementation required more effort since Persephone operates in a run-to-completion manner. Thus, we had to implement userlevel threading and ported Shinjuku's implementation to Persephone for the same. JBSQ was easier to implement here since Persephone already supports multi-request queues. In total, Concord-persphone adds 2358 LOC to Persephone.

3.4.3 Concord Compiler

We implemented Concord's code instrumentation as two LLVM passes; one each for polling a shared cache line and checking `rdtsc()`, respectively. Both passes use LLVM version 9 and comprise $\approx$ 350 LOC each.

The Concord compiler places probes at the beginning of each function call, before and after any call to un-instrumented code (e.g., syscalls) and at every loop back-edge. Placing probes as such has been shown empirically to be sufficient to yield on all long paths through code [23, 130]. For non-loop code, this translates into a probe being placed approximately once every 200 LLVM IR instructions [23], and so, to avoid prohibitive overheads arising from tight program loops, we unroll each loop body until it has at least 200 LLVM IR instructions. With additional engineering effort, it should be feasible to place probes more infrequently for both loops and non-loop code. We did not to pursue this goal in our work since Concord's instrumentation overhead is already low ($\approx$1% on average).

3.5 Evaluation

We evaluate Concord to answer the following questions:

- How does Concord perform across different service time distributions for which different scheduling policies are optimal? (§3.5.2)
- How does Concord perform for a real, latency-sensitive application? (§3.5.3)
- What is the contribution of each of Concord's proposed mechanisms to its overall performance benefits? (§3.5.4)
- What are the drawbacks of Concord's design? (§3.5.5)
- Do Concord's mechanisms remain relevant as datacenter server hardware evolves to provide increased support for microsecond-scale scheduling? (§3.5.6)

3.5.1 Methodology

Baselines: We focus on blind policies, i.e. policies that do not rely on application-level information, and pick two baselines that represent the state of the art for workloads with high and low service time dispersion, respectively. Shinjuku represents the state of the art for workloads with high service time dispersion since it implements both a single queue and preemptive scheduling. To compare against recent systems [66, 185] that implement only an FCFS single queue for workloads with low dispersion, we configure Persephone to use the C-FCFS policy. We refer to this baseline as "Persephone-FCFS".

For all experiments on our own cluster, we use the implementation that builds on top of Shinjuku. Since Shinjuku is the best-performing baseline in this context, using this implementation enables an apples-to-apples comparison. As detailed in §3.4, the performance differences between the two implementations of Concord are minuscule.

Testbed: We use a testbed set up as per RFC 2544 [229] with two directly connected machines—a server that runs Concord or the baselines and a client that runs a load generator. Both machines are identical Cloudlab [51] *c*6420 nodes with a 32-core (64-thread) Intel Xeon Gold 6142 CPU running at 2.60GHz, with 376 GB of RAM, and an Intel X710 10 Gbps NIC. The average network round trip time between the client and server is 10µs. The server machine runs Ubuntu 18.04 with the 4.4.185 Linux Kernel since this is the version that Shinjuku's kernel module requires. We set up each system as in prior work [66]: Shinjuku uses one hyperthread for the networker and another for the dispatcher, co-located on the same physical core. Persephone runs both its net worker and dispatcher on the same hardware thread. Unless otherwise specified, all systems use 14 worker threads running on dedicated physical cores.

The client's load generator sends requests according to a Poisson process centered at the workloads' mean service time to mimic the bursty behavior of production traffic [13]. Unless specified, all measurements are performed at the client, ensuring end-to-end evaluation of Concord. Each experiment runs for 60 seconds and we discard the first 10% of samples to remove warmup effects.

Workloads: We used one synthetic and one real application to evaluate Concord across several service distributions from both academic and industrial references. The synthetic workload is a server application that spins for the amount of time specified by each request; this application allows us to evaluate Concord across a variety of service time distributions. In §3.5.2 we describe four such distributions, three of which are based on workload A from the YCSB benchmark [52], Meta's USR workload [13] and TPCC running on an in-memory database [66], respectively.

The real application is a server running LevelDB [150], a popular and widely deployed key-value store developed by Google that supports both point queries (put/get requests) and range queries (scans). We evaluated LevelDB on two service time distributions, one from Meta's ZippyDB traces [40] and the other from prior work [66, 134]. Unless otherwise specified, we use two scheduling quanta—5µs and 2µs respectively—for all workloads. Many of our workloads were also used by Shinjuku and Persephone; in all such cases, we were able to replicate their published results. When running on Shinjuku and Persephone, the application code (both synthetic and LevelDB) was not instrumented by the Concord compiler.

Metrics: For each workload, we primarily compare the throughput that the two systems can sustain given a target 99.9$^{\text{th}}$ percentile *Slowdown* — which is the ratio of the total time the request spends at the server to its un-instrumented service time. Using tail *Slowdown* (instead of latency) allows us to evaluate all workloads at a common Service Level Objective (SLO), despite their absolute latencies varying significantly. For all experiments, we set the slowdown SLO at 50× the service time which is consistent with prior work [66, 134].

3.5.2 Evaluating Concord Using Synthetic Microbenchmarks

Here we mimic four service time distributions, two each with high and low dispersion respectively. The first two distributions stress Concord's approximate preemption and its approximate single queue, while the last two only stress its single queue.

Workloads with high dispersion that benefit from preemption: Both high-dispersion workloads follow a bimodal service-time distribution. In the first (Bimodal(50:1, 50:100)), 50% of the requests have a 1µs service time and the other 50%, 100µs. Such a distribution with an equal amount of short and long requests is based on workload A from the YCSB benchmark [52]. In the second (Bimodal(99.5:0.5, 0.5:500)), 99.5% of the requests have a 0.5µs service time and

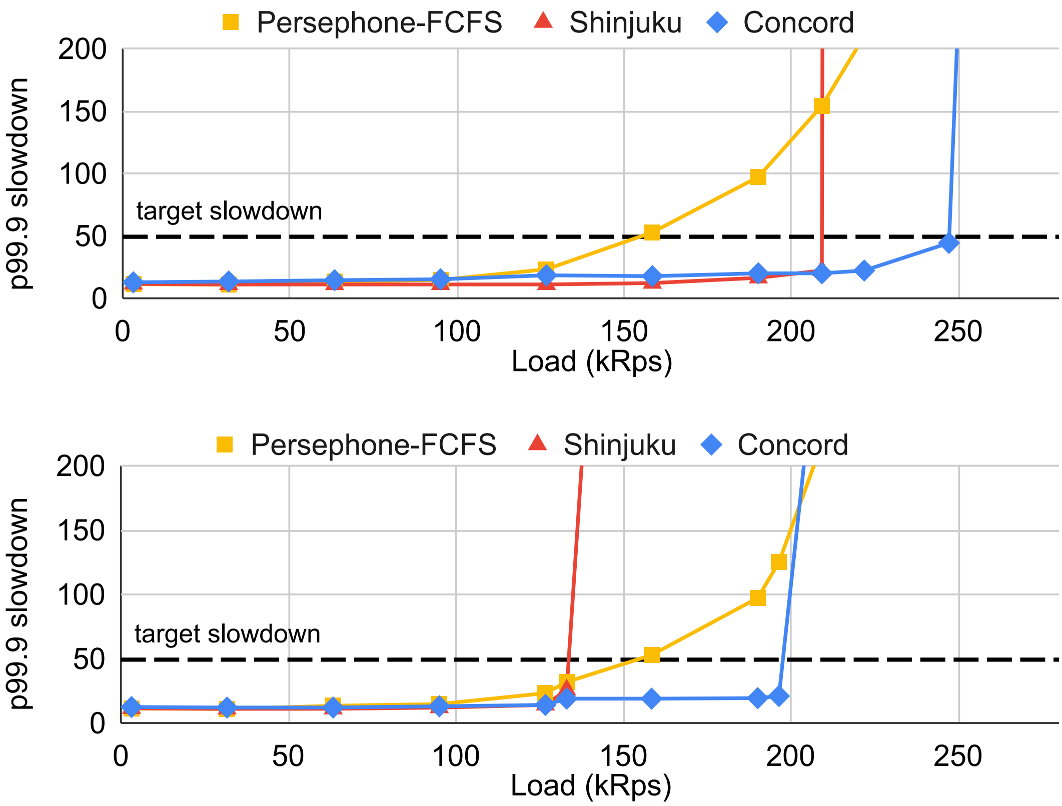


Figure 3.8: 99.9^{th} percentile slowdown vs load for Bimodal(50 : 1, 50 : 100). Scheduling quantum is 5µs (top) and 2µs (bottom).

0.5%, 500µs. This distribution, with a majority of short requests and a few very long requests, is based on Meta's USR workload [13].

Fig. 3.8 and Fig. 3.9 illustrate the results for the two high-dispersion distributions for scheduling quanta of 5µs and 2µs, respectively. For a scheduling quantum of 5µs, Concord can support 18% and 20% greater throughput than Shinjuku for our 99.9^{th} percentile slowdown SLO of 50×. Similarly, for a scheduling quantum of 2µs, Concord supports 45% and 52% greater throughput than Shinjuku. Due to its lack of preemptive scheduling Persephone-FCFS crosses the slowdown SLO much earlier than the other two systems for workloads with high dispersion.

Workloads with low dispersion that do not benefit from preemption: The first low-dispersion workload (Fixed(1)) uses a fixed service time of 1µs for all requests. The second workload (TPCC) is based on the service time distribution of TPCC [227] running on an in-memory database [228] and is taken from prior work [66]. The distribution of request types and service times is as follows: Payment (5.7µs) - 44%, OrderStatus (6µs) - 4%, NewOrder (20µs) - 44%, Delivery (88µs) - 4%, and StockLevel (100µs) - 4%. For Fixed(1), we continue to use scheduling quanta of 5µs and 2µs. For TPCC, we set the quantum to 10µs to avoid unnecessary preemptions, since all requests run for longer than 5µs.

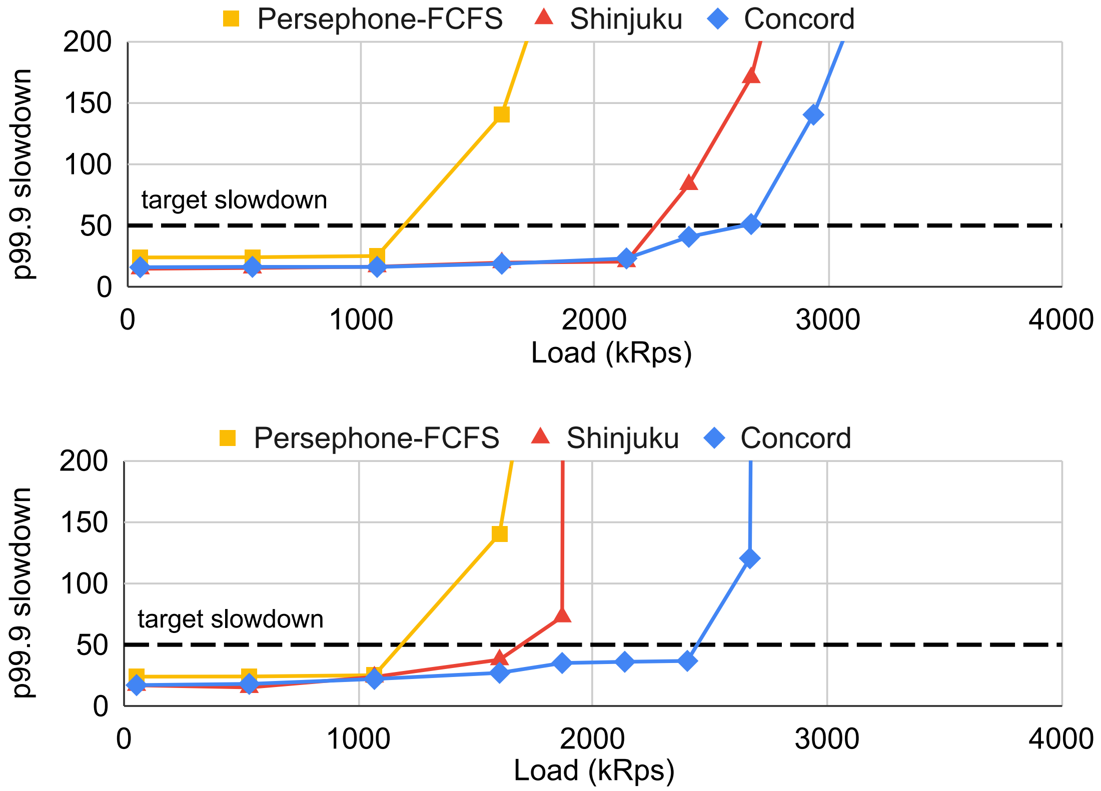


Figure 3.9: 99.9th percentile slowdown vs load for Bimodal($99.5:0.5, 0.5:500$). Scheduling quantum is 5µs (top) and 2µs (bottom).

While such workloads do not benefit from preemption (since there are too few long requests that block the shorter requests), we observe that Concord still performs favorably w.r.t the state-of-the-art; Fig. 3.10 illustrate the results. We see that for the Fixed(1) workload (Fig. 3.10(a)), Concord achieves effectively the same (2% less) throughput than Shinjuku and Persephone. In such situations, the bottleneck is the dispatcher thread—common to all three systems—which cannot deliver requests to workers fast enough. Concord's dispatcher incurs the 2% penalty since it must calculate the "shortest queue" for each incoming request to implement JBSQ(2). For the TPCC workload (shown in Fig. 3.10(b)), which has low dispersion and the dispatcher is not the bottleneck, preemption overheads in Shinjuku and Concord harm throughput compared to Persephone-FCFS, yet Concord still outperforms Shinjuku given its low-overhead preemption mechanism.

3.5.3 Evaluating Concord Using Google's LevelDB

We now compare Concord's performance to Shinjuku's and Persephone's for a LevelDB server that supports both point and range queries. We set up LevelDB in a manner similar to prior work [66, 134]. We populate the database with $15,000$ unique keys and use memory-mapped plain tables to keep all data in memory. In this setup, GET requests take $\approx 600ns$ each, PUT,

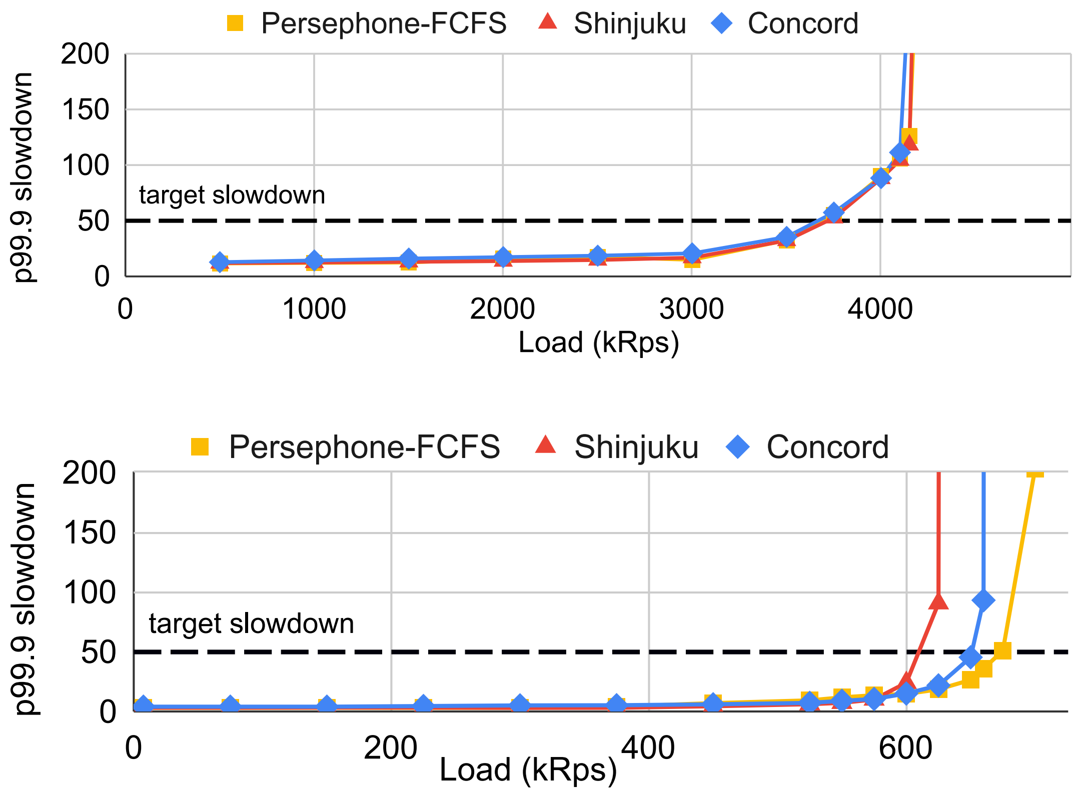


Figure 3.10: 99.9^{th} percentile slowdown vs load for Fixed(1) (top) and TPCC (bottom) service time distributions.

DELETE requests take ≈ 2.3μs each and SCANs take approximately 500μs.

We evaluate Concord's throughput improvements for LevelDB using two request distributions. The first distribution consists of 50% GET requests for a single key and 50% SCAN requests that scan the entire database. This workload strikes a balance between the previous two Bimodal distributions and was used by both Shinjuku and Persephone. The second distribution is based on recently published Meta traces [40] from their ZippyDB service. This workload consists of 78% GETs, 13% PUTs, 6% DELETEs and 3% SCANs. We use scheduling quanta of 5μs and 2μs for the first distribution. We use only a 5μs quantum for the second distribution since all requests in the workload run for longer than 2μs and so a 2μs quantum leads to unnecessary preemptions. Both distributions also allow us to evaluate how Concord performs for real application code with locks since in LevelDB, both PUT and GET requests acquire locks.

Fig. 3.11 illustrates the results for the first distribution. We observe that for our 99.9th percentile slowdown target of 50×, Concord supports 52% greater throughput at a scheduling quantum of 5μs and 83% greater throughput at a scheduling quantum of 2μs. Concord's throughput improvement over prior work is larger for this workload because it has greater dispersion (1000×) than the previous microbenchmarks. At such dispersions—which are common in production workloads [10,41,164]—all three of Concord's mechanisms shine. JBSQ(2) ensures no worker thread is ever idle (minimizing c_{next}), compiler-enforced cooperation ensures that long requests do not suffer prohibitive overheads due to frequent preemption (eliminating c_{notif}) and the incoming requests per second is low enough for the Concord dispatcher to frequently remain idle and thus contribute to application goodput (improving $Overhead_d$). In §3.5.4, we provide a quantitative breakdown for this improvement.

Fig. 3.12 illustrates the results for the second distribution, which is based on Meta's production traces from their ZippyDB service. We see that Concord supports 19% greater throughput than Shinjuku for the target 50× slowdown. This improvement is in line with Concord's results for Bimodal(99.5 : 0.5, 0.5 : 500), shown in Fig. 3.9(a). This is unsurprising, since the two service time distributions are similar.

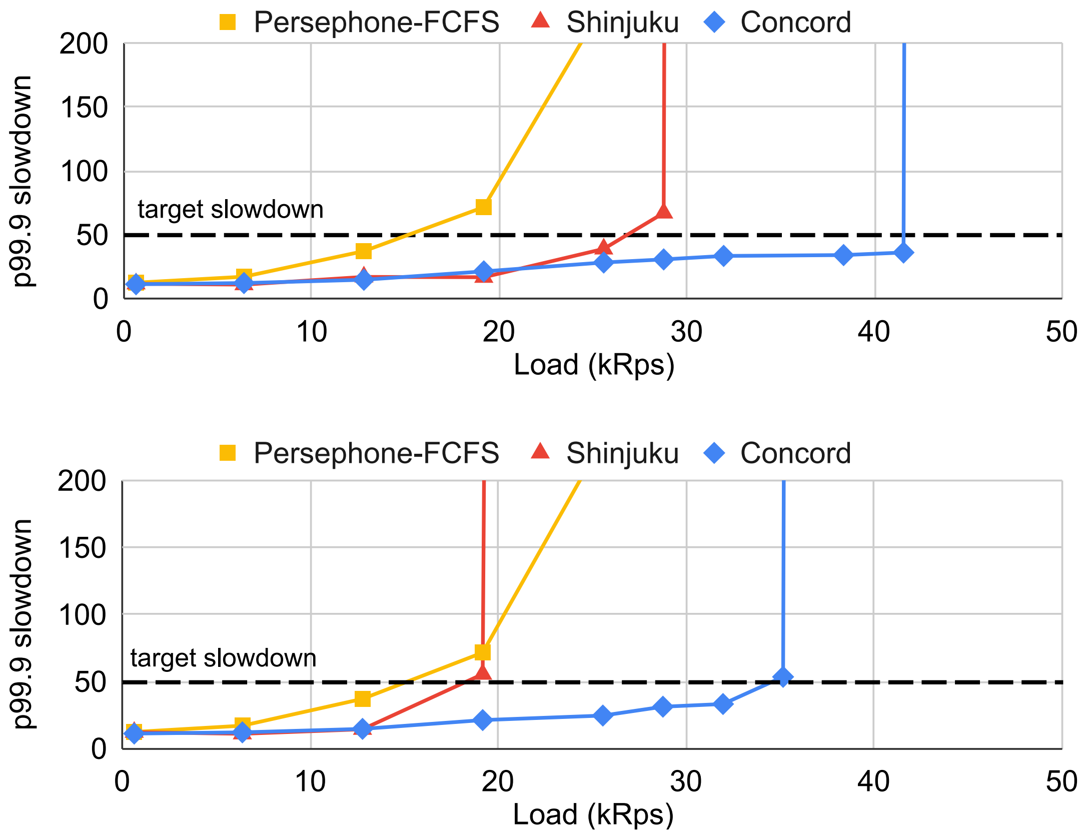


Figure 3.11: 99.9th percentile slowdown vs load for a levelDB server running 50% GETs, 50% SCANs. Scheduling quantum is 5µs (top) and 2µs (bottom).

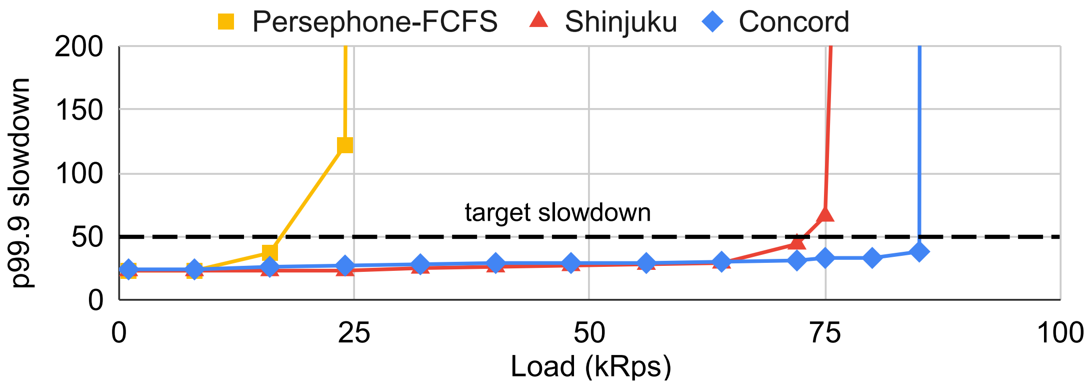


Figure 3.12: 99.9^{th} percentile slowdown vs load for a LevelDB server running a workload based on ZippyDB production traces [40]. Scheduling quantum is 5µs. We do not use a 2µs quantum since all requests run longer than 2µs

3.5.4 Evaluating Concord's Mechanisms Individually

Instrumentation overhead and precision across applications

Since compiler instrumentation rarely produces uniform slowdowns, we evaluated the overhead and timeliness of Concord's instrumentation across 24 benchmarks from the Phoenix [191], Parsec [187] and Splash-2 [217] benchmark suites. We compared Concord's instrumentation overheads to published numbers from prior work [23] that uses `rdtsc()`-based instrumentation. We used their published numbers because we were unable to accurately replicate their results. To obtain optimal overhead numbers, their LLVM pass must be differently configured with 8 parameters for each application and naive configurations lead to significant overheads. They do not publish numbers for preemption timeliness.

Table 3.1 presents the results across the 24 benchmarks. We observe that Concord's average instrumentation overhead is not only low enough to be acceptable (≈1.04% on average) but also 13.1× lower than the state of the art, with the maximum overhead being 5.5× lower.

To evaluate Concord's preemption timeliness, we set a quantum of 5µs and measured the standard deviation from the target quantum for the same set of applications (last column of Table 3.1). Across all benchmarks, we see that the standard deviation is smaller than 2µs and so well within the tolerable imprecision (§3.3.1). Further, the 99th percentile of the achieved scheduling quanta was always within 3 standard deviations ensuring that Concord's imprecise scheduling quanta do not significantly impact tail latency.

Breaking down throughput improvements

We evaluated the contribution of each of Concord's mechanisms to its throughput improvement by measuring the throughput sustained by a system that cumulatively employs Concord's key

Program name	**Benchmark Suite**	**Concord overhead**	**CI overhead**	**Concord std.dev**
`water-nsquared`	Splash-2	-0.3%	3%	0.24µs
`water-spatial`	Splash-2	-0.6%	4%	0.23µs
`ocean-cp`	Splash-2	0.1%	10%	1.8µs
`ocean-ncp`	Splash-2	1%	6%	1.1µs
`volrend`	Splash-2	0.5%	13%	0.47µs
`fmm`	Splash-2	0.4%	-2%	0.11µs
`raytrace`	Splash-2	-0.2%	4%	0.03µs
`radix`	Splash-2	0.9%	4%	0.56µs
`fft`	Splash-2	1.2%	1%	0.63µs
`lu-c`	Splash-2	4.6%	13%	0.63µs
`lu-nc`	Splash-2	-3.7%	23%	0.58µs
`cholesky`	Splash-2	-2.9%	29%	0.86µs
`histogram`	Phoenix	1.6%	20%	0.57µs
`kmeans`	Phoenix	-0.3%	3%	1µs
`pca`	Phoenix	-2.7%	25%	0.06µs
`string_match`	Phoenix	2%	18%	0.86µs
`linear_regression`	Phoenix	6.7%	37%	0.78µs
`word_count`	Phoenix	2.4%	30%	1.11µs
`blackscholes`	Parsec	4%	10%	1.14µs
`fluidanimate`	Parsec	1.3%	2%	0.04µs
`swapoptions`	Parsec	2.2%	24%	0.86µs
`canneal`	Parsec	1.5%	34%	0.02µs
`streamcluster`	Parsec	-2.1%	6%	0.08µs
`dedup`	Parsec	0.4%	4%	1.2µs
Average	-	**1.04%**	**13.7%**	**0.29µs**
Maximum	-	**6.7%**	**37%**	**1.8µs**

Table 3.1: Overhead and timeliness of Concord's instrumentation compared to Compiler-Interrupts (CI) [23]. The baseline (0% overhead) corresponds to un-instrumented code. Concord's overhead is often negative due to its loop unrolling.

mechanisms for a LevelDB workload consisting of 50% GETs and 50% SCANs.

Fig. 3.13 illustrates the results. We observe that in comparison to the ≈19kRps sustained by Shinjuku at the target 50× slowdown, systems that cumulatively employ compiler-enforced cooperation, JBSQ(2) scheduling and a work-conserving dispatcher sustain a throughput of ≈22.5 kRps, ≈32 kRps, and ≈35 kRps, respectively.

We now provide an intuitive argument for these improvements. First, the 3.5kRps improvement due to cooperation can be seen as nearly eliminating the 20% cost of interrupt-based preemptions $(3.5 \approx 0.2 * 19)$. Second, JBSQ(2) eliminates $\approx 400ns$ of idle time per request; this time is used to effectively double the number of GET requests processed which leads to an additional 9.5 kRps (0.5×19) in throughput. Finally, since the absolute load (in kRps) is $\approx 100\times$ lower than the maximum throughput the dispatcher can sustain (Fig. 3.10), the dispatcher spends most of its time idle, allowing it to also contribute to application throughput.

In a nutshell, the absolute service latencies and the large dispersion prevalent in this LevelDB workload (which is very similar to key-value store workloads from prior work [66, 134]) provides the perfect setting for all of Concord's mechanisms to shine.

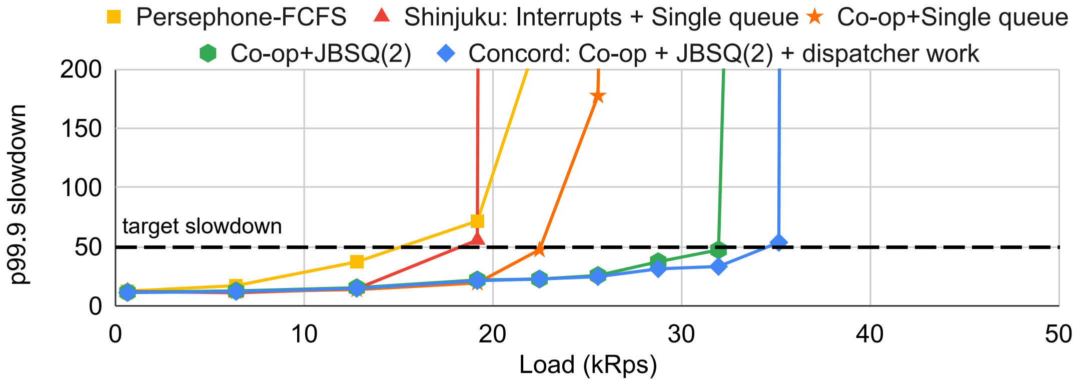


Figure 3.13: Contribution of each Concord mechanism towards throughput improvement for the LevelDB server in Fig. 3.11(b)

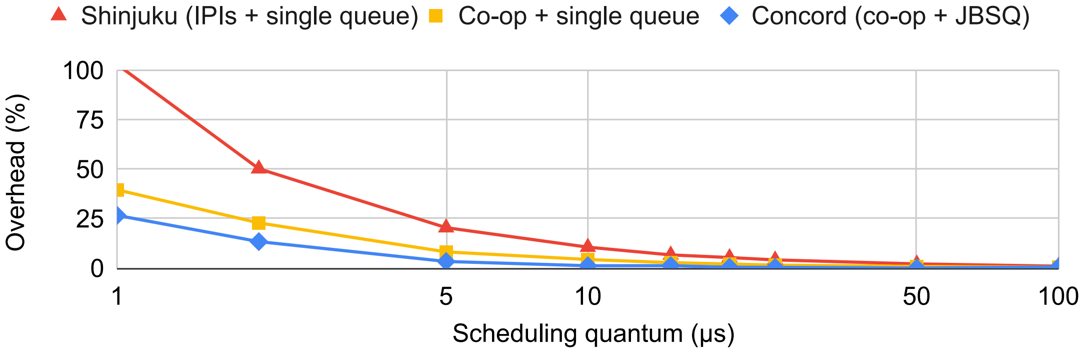


Figure 3.14: Preemption overhead across scheduling quanta.

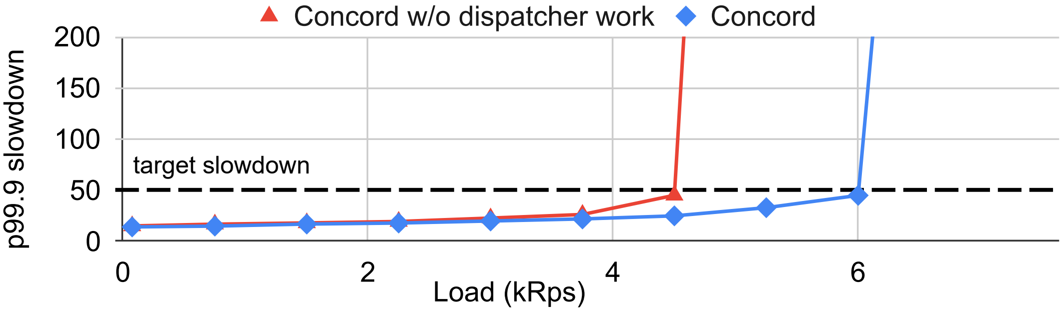


Figure 3.15: Application throughput for dedicated dispatcher vs. Concord dispatcher in a 4-core configuration.

Reduction in preemption overhead

Next, we demonstrate how Concord reduces the throughput overhead of preemptive scheduling, and break down the contributing factors. To do so, we measure the time it takes to service $1M$ requests, each running for 500µs, while handling preemption notifications and yielding. Since the preemption overhead (Eq. 3.3) includes both the cost of the preemption notification (addressed by compiler-enforced cooperation) and the time spent waiting for the next request to arrive (addressed by JBSQ(2) scheduling), we break down the contribution of each mechanism by evaluating the overhead of systems that cumulatively use the above two mechanisms. We use the state-of-the-art interrupt-based approach (Shinjuku) as a baseline and perform this experiment for scheduling quanta ranging from 1-100µs.

Fig. 3.14 illustrates the results. We observe that Concord reduces the overhead of preemptive scheduling by 4× in comparison to Shinjuku. In this setting (unlike Fig. 3.13), we observe that compiler-enforced cooperation is the mechanism that contributes the most towards Concord's improvements. This is because unlike the workload in Fig. 3.13, every request must be preempted, and so reductions in the cost of the preemption notification dominate.

Does the dispatcher do useful work?

Finally, we demonstrate the benefit of running application logic on the dispatcher thread in resource-constrained environments (e.g., smaller VMs in the public cloud).

Fig. 3.15 illustrates our results. We ran the same LevelDB workload on 4 cores—1 dispatcher, 1 networker, two workers—to simulate smaller VMs in the public cloud. In such situations, particularly with the low incoming load (in absolute kRps), the dispatcher is almost entirely idle. Hence, running application logic on the dispatcher thread improves application throughput by 33%.

3.5.5 The Drawback of Approximate Scheduling

While approximating optimal scheduling enables Concord to sustain higher application throughput for the same tail latency/slowdown SLO, it comes with the drawback of slightly increasing tail latency (and hence slowdown) at lower loads.

Fig. 3.16 illustrates this increased slowdown for the workload used in Fig. 3.8 (Bimodal(50 :1, 50 : 100)); we observed similar results across all workloads. We observe that Concord increases the 99.9th percentile slowdown by ≈3 in comparison to Shinjuku at lower loads. This increase in tail slowdown occurs when the Concord dispatcher steals requests during occasional bursts even at low loads. Since these requests cannot be migrated to worker cores once the dispatcher has started processing them due to the different code instrumentation (§3.3.3), they run slower than ones processed by worker cores, which leads to this additional slowdown. That said, we

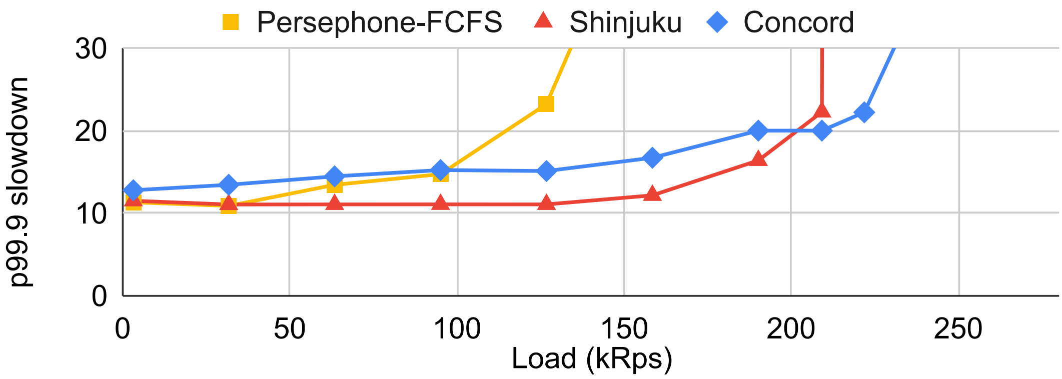


Figure 3.16: Zoomed-in version of Fig. 3.8(a) to show Concord's increased slowdown at low loads. We observed similar increases across all other workloads.

believe that this increase in slowdown is acceptable since it is much smaller than typical SLOs, which are usually $10-50\times$ the service time to account for queueing and network round trip times. Users unwilling to tolerate even this slight increase in slowdown at low loads can disable the Concord dispatcher's work-stealing mechanism and retain the throughput benefits of Concord's compiler-enforced cooperation and JBSQ(2) scheduling.

3.5.6 Is Concord Future-Proof?

Finally, we evaluate whether Concord's mechanisms will remain useful as datacenter server hardware evolves and provides increasing support for microsecond-scale scheduling.

To do so, we compare the throughput overhead of Concord's preemption mechanism (compiler-enforced cooperation) with that of user-space IPIs (UIPIs), a feature recently introduced by Intel on their new Sapphire Rapid servers. UIPIs reduce the overhead of traditional IPIs by allowing application threads to directly send each other interrupts while bypassing the kernel [245]. We set up our experiment just like the one in Fig. 3.5; we measure the time it takes to service $1M$ requests, each running for 500µs and isolate the overhead of the preemption mechanism by running the application with no-op preemption handlers. We use a 192-core Intel Xeon Platinum 8648 Sapphire Rapid server to compare the mechanisms.

Fig. 3.17 demonstrates the results. We observe that Concord's compiler-enforced cooperation outperforms even the most recent hardware support and imposes a throughput overhead that is $\approx 2\times$ lower. This is unsurprising, since no matter how fast interrupts get, they will always be slower than Concord's reads and writes to a shared cache line which will remain the fastest way for two cores to communicate on shared memory hardware. Note, the absolute value of Concord's overhead is slightly higher on this machine as compared to the machines we used in our evaluation and for Fig. 3.5. This is because of the large number of cores, which makes cache coherence misses approximately 1.5× more expensive. That said, the *relative* overhead of Concord with respect to UIPIs will remain the same even on Sapphire Rapid machines with fewer

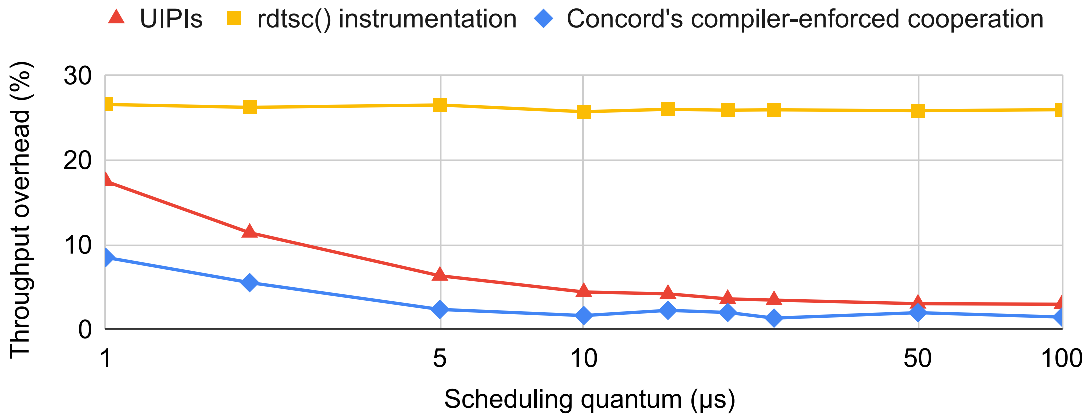


Figure 3.17: Comparing the overhead of Concord's compiler-enforced cooperation to Intel's new user-space IPIs.

cores, since sending an interrupt also requires writing to memory-mapped registers and thus relies on the same cache coherence mechanisms as Concord's compiler-enforced cooperation.

3.5.7 Conclusions Drawn

Recall that the goal of this chapter was to build a microsecond-scale scheduler that preserves the tail latency properties of the state of the art while improving application throughput and eschewing reliance on custom hardware or application-level assumptions (§3.1).

Our evaluation of Concord demonstrates that it accomplishes this goal across a wide range of service time distributions and for both synthetic microbenchmarks and real applications. While Concord is particularly effective—18-83% higher throughput than the state of the art—for service time distributions that benefit from preemptive scheduling (Figures 3.8, 3.9, 3.11, 3.12) it continues to perform favorably even for service times that only require single queue scheduling (Fig. 3.10). Finally, each of Concord's mechanisms contributes to its overall throughput improvements (Figures 3.13, 3.14) and will remain relevant even as datacenter servers provide increasing hardware support for microsecond-scale scheduling (Fig. 3.17).

3.6 Discussion

Limitations

Concord has two main limitations:

First, it requires the application source code to be available and written in a compiled language with an LLVM backend. We believe that access to source code should not be an issue for developers deploying Concord on bare metal, or for tenants deploying their low-latency systems on VMs in the public cloud. That said, the current Concord prototype is ill-suited to being a runtime provided by public cloud providers for their tenants since this would require that all tenants share their code with the cloud provider. We plan to explore replacing Concord's source code instrumentation with approaches that directly instrument the binary to overcome this limitation.

Second, the current Concord prototype is restricted to single-dispatcher systems. This will not be a limitation for low CPU count, e.g. small VMs, but the single dispatcher can become a bottleneck as the number of CPUs increases and the service time (as well as service time variability) decreases. In such cases, replication, i.e. creating multiple single-dispatcher instances that feed disjoint sets of cores, or trading off throughput for tail-latency, e.g. using batching, can help improve scalability [134].

How Concord extends to work-stealing systems

Concord's compiler-enforced cooperation and work conserving dispatcher can enable low-overhead preemption even in systems that implement a single logical queue such as Shenango [185] and Caladan [91]. This is because compiler-enforced cooperation only requires the dispatcher to monitor the elapsed time and does not require it to maintain the single queue. Such a system would also overcome the throughput bottleneck of a single dispatcher.

Incorporating the above mechanisms will require a dedicated hyperthread (referred to as the "scheduler" henceforth) that only monitors whether a worker thread has been processing a request for longer than the scheduling quantum. Some single-logical-queue systems (e.g., Caladan [91]) already have such a scheduler thread. When the quantum has elapsed, the scheduler writes to the cache line, and the worker—instrumented to poll for it—stops processing the current request. Since the dispatch of a request is not synchronous with when a worker begins processing it, the worker must start the timer at the beginning of the quantum, but this is already the case with Concord's asynchronous dispatch for JBSQ (§3.3.2). Finally, the scheduler can steal requests safely using `rdtsc()` instrumentation (§3.3.3) since it is likely to be idle for extended periods.

Broader use of compiler-enforced cooperation

We believe Concord's compiler-enforced cooperation mechanism can be used as a replacement for IPIs in many settings beyond scheduling. For instance, any periodic event, such as garbage collection and timer management or Unix signals and global synchronization mechanisms, implemented through `membarrier()` on Linux or `FlushProcessWriteBuffers` on Windows, can eschew IPIs in favor of compiler-enforced preemption. Compiler-enforced preemption can also be used in deployments where IPIs are not available or untrusted. This is the case for confidential VMs [9, 119] in which the hypervisor is considered potentially malicious and can inject virtual interrupts at any point in time.

3.7 Related Work

Having already discussed the closest related work—Shinjuku, Persephone, Compiler Interrupts—in detail, we do not do so again here.

Microsecond-scale schedulers deployed by major cloud providers (e.g., ghOSt from Google [114]) typically have lower raw performance than academic ones (e.g., ghOSt aims to be within 5% of Shinjuku's maximum sustainable throughput). However this is because they prioritize requirements that academic schedulers (including Concord) do not; for instance, the ability to simultaneously support multiple tenants with different scheduling policies and the ability to provide fault isolation. In this work, we focused on extracting maximum raw performance from a single application with Concord. We plan to pursue extending Concord to support the above constraints as immediate future work.

The key ideas underlying Concord's compiler-enforced cooperation, namely cooperative scheduling and inter-core communication using dedicated cache lines, are well known. For instance, cooperative scheduling and user-level threading go back to the seminal paper on Scheduler Activations [11] and are widely used in different contexts such as language runtimes, e.g. goroutines [104], and modern thread library implementation for the datacenter, e.g. Arachne [196]. Similarly, using dedicated cache lines instead of interrupts or barriers to enable low-overhead core-to-core communication is widely used for high-performance computing [7, 97]. That said, the key difference between such prior work and Concord is that the former relies on the programmer to correctly insert both yield points and reads/writes to cache lines, while Concord does so automatically using a compiler pass.

While programming language approaches have been used extensively in the context of memory isolation [116, 186], splitting CPU time using such approaches is more challenging. Lilt [231] introduces a new language to statically enforce timing policies and the Erlang scheduler [81] depends on the underlying language virtual machine implementation to count the number of executed instructions and preempt Erlang processes in a timely manner. Finally, Libringer [31] introduces the abstraction of a preemptible function but depends on Unix signals to implement it, thus incurring high overheads for µs-scale tasks.

3.8 Key Takeways

Designing and implementing Concord made us acutely aware of the two main challenges that developers face in reasoning about the latency behavior of systems code: (1) The differing latency behavior of different execution paths through the application logic (e.g., processing of different request types) which necessitates preemptive scheduling, and (2) Micro-architectural (specifically CPU cache) effects that lead to unexpected latency overheads and necessitate JBSQ(k) scheduling.

In Concord, we approached these challenges using the traditional systems approach: measure, analyze, optimize, and repeat. While this approach eventually enabled us to correctly add cache line probes at all important points in the code to ensure preemption and realize that a significant portion of the overhead of preemption came from cache coherence induced stalls, it was a painstaking effort that will likely need to be repeated for a different system.

This experience motivated us to seek a more precise representation that enables developers to reason precisely about both the latency and micro-architectural behavior of systems code. We describe our proposed representation in the next chapter.

Latency Interfaces for Systems Code Part III

4 Latency Interfaces as Simple, Executable Programs

In this chapter, we present our proposal for latency interfaces as programs. We first describe the design goals and target audience for latency interfaces (§4.1), then define (§4.2) and illustrate (§4.3) our programmatic representation before explaining our design rationale (§4.4)

4.1 Design Goals and Target Audience

We envision latency interfaces providing succinct descriptions of a system's externally visible *latency* behavior, just like semantic interfaces (e.g., abstract classes, specifications, header files, documentation) describe a program's externally visible functionality [147].

We design latency interfaces for two categories of audience for any system. The *developers* write the code for the system and are familiar with its low-level implementation details, but not necessarily with all possible latency behaviors it can exhibit. The *operators* did not write the code but instead seek to use/deploy/build on top of the system in their respective environments. They are unfamiliar with and do not necessarily want to understand its low-level details. Further, unlike the developers who care about the system's latency in all settings, they care primarily about its latency in their specific use-case/deployment. These categories can vary from system to system—the developer of an application A might themselves be building upon a network stack B, making them an operator for that stack.

Recall that to be useful to developers and operators, a latency interface must provide a balance between two, typically conflicting properties: (1) *Accuracy*, i.e., the ability to summarize latency *completely* (for every possible input and runtime environment) and *precisely* (with a small error). (2) *Readability*, i.e., being *smaller* than the code and as *abstract* as possible—summarize latency in terms of primitives appropriate for a semantic interface of the system, and reveal implementation details only when necessary. Accuracy and readability are typically in conflict because improving the accuracy of an interface typically involves adding more detail, which makes it harder to read.

Summarizing latency in an accurate yet readable interface is more challenging than doing the same for semantics because systems typically employ functional modularity but close to no latency modularity, i.e., they expose a greater variety of latency behaviors than semantic ones. For instance, a `mov (%ebx),%eax` instruction works the same way on all x86 machines no matter what but when it comes to latency, the modularity is much weaker: the time to execute `mov (%ebx),%eax` can vary by orders of magnitude depending on several factors such as the micro-architectural specifics of the machine and other processes executing on the same machine. As a result, a system's latency can depend significantly on its deployment environment (e.g., the hardware it runs on). So the design of latency interfaces should make it easy to quantify the impact of a particular environment on latency, enabling developers and operators to accurately understand latency across different deployments.

4.2 Definition

The *latency interface* of a program P with procedures $p_1, p_2, ...$ is a program $L_P = \{p'_1, p'_2, ...\}$. A procedure $p'_i \in L_P$ takes the same inputs as the corresponding $p_i \in P$ and returns the latency of executing p_i. The latency can be reported in computation steps, in `x86` instructions, in `x86` memory operations, in cycles of a particular processor, etc. We call all these *metrics* for latency.

Every latency interface L_P has a *resolution* r that represents the smallest difference in latency that L_P can specify: if $\mathscr{P}(p_i(I))$ is p_i's latency given input I, then $|p'_i(I) - \mathscr{P}(p_i(I))| < r, \ \ \forall p_i, I$.

A latency interface can be for the "general case" or specific to a deployment.

In a *general-case latency interface*, the procedures p'_i return latency not as concrete numbers but as a function of random variables that we call latency-critical variables (LCVs). LCVs capture how the environment that p_i is deployed in impacts its latency for processing the current input(s). We define p_i's deployment environment by the configuration parameters it reads at startup, a representative workload, and the specific hardware it runs on. LCVs ensure that the interface can describe p_i's latency in full generality, i.e., for arbitrary deployment environments.

An LCV is not always an explicit variable in p_i's implementation, rather it can be an implicit "ghost" variable [87, 96]. For instance, for a p_i that uses a hash map to implement a network flow table, an LCV could be the "number of collisions" encountered while looking up the input flow—this ghost variable allows latency to be expressed as a function of, among other things, the number of collisions. While LCVs appear in the general-case latency interface as uninterpreted functions, they are a deterministic function of p_i's environment. So, given a specific ⟨configuration, workload, hardware⟩ tuple, one can correctly compute the corresponding LCV distribution.

A *deployment-specific latency interface* is simpler than the general-case one and does not contain LCVs. Instead, procedure p'_i returns latency as a statistic (e.g., median, max, 99th percentile), computed for a given joint probability distribution of the LCVs that describes P's environment for a particular deployment. Said differently, deployment-specific interfaces represent partial evaluations of the general-case interface and are derived by instantiating the LCVs in the general-case interface with deployment-specific distributions.

4.3 Example

We now illustrate our programmatic latency interfaces using an example implementation of a MAC learning bridge (shown in Fig. 4.1). Our example bridge uses a fast MAC table, implemented in hardware, and a slow software-based table, based on a cuckoo hash table.

```
void bridge(pkt* p, time_t now) {
  expire_stale_ports(now);
  if (invalid_hdr(p)) {
      DROP(p);
      return;
  }
  /* Learning source MAC addr */
  if(!slow_MACtable_get(p->src_mac, &p->port))
     slow_MACtable_put(p->src_mac,&p->port);
  else
     slow_MACtable_update(p->src_mac, now);
  /* Forwarding based on dest MAC addr */
  if (fast_MACtable_get(p->dst_mac,&out_port))
     FORWARD(p,out_port);
  else if (slow_MACtable_get(p->dst_mac,&out_port))
     FORWARD(p,out_port);
  else
     BROADCAST(p,p->port);
}
```

Figure 4.1: Example implementation of a MAC learning bridge

Table 4.1 shows the latency cost of this implementation's procedures in terms of executed lines of pseudocode (LOP), a latency metric we use for illustration only. Two have non-constant latency: expiring learned ports is linear in the number of stale ports, and doing a `put()` in the cuckoo hash table depends on the number of keys that must be evicted and whether rehashing is necessary. For now, we assume these costs, we elaborate on how they are obtained in Chapter 5.

Operation	Performance [LOP]
expire_stale_ports()	40 + 60× n_stale
invalid_hdr()	5
DROP	1
FORWARD	60
BROADCAST	200
fast_MACtable_get()	10
slow_MACtable_get()	50
slow_MACtable_update()	70
expire_stale_ports()	40 + 60 × n_stale
slow_MACtable_put()	110 + 80 × n_evicted + 120 × occ × rehashing

Table 4.1: General-case performance of procedures called by the code in Fig. 4.1.

Figures 4.2, 4.3 illustrate the general-case and deployment-specific latency interfaces, respectively for this bridge implementation. Since the implementation exposes a single procedure, both interfaces also have only a single procedure. Both interfaces have a resolution $r = 50$ LOP and so they only distinguish between execution paths that can lead to latency variability > 50 LOP. For example, they do not differentiate between successful lookups in the fast or slow MAC tables.

The general-case interface (Fig. 4.2) describes latency as a function of 4 LCVs: number of stale flows (`n_stale`), hash-table occupancy (`occ`), number of hash-table evictions triggered by the current input (`n_evictions`), and whether rehashing is needed (`rehashing=1` if yes, 0 otherwise). Since the latency metric LOP is independent of the underlying hardware, all 4 LCVs are uniquely determined by the bridge's implementation. If the bridge stored the MAC table using a binary tree instead of a cuckoo hash table, the interface would describe latency using different LCVs (e.g., `tree_depth` instead of `rehashing`).

```
def latency_bridge_gc(p,now):
  # Metric: LOP, Resolution: 50
  # NF state: slow_MACtable, fast_MACtable

  if invalid_hdr(p):
    return 46 + 60* n_stale
  if fast_MACtable_get(p->dst_mac) or slow_MACtable_get(p->dst_mac):
    return 280 + 60* n_stale + 80* n_evictions + (120* occ) * rehashing
  else
    return 445 + 60* n_stale + 80* n_evictions + (120* occ) * rehashing
```

Figure 4.2: General case latency interface for MAC learning bridge

The deployment-specific interface (Fig. 4.3) returns the median latency for a deployment where the expected workload is such that 50% of input packets encounter no hash collisions and expire ≤ 1 stale ports. The interface produces concrete numbers corresponding to this deployment-specific LCV distribution. Note, the deployment-specific interface does not restrict the inputs (e.g., the types of packets), it only instantiates the LCVs.

This latency interface captures all the latency behaviors of the bridge that are externally visible at resolution r=50. It is accurate, in that it correctly predicts latency (at the given resolution) for every possible input. It is smaller and simpler than the implementation: each procedure considers only three operations (invalid header check, fast table lookup, and slow table lookup), since these are the only ones that affect latency at r=50. However, unlike the general-case interface, the deployment-specific interface makes assumptions about the expected workload.

```
def latency_bridge_ds(p,now):
  # Metric: LOP, Resolution: 50
  # Statistic: 50th percentile
  # NF state: slow_MACtable, fast_MACtable

  if invalid_hdr(p):
    return 106 #(46+60)
  if fast_MACtable_get(p->dst_mac) or slow_MACtable_get(p->dst_mac):
    return 340 #(280+60)
  else
    return 505 #(445+60)
```

Figure 4.3: Example deployment-specific latency interface for MAC learning bridge

4.4 Design rationale

Why represent the interface as a program that takes the same input(s)? We chose such a representation for three reasons: (1) both developers and operators are intuitively familiar with code, allowing them to quickly eyeball such interfaces and gain visibility into the latency behavior of the program in question without having to run it, (2) accepting the same input(s) allows the interface to describe performance corresponding to each developer or operator's expected workload, something that upper/lower bounds or today's SLOs cannot do, and (3) programs are executable, and enable empirical reasoning about environmental factors that are known to be hard to reason about analytically (e.g., the precise cycles per instruction (CPI) or *effective* last level cache (LLC) miss latency incurred by the program on arbitrary hardware).

Resolution: Often, developers and operators do not care about certain latency differences, either because they do not affect their latency targets, or because they are masked by the environment. For example, developers building second-scale applications may not care about µs-scale variability in the networking stack, while those building µs-scale ones typically do.

The latency resolution enables the developer/operator reading the interface to choose between multiple levels of abstraction (trading off accuracy for improved readability) in a controlled manner. A latency interface at a specified resolution only differentiates between input classes whose latency differs by more than the resolution—implementation details that cause variability relevant to the specific developer/operator are abstracted away. In our bridge example, a latency interface with a resolution of 1 LOP must report the latency of each forwarding behavior separately; an interface with resolution >= 45 LOP can abstract away the difference between a fast and slow lookup, and an interface with resolution >= 115 LOP can abstract away the difference between a successful and unsuccessful lookup.

We envision developers/operators picking their respective resolutions based on the latency variability they are willing to tolerate in their deployment scenarios. In §5.2, we show how our tool (LINX) goes a step further for those unsure of the "right" resolution, by identifying a minimal set of resolution thresholds that yield all the possible different latency interfaces. This is possible since the latency interface can only elide each implementation detail at a distinct resolution threshold, which results in it not changing between two such thresholds. In our bridge example, {1, 20, 45, 115, 210} is such a minimal set of resolution thresholds, i.e., other resolutions don't yield different interfaces (e.g., the interface at r = 50 is identical to that at r = 46). By identifying these resolution thresholds, LINX enables developers and operators to easily pick the resolution (and corresponding interface) that achieves the desired trade-off between accuracy and simplicity.

General-case vs Deployment-specific interfaces: We chose to have separate general-case and deployment-specific interfaces to provide a different balance between accuracy and simplicity for operators and developers respectively.

General-case interfaces are meant for developers. Developers cannot always predict where/how their code will be deployed, and are hence often interested in the latency of their system when deployed in arbitrary environments. The general-case interface provides them with such a description by summarizing the impact of the environment on the system's latency using LCVs. While LCVs do reveal implementation details (e.g., `n_evicted, rehashing` reveal the use of a cuckoo-hash table), these details *are necessary* to summarize latency *for an arbitrary workload*, so they must be represented in the general-case interface.

We designed the deployment-specific interface for operators. Since operators are unfamiliar with the system's implementation and only care about the system's latency behavior in their particular deployment environment, the deployment-specific interface does away with the hard-to-understand LCVs by instantiating them with a distribution specific to that deployment. This enables the deployment-specific interface to summarize latency in an NF-generic way—any NF would normally involve a header check and state lookups—and be understood by almost any NF operator. Of course, it does reveal one important aspect of the implementation, namely the distinct fast and slow tables. However, this aspect (which would have no place in a semantic interface) is crucial to any bridge operator interested in latency.

That said, we do not envision the separation between the general-case and the deployment-specific interfaces being set in stone—developers may refer to the deployment-specific interface to understand latency in the face of specific workloads, while operators may refer to the general-case interface to understand latency beyond their expected workload.

4.5 Putting It All Together

In summary, we propose that a latency interface be a program that takes in the same inputs as the original program and returns its execution latency. We propose two kinds of latency interfaces—general-case and deployment-specific—tailored to the needs of developers and operators respectively. The general-case interface expresses latency as formulae with LCVs. This allows it to be both precise and complete while being readable for a developer who understands the program's implementation details. The deployment-specific interface expresses latency as concrete statistics tailored to a particular deployment environment. This allows it to be simple, abstract, and precise while being complete for the deployment and operator in question. Both interfaces come with a resolution that specifies the granularity at which the interface summarizes latency. The resolution provides readers of the interface with explicit control over the trade-off between readability and accuracy.

5 LINX: Automatically Extracting Latency Interfaces for Software Network Functions

In this chapter, we concretize our proposal for latency interfaces in the context of software network functions (NFs)—in-network packet processing applications such as load balancers, firewalls and NATs— and present LINX (**L**atency **IN**terface e**X**tractor)[1], a technique and tool to automatically extract latency interfaces from NF implementations. LINX takes as input NF code written in C and outputs general-case latency interfaces in the form of small Python programs that it can then specialize into deployment-specific interfaces for individual deployments.

The rest of this chapter is organized as follows: we first describe why we chose NFs as our starting point for latency interfaces (§5.1) before providing an overview of LINX's design (§5.2) and describing how it extracts general-case (§5.4) and deployment-specific (§5.4) interfaces from NF code, respectively. We then present our evaluation of LINX (§5.5), summarize related work (§5.8), and conclude (§5.9).

5.1 Why Network Functions?

We chose Network Functions (NFs) as our starting point for latency interfaces for three reasons:

NFs are ubiquitously deployed

NFs are an integral part of today's networks [19, 88, 145] and are used to implement a wide range of functionality such as security (e.g., firewalls, intrusion detection systems), performance (e.g., caches, WAN optimizers) and support for new applications and protocols (e.g., TLS proxies). Recent surveys of enterprise datacenters have shown that the number of NFs deployed in their networks is approximately equal to the number of routers [205, 207].

Understanding NF latency is critical

While NFs were traditionally deployed using dedicated, specialized hardware (ASICs), this changed in the early 2010s when major carriers began deploying NFs as software applications running on general purpose hardware, an approach known as Network Functions Virtualization (NFV) [172]. While NFV makes it easier and cheaper to deploy new NF functionality, it introduces the challenge of unpredictable latency because processing packets on general-purpose CPUs (as opposed to ASICs) can lead to significant latency variability [14, 68, 69, 156, 188, 194]. This latency variability directly impacts user-perceived latency since NFs are typically on the critical path of serving user requests. For example, any packet that enters a cloud provider's data center traverses at least one load balancer and typically also a firewall. A recent survey [163]

[1] Pronounced "Lynx"

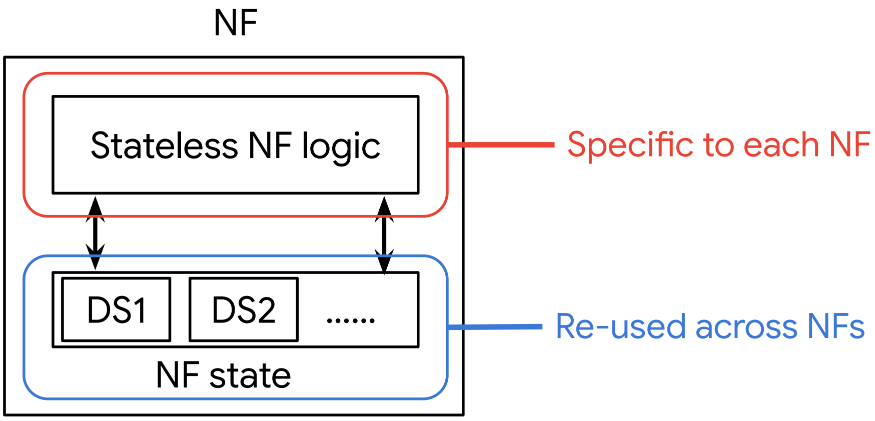


Figure 5.1: Typical architecture of NF code. DS refers to data structures that the NF uses to store mutable state.

of network operators found NF performance degradation due to unexpected workloads to be a frequent pain point, and such performance bugs to be among the hardest to diagnose.

NFs are amenable to automated program analysis

While most NFs maintain mutable state and are thus not directly amenable to exhaustive, automated program analysis, recent work [251, 252] has shown that such analysis is feasible because of how most (but not all) NF code is architected.

Fig. 5.1 illustrates the typical architecture of NF code. NF code typically consists of two distinct parts: stateless packet processing logic and cleanly separated NF state stored in data structures such as queues and hash tables. Many popular NF development frameworks (e.g., eBPF [246], NetBricks [186], FastClick [20], Vigor [251]) enforce this clean separation of state and provide a library of data structure implementations that all NFs written using the framework must use to store state. For example, all NFs written using the increasingly popular eBPF framework are architected as stateless modules that maintain state in cleanly-separated, kernel-maintained eBPF maps [76] with the stateless code being the only distinguishing aspect across NFs.

Vigor [251] leverages the above architecture to formally verify semantic properties of NFs as follows: First, the maintainers of the NF development framework manually verify each API call provided by the library of data structures and produce a semantic contract with pre- and post-conditions for each call. While this is a tedious task, it must only be performed once, with the manual effort being amortized across all NFs written using the framework. Once this is done, Vigor uses symbolic execution (SE) to automatically explore all execution paths through the stateless NF code, with the lack of state ensuring that path explosion is avoided. Finally, Vigor automatically combines the results of symbolic execution with the semantic contracts of the API calls that the NF uses to generate a proof that the NF as a while satisfies the target semantic properties. Note that Vigor's semantic proofs are unrelated to latency interfaces and do not provided any latency related information.

In summary, we chose NFs as our starting point for latency interfaces because they are an example of ubiquitously deployed systems code whose latency matters and are amenable to automated analysis which is necessary for the automated extraction of latency interfaces.

5.2 LINX Overview

LINX is a program analysis tool which takes as input an NF implemented in C and automatically extracts latency interfaces in the form of Python programs.

We designed LINX to meet two goals: (1) *minimal developer effort*: NF developers/operators should not need to write test suites or proof lemmas, and (2) *allow for proprietary NFs*: NF vendors typically provide operators with only binaries [166]; it's ok for them to provide a latency interface, but not source code.

Fig. 5.2 presents an overview of LINX. The NF developer gives the LINX back-end the NF source, augmented with a few single-line annotations akin to instantiating a type in a higher-level language. LINX combines this with a pre-analysis of the data structures used by the NF and extracts the general-case interfaces for all meaningful resolution ranges. The NF operator provides the LINX front-end with an NF binary and general case interface (provided by the NF developer), along with a (set of) packet trace(s) that represent the expected workload in their deployment. From these, LINX extracts the deployment-specific interfaces for all meaningful resolution ranges. NF developers/operators can also query LINX with a specific resolution, to get the interface at that resolution.

LINX currently supports three latency-related metrics: number of instructions, number of memory operations, and number of CPU cycles. For each metric, LINX outputs one set of Python programs; each set contains one Python program per relevant range of resolutions.

5.2.1 Limitations and Assumptions

LINX is designed for NFs written using popular development frameworks (e.g., eBPF [246], NetBricks [186], FastClick [20], Vigor [251]) that enforce cleanly-separated state and provide a common, pre-analyzed library of data structures that all NFs must use to store state. The LINX back-end uses exhaustive symbolic execution (ESE) [140] to automatically analyze the NF code. This requires that the NF logic (in addition to being stateless) be single-threaded, and all its loops except the top-level event loop have statically computable bounds. Many (not all) data-plane NFs meet these requirements. For instance, all NFs written using the eBPF [246] framework as stateless, single-threaded modules that keep their state in cleanly separated, kernel-maintained eBPF maps [76]. Counterexamples include TCP-terminating NFs and Intrusion Detection Systems (IDSes); running LINX on such NFs causes symbolic execution to time out due to path explosion stemming from symbolic pointers into the reconstructed TCP bytestream.

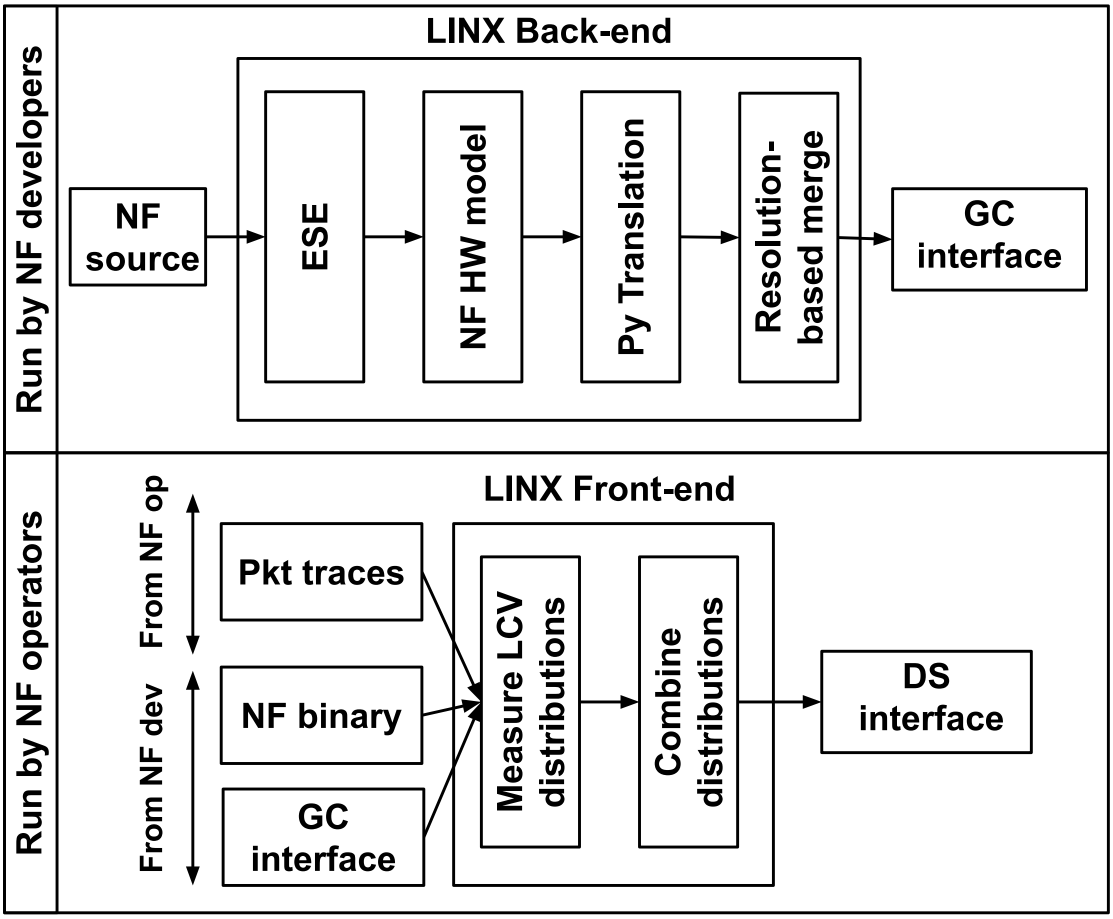


Figure 5.2: Overview of LINX. GC and DS refer to general case and deployment-specific respectively. ESE refers to Exhaustive Symbolic Execution.

LINX-extracted interfaces only summarize the processing latency for each packet and do not reason about queuing latencies. Reasoning about these latencies would require LINX to reason about multiple inputs together, and for this, we need to employ techniques more sophisticated than ESE [251]. Reasoning only about processing latency allows LINX to avoid reasoning about load-based variability since processing latency (unlike queueing latency) does not vary with load.

To capture how hardware affects latency with reasonable accuracy, LINX assumes that the NF runs pinned to a core and does not significantly contend for hardware resources, e.g., due to smart process isolation [44, 91, 156, 226]. We believe network operators keen on predictable latency are likely to employ such techniques.

5.2.2 Running Example

Algorithm 1 shows pseudocode for a simplified longest prefix match (LPM) IPv4 router that we use as a running example through the rest of the chapter. The router stores its state (the forwarding table) in a cleanly separated Patricia trie that exposes a single external method (line 8). The router first classifies packets based on whether they are IPv4 or not (line 2). Invalid packets are immediately dropped (line 6), thus incurring a constant performance cost. Valid packets lead to a lookup in the LPM data structure (line 3), which has a more complex performance profile (lines 10–17), with the number of loop iterations being data-dependent (see lines 12 and 15).

Algorithm 1: Simple LPM Router

```
 1 function processPacket (packet pkt)
 2 if pkt.etherType == IPv4 then
 3     dst_port = lpmGet (pkt.ipv4.dst_addr)
 4     FORWARD (pkt, dst_port)
 5 else
 6     DROP (pkt)
 7 end

 8 function lpmGet (bit ip[32])
 9 node = lpmRoot
10 for i in 0..31 do
11     b = ip[i]
12     if exists node.children[b] then
13         node = node.children[b]
14     else
15         break
16     end
17 end
18 return node.port
```

5.3 Extracting General Case Latency Interfaces

We now describe how the LINX back-end extracts general-case interfaces from NF source code. We first describe the manual pre-analysis whose results are reused across NFs, before describing the four stages illustrated in Fig. 5.2.

5.3.1 Pre-Analysis of Data Structures

To extract latency interfaces for any NF development framework, LINX requires the library of data structures provided by the framework to be manually pre-analyzed, with the analysis cost being amortized across all NFs that use the library. We believe such manual effort is reasonable because it is a rare effort (e.g., once per update to the data structure library) and it is done by the maintainers of the data structure library instead of its users. To illustrate, there were 34 new commits in Linux's eBPF maps last year [75] while the Cilium project [50] alone—just one among hundreds of projects that leverage eBPF maps—had an order of magnitude more commits during that same period [49]. Naturally, in this work, we played the role of the maintainers

```
def latency_interface_lpmGet(ip):
    # Metric: Instruction count
    # LCV: l: matched prefix length
    return 4*l + 2
```

```
def latency_interface_lpmGet(ip):
    # Metric: Memory Accesses
    # LCV: l: matched prefix length
    return l + 1
```

Figure 5.3: Latency interfaces extracted for the `lpmGet` method from Algorithm 1

Algorithm 2: `lpmGet` function model.

```
function lpmGet ( bit $ip[32]$ );
return <new symbol>
```

ourselves.

This pre-analysis involves three manual tasks for each method exposed by the library: (1) identifying the LCVs relevant to the method's implementation, (2) identifying unique execution paths through the method as a function of the LCVs, and (3) writing a simple symbolic model for the method.

While identifying LCVs and identifying unique execution paths as a function of these LCVs might seem daunting, we found it to be fairly simple in practice (taking ≤ 1 person-hour for someone familiar with the data structure code.). Our experiences corroborate those of independent prior work [107] which observed that most data structures require only a "few" LCVs, and identifying them is "straightforward". For instance, the `lpmGet` method in our running example only requires a single LCV "l", which denotes the matched prefix length; this LCV is sufficient because it fully captures how anything other than the input packet (in particular, the configuration of the LPM table) influences latency. Finally, LINX makes it easier to identify execution paths by providing developers with the control-flow graph of the method and have them express the path as a function of the basic blocks.

Given the above information, LINX extracts a simple latency interface for each method call in terms of the number of instructions and memory accesses as a function of the LCVs. Fig. 5.3 illustrates these interfaces for the `lpmGet` method from our running example.

Finally, writing a symbolic model is straightforward, since the model only needs to be detailed enough to differentiate between the execution paths identified above. For instance, our symbolic models for the eBPF map API required only 200 LOC in C across all API calls. Algorithm 2 illustrates the model for our running example.

5.3.2 Exhaustive Symbolic Execution (ESE)

In this step, LINX analyzes the NF source code to extract for each execution path a formula for the number of instructions and memory accesses executed along that path. This formula is expressed in terms of the LCVs specific to the implementation of the NF and the data structures it uses. We refer to these LCVs as hardware-independent LCVs henceforth.

To achieve this, LINX generates a special build of the NF code where all calls to stateful methods are replaced at link time with calls to corresponding symbolic models. For example, in our LPM router, the call to `lpmGet` is replaced with a call to the symbolic model shown as Algorithm 2. Next, LINX symbolically executes this special build exhaustively and obtains all feasible execution paths through the stateless NF code. For our LPM router, this results in 2 paths, one for valid IPv4 packets and one for invalid packets. For each execution path, LINX also obtains symbolic path constraints, which consist of two categories of constraints: (1) constraints on NF inputs that cause it to go down the particular execution path and (2) constraints on the abstract state of each data structure, before and after each call to a stateful method. The second category of constraints tells LINX how stateless and stateful code interact along the execution path.

Once it has obtained all feasible execution paths and their path constraints, LINX analyzes each path: First, it passes the path's constraints to a solver to obtain concrete inputs that exercise the path; these inputs include a packet, as well as values for any symbols generated by the symbolic models of the stateful methods. For our LPM router, one path will yield a concrete invalid IPv4 packet, while the other will yield a concrete valid IPv4 packet and a concrete port that would result from the LPM lookup (e.g., port 0). Next, for each of these concrete inputs, LINX replays the NF execution and obtains a unique trace of machine instructions.

Finally, LINX characterizes the latency of each feasible execution path by stepping through the corresponding instruction trace: it traverses the trace, adding the number of instructions and memory accesses, until it hits a call to a modeled method; when this occurs, it picks the right branch of the method's latency interface based on the constraints on the abstract state of the data structure. While our simple `lpmGet` example has no branches, this is typically not the case for more complex data structures and methods. For example, the latency interface of a flowtable `get` method will have different formulae depending on whether the flow is present or absent in the flow table. In such a scenario, LINX uses the path constraints to pick the right formula.

5.3.3 Hardware Model for NFs

This step characterizes the latency of each execution path of the NF in terms of hardware-dependent metrics (CPU cycles), by introducing hardware-dependent LCVs; i.e., LCVs that capture the interaction between NF and hardware.

LINX uses the notion of a CPI (Cycles Per Instruction) stack [82] to compute the number of CPU cycles of an execution path. A CPI stack breaks down the average CPI for a program executing on a given microprocessor into a base CPI plus various CPI components that reflect "lost" cycle opportunities due to miss events such as branch mispredictions and cache/TLB misses. In general, replicating a perfect CPI stack is infeasible—it is equivalent to analyzing each execution path to the depth provided by a cycle-accurate simulator.

We leverage NF-domain knowledge to eliminate CPI components and pick only the necessary set of hardware-dependent LCVs. When an NF runs pinned to a core and with limited contention for hardware resources, the dominant hardware factor that influences its latency is the last-level cache (LLC) [68, 156, 226]. Hence, LINX introduces only two hardware-dependent LCVs—*base_CPI* and *LLC_miss_latency*—and expresses a path's CPU cycle count as $instructions \cdot base\_CPI + LLC\_misses \cdot LLC\_miss\_latency$. Note, while LINX uses the same two LCVs for all NFs, the values of these LCVs vary with each <*NF, HW*> pair (§5.4). To track possible LLC misses, LINX leverages taint-analysis [204] to identify independent heap accesses specific to the current input; it then branches on each such access, with one outcome being an LLC miss and the other an LLC hit.

5.3.4 Python Translation

The previous steps specify an NF execution path as a set of symbolic constraints on the input packet and symbols arising from calls to data structures; this step translates these constraints into human-readable Python code and outputs a general case latency interface of the NF with a resolution of 1.

LINX translates symbolic constraints on the input packet using knowledge of the header format of the popular networking protocols (e.g., IPv4, TCP, QUIC). For instance, the constraint $pkt[23:24] == 6$ on a non-tunneled IPv4 packet is translated to *pkt.isTCP* since the packet's 24^{th} byte specifies the transport protocol, and value 6 corresponds to TCP.

LINX translates symbols arising from calls to data structures using call context and developer-provided annotations (one annotation per instantiated data structure). Fig. 5.4 illustrates such a translation: Line 2 shows a developer's annotation for a data structure of type `map`: it indicates that this NF uses this map as a `"macTable"`, which maps `"ethaddr"` keys to `"port"` values; these are human-friendly terms chosen by the developer to help the generation of simple latency interfaces. Line 5 shows a constraint derived from the NF code that concerns this map. Line 7 shows how LINX rewrites this constraint because it knows that this is a call to `bpf_map_lookup_elem()` with an argument corresponding to bytes $7-11$ of the input packet. Line 9 shows how LINX further rewrites the constraint because the developer's annotation enables LINX to identify the given bytes as the input packet's source MAC address.

The annotation on Line 2 is the only annotation that the NF developer needs to provide. We believe such one-line annotations are reasonable since they are similar to instantiating a type in a higher-level language.

5.3.5 Resolution-Based Merging

While the interface is now an understandable Python program its cyclomatic complexity is still equal to that of the NF's implementation. This step uses the notion of resolution to simplify the la-

```
# Developer annotation:
DS_INIT(&map,"macTable","ethaddr", struct eth_addr,"port", int);

# Starting condition derived from implem:
if bpf_map.unnamed_symbol
# Transform based on called library function
if bpf_map.contains(pkt[7:12])
# Transform based on developer annotation
if macTable.contains(pkt.src_mac)
```

Figure 5.4: Example of LINX's constraint rewriting.

tency interface as follows: First, it calculates the maximum latency impact of each constraint, i.e., the maximum latency difference between two execution paths that only differ w.r.t this constraint. The set of distinct "maximum latency impacts" forms the minimal set of resolution thresholds that yield latency interfaces with different complexity. Then, it eliminates all constraints with an impact smaller than the target resolution.

5.4 Extracting Deployment-Specific Latency Interfaces

To extract a deployment-specific interface, the LINX front-end takes as input the NF binary and its general case interface[2], provided by the NF developer/vendor; along with a (set of) deployment-specific packet trace(s), provided by the NF operator. It then runs the NF binary using the packet trace(s) as input, infers the deployment's LCV distributions, and instantiates the deployment-specific interface. Running the NF allows LINX to extract accurate deployment-specific interfaces since it can precisely measure the impact of the NF's environment on latency as opposed to modeling it.

LINX infers three LCV distributions per NF, deployment:

Hardware-independent LCVs: For each packet in the provided trace(s) LINX computes the values of each hardware-independent LCV. It then computes a joint probability distribution of these LCVs, since they tend to be highly correlated. While this is not necessary for our LPM router which only has one LCV, it is needed for complex NFs, such as the bridge in Fig. 4.1 since its LCVs are highly correlated. (e.g., `n_stale` and `n_evictions` are both functions of `occ`).

Base CPI: LINX measures the base CPI using hardware performance counters [223] available on all major processors today. Since the packet trace(s) may not exercise all execution paths, LINX assumes the same base-CPI distribution across all paths, and it provides warnings if it detects significant differences (e.g, some paths use expensive x86 instructions, like integer divide, while others don't). We think this is a reasonable assumption because the base CPI is only a function of the instruction mix (it does not include any miss events). In §5.5.2, we experimentally validate this.

[2] The operator cannot be certain that this general case interface is accurate for the production binary, but we do not see this as a barrier to adoption: operators routinely deploy NF binaries while relying only on non-attested configuration interfaces and vendor manuals [166].

LLC miss latency: Measuring the distribution of LLC miss latency ideally requires sophisticated NF-specific testing [188], to account for the NF's particular instruction- and memory-level parallelism. LINX avoids this because it targets NFs that keep all their state in a relatively small set of pre-analyzed data structures. For each data structure, we craft a microbenchmark that triggers LLC misses.[3] LINX estimates the LLC-miss-latency distribution of each data-structure call in a given deployment, by running the corresponding microbenchmark on the deployment's hardware. In §5.5.2, we experimentally show that our approximation performs well in practice (avg. error of < 10%). Note, our approximation concerns the *latency* introduced by LLC misses, *not the number* of LLC misses—the LINX back-end tracks LLC misses per path using taint analysis (§5.3.3).

Finally, LINX instantiates each formula in the general case interface with these inferred distributions to compute the requested latency statistic (e.g., 50^{th} percentile in Fig. 4.3). We show further examples of deployment-specific interfaces and their distributions in §5.5.

5.5 Evaluation

In this section, we address two main questions: (1) does LINX extract good latency interfaces, and (2) can latency interfaces make NF developers and NF operators more productive? To answer the former, we quantitatively evaluate the complexity of LINX-extracted interfaces, their accuracy, and the time it takes to obtain them (§5.5.1, §5.5.2, and §5.5.3, respectively). To answer the latter question, we show how developers can use LINX-extracted interfaces to catch latency regressions and fix latency bugs (§5.5.4). We then show how operators can use interfaces to pick the NF variant best suited for their target hardware and to perform root-cause diagnosis of latency anomalies (§5.5.5).

We evaluate LINX on 12 dataplane NFs that cover a wide variety of functionality and network protocols (Table 5.1). These include the Katran load balancer used in production at Facebook [208], the Natasha NAT used in production at Scaleway [169], the XDP packet filter from the Cilium project [50] and an implementation of Google's Maglev load balancing algorithm [79]. The NFs were written using DPDK [70] and eBPF XDP [246], arguably the two most popular frameworks today for building high-performance software NFs. VigNAT, Policer, Router and Bridge come from the Vigor project [251], the CRAB load balancer from [141], and the hXDP firewall from [32]. The Vigor and eBPF NFs are written in the commonly used stateless/stateful split model, which makes them amenable to exhaustive symbolic execution. We modified Natasha and DPDK NAT to also have such a clean split; this took ~3 person-days per NF.

The latency metrics we use for DPDK-based NFs are `x86` instruction count, `x86` memory access count, and `x86` CPU cycles (thus wall-clock time). Note, LINX is not specific to x86 and can just as easily predict the corresponding metrics for another ISA (e.g., ARM) if the LINX front-end is given the corresponding binary. For eBPF NFs, we only analyze the NF itself, and not the eBPF

[3] The maintainer of the library must do this once per data structure, like the pre-analysis.

Framework	NF	Functionality
eBPF XDP	Katran LB	Per-flow state, per-VIP state, consistent hashing, IPv6, ICMP, QUIC, tunneling
	Cilium filter	Longest prefix matching, IPV6
	CRAB LB	Read-only state
	hXDP firewall	Per-flow state
DPDK	Natasha NAT	Per-flow state, handles fragmentation, UDPLite, ICMP, ARP
	Maglev LB	Per-flow state, consistent hashing
	VigNAT	Per-flow state, header rewriting
	Bridge	Packet duplication
	Router	Longest prefix matching
	Policer	Per-flow state, fine-grained timing
	DPDK NAT	Per-flow state, header rewriting, cksum offload
	DPDK firewall	Per-flow state

Table 5.1: Network functions used to evaluate LINX.

maps that are part of Linux, so we only report hardware-independent metrics.

Our testbed consists of two directly connected servers: a device under test (DUT) and a traffic generator and sink (TG). The servers are identical, with an Intel Xeon E5-2667 v2 processor @ 3.30 GHz, 32 GB of DRAM, and Intel 82599ES 10-Gbps NICs. The DUT runs one of the NFs and measures the latency, while the TG uses MoonGen [80] to generate traffic.

5.5.1 Are LINX-Extracted Interfaces Easy to Read?

To evaluate the readability of the latency interfaces, we (1) measure their complexity in terms of both lines of code (LOC) and cyclomatic complexity (CC) [233], and (2) evaluate whether the primitives exposed by the latency interfaces are those that NF developers and operators are familiar with.

Table 5.2 illustrates the complexity of the LINX-extracted interfaces measured as a fraction of LOC and CC of the implementation. We see that the extracted interfaces have 26−210× fewer LOC than the corresponding implementations and are 3−124× less cyclomatically complex, ignoring CRAB, which is already simple to start with. The more complex an NF, the higher this reduction in complexity, which argues for the real-world utility of latency interfaces.

Fig. 5.5 illustrates the impact of varying resolution on the complexity of Katran's latency interface. At the finest granularity, Katran's instruction-count interface is fairly complex (LOC=9675, CC=3226 independent paths) and contains more LOC than the implementation since LINX's Python translator explicitly lists each program path individually. At the other end of the spectrum, since no two packets in Katran can incur an instruction count that differs by more than 854 instructions (number determined by LINX and verified by us), for resolutions above 854 the interface becomes a simple upper bound. In between these two extremes, we see how low-level

NF	Implementation		HW-independent interface		HW dependent interface	
	LOC	CC	LOC	CC	LOC	CC
Natasha	2932	192	1.8%	8.9%	2.8%	15.1%
Maglev	3168	29	0.9%	37.9%	1.6%	65.5%
VigNAT	2770	22	0.7%	36.3%	0.9%	52%
Bridge	2837	219	0.5%	2.7%	2.1%	10.5%
Router	1260	17	0.4%	17.6%	1.0%	29.4%
Policer	2466	16	0.4%	31.2%	0.6%	37.5%
DPDK FW	2508	21	0.8%	38%	1.0%	45%
DPDK NAT	1780	35	0.6%	27%	0.9%	39%
Katran	2661	3226	2.8%	0.8%	-	-
Cilium filter	784	42	3.2%	14.3%	-	-
CRAB	437	4	2.0%	100%	-	-
hXDP FW	312	33	3.8%	15.1%	-	-

Table 5.2: Complexity of extracted interfaces vs NF implementation. "(x%)" means "x% of implementation". For each NF, the complexity is calculated for an interface with resolution equal to 10% of the maximum latency variability the NF can exhibit.

details get abstracted away—for instance, at resolution=50 instructions, we see a 125× drop in complexity (LOC=75, CC=26).

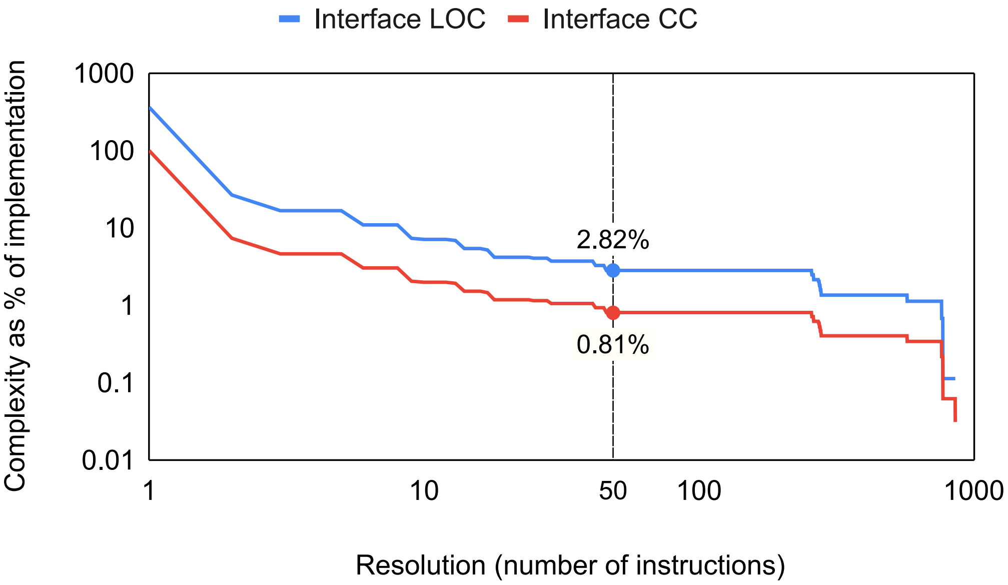


Figure 5.5: Impact of varying resolution on the size (LOC) and complexity (CC) of Katran's latency interface. Both axes are in log scale.

We conclude that LINX-extracted latency interfaces are significantly simpler than the NF implementations. The notion of resolution succeeds in abstracting a latency interface, giving the reader a knob with which to control the amount of detail contained in the interface.

```
def latency_vignat_gc(pkt):
  # Metric: x86 instructions
  # Resolution: 10
  # NF state:
  #   flowtable
  # LCVs:
  #   e - expired flows
  #   t - bucket_traversals
  #   c - hash_collisions

  x = 19*e*t + 40*e*c + 227*e + 123

  if not (pkt.is_IP) or not(pkt.is_TCP or pkt.is_UDP):
    return x + 7
  else:
    if pkt.port != internal_network_port:
      if flowtable.contains(pkt.flow):
        return x + 289
      else:
        return x + 68
    else:
      if flowtable.contains(pkt.flow):
        return x + 18*t + 30*c + 395
      else:
        return x + 31*t + 30*c + 547
```

Figure 5.6: Extracted general case interface for VigNAT.

We now evaluate how familiar the interface looks to a human reader. We show an example of the general case interface for VigNAT in Fig. 5.6, restricted to TCP/UDP packets for space considerations. The interface is a succinct, self-descriptive Python program. The conditions in `if` statements are expressed in terms of fields in the input packet header (e.g., `pkt.port`) or semantic operations on data structures (e.g., `nat_flowtable.contains`), which are primitives we expect both developers and operators to understand. Being a stateful NF, VigNAT's latency is influenced by NF state, and the interface reflects this via LCVs, documented in the header.

Finally, we illustrate the impact of deployment-specific instantiation of interfaces on their readability. Fig. 5.7 shows the interfaces for VigNAT's 50th and 95th percentile latencies and the distribution underlying them, for a particular <*workload, HW*> pair. The deployment-specific instantiation turns each formula (expressed in terms of LCVs in the general case interface) into concrete values specific to the environment and workload, thus tailoring the interface to an operator's needs. The latency CDF also enables interested operators to understand how VigNAT's percentile latency varies.

5.5.2 Do LINX-Extracted Interfaces Predict Latency Accurately?

We now evaluate the prediction error of LINX-extracted interfaces, i.e., the difference between the latency predicted by the interfaces and the measured latency.

To do so, we use LINX to extract interfaces for all 8 DPDK NFs[4] for two hardware-independent metrics (`x86` instructions and memops) and one hardware-dependent one (`x86` cycles). For each NF, we instantiate two deployment-specific interfaces corresponding to two very different

[4] LINX does not support HW-dependent metrics for eBPF NFs

```
def latency_vignat_ds(pkt):
  # Metric: CPU cycles
  # Resolution: 200
  # Statistic: 50th percentile
  # NF state:
  #   flowtable

  if flowtable.contains(pkt.flow):
    return 301
  else:
    if pkt.port !=
  internal_network_port:
      return 92
    else:
      return 558
```

```
def latency_vignat_ds(pkt):
  # Metric: CPU cycles
  # Resolution: 200
  # Statistic: 95th percentile
  # NF state:
  #   flowtable

  if flowtable.contains(pkt.flow):
    return 395
  else:
    if pkt.port !=
  internal_network_port:
      return 97
    else:
      return 1037
```

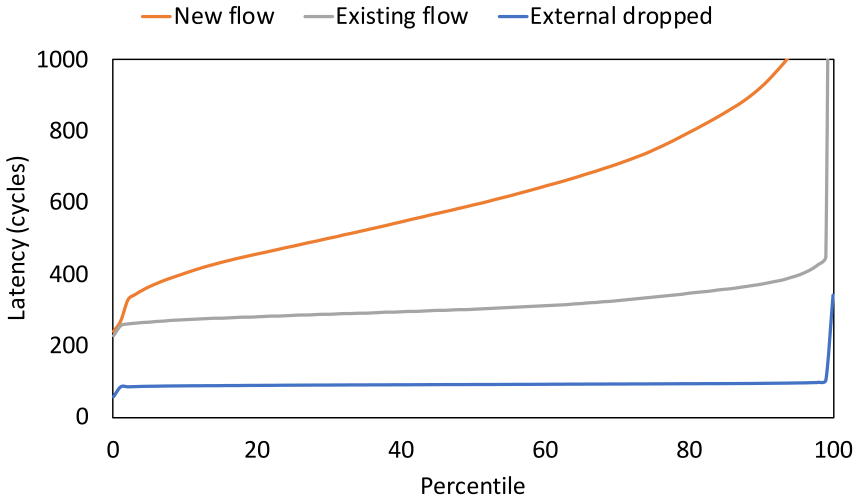


Figure 5.7: Deployment-specific interfaces for VigNAT (50^{th} and 95^{th} percentile) and the latency CDF (resolution=200 cycles)

deployments—typical traffic representative of university networks [25] and adversarial traffic that seeks denial-of-service [188]. The above deployments represent opposite ends of the spectrum for *absolute* NF latencies [188]— e.g., adversarial traffic incurs 2.1× greater latency than typical traffic in VigNAT. To instantiate each deployment-specific interface, we use PCAP traces of 100M packets each. These traces are similar to what an operator could obtain with `tcpdump` on their domain gateway and are not specific to any particular NF implementation.

For ground-truth measurements, we manually generate synthetic packet traces for each <*NF, deployment*> pair akin to Scaleway's NAT test suite [168]. We play back these traces against the NF and measure the latency of each packet (the ground truth). Note, the synthetic traces are only used to measure the ground truth and not for predicting latency, thus avoiding any overfitting.

As a baseline for CPU cycles, we replace LINX's empirically derived hardware model with a state-of-the-art Worst Case Execution Time (WCET) model [124]. For compute instructions, the model conservatively assumes the worst case latency cost of each instruction as reported in the Intel manual [120] due to the proprietary nature of out-of-order (OOO) instruction scheduling within the processor. For memory instructions, thw WCET model conservatively assumes that every memory access is serviced from main memory unless it can definitively prove otherwise, based on previous memory accesses. Being a conservative model, it does not take into account memory-level parallelism (MLP), or pre-fetching.

We present the prediction error for the 50th percentile, 90th percentile and 99th percentile latencies (which is the point at which LINX's limitations become evident). We compute all prediction errors by subtracting the relevant statistic of the measured latency distribution from that of the predicted latency distribution. The results reported are at resolution 1, where LINX does the worst.

Table 5.3 reports the results for the two hardware-independent metrics—instruction count and number of memory accesses. We find that, even in the worst case for LINX (i.e., finest resolution), it can accurately predict these metrics (≤4% error) irrespective of the percentile. The small prediction error arises due to small differences between the analyzed code (linked against models) and the production build (linked against real data structure implementations). This error vanishes at any reasonable resolution and LINX becomes 100% accurate.

Percentile	Instruction count		Memory Accesses	
	Typical traffic	Adversarial traffic	Typical traffic	Adversarial traffic
50^{th}	1.5% (1.8%)	1.2% (1.7%)	1.6% (4%)	1.5% (3.7%)
90^{th}	0.9% (1.4%)	0.9% (1.2%)	1.2% (3.6%)	1.2% (3.1%)
99^{th}	0.9% (1.3%)	0.8% (1.2%)	1.1% (3.5%)	1.2% (2.9%)

Table 5.3: LINX's average (maximum) prediction error for hardware-independent metrics for typical (Typ) and adversarial (Adv) traffic.

Table 5.4 reports the results for CPU cycles, in comparison to the WCET-based model. At the 50^{th} and 90^{th} percentile, LINX has a prediction error that is larger than the error for the hardware-independent metrics (≤26%). This increased error is due to the overhead of the instrumentation used to measure the CPI and LLC miss latencies. Nevertheless, LINX's accuracy is an order of magnitude better than WCET's since LINX reasons about hardware latency as a distribution, while WCET only models the worst case.

LINX cannot accurately predict the latency at the very end of the tail (nor can WCET). LINX's predictions have an error of ≤61% (average 22%), while WCET's predictions have an error of ≤45% (average 14%).

It is interesting to note that LINX *underestimates* the 99th percentile latency while WCET *overestimates* it; this contrasting behavior is due to the different hardware models underlying the two tools. LINX underestimates the 99th percentile latency since its simple hardware model ($instructions * CPI + LLC\_misses * miss\_latency$) is invalid at this percentile where other hardware aspects also impact latency significantly. WCET, on the other hand, overestimates the 99th percentile latency since its hardware model is designed to estimate the absolute worst-case latency. However, LINX's simple hardware model enables it to accurately predict latency at all percentiles except the tail, a task that the WCET model is incapable of.

Percentile	Tool	Typical traffic	Adversarial traffic
50^{th}	LINX	11% (26%)	9% (24%)
	WCET	164% (308%)	103% (186%)
90^{th}	LINX	10% (22%)	7% (19%)
	WCET	122% (234%)	94% (153%)
99^{th}	LINX	-19% (-54%)	-22% (-61%)
	WCET	14% (45%)	12% (39%)

Table 5.4: LINX's average (maximum) prediction error for CPU typical in comparison to a WCET-based model for typical (Typ) and adversarial (Adv) traffic.

5.5.3 How Long Does LINX Take to Extract Latency Interfaces?

Table 5.5 shows the time it takes LINX to extract the general case interfaces for all the NFs in this evaluation. We believe that these numbers make it feasible to incorporate latency interfaces extraction part of the regular NF development cycle, e.g., as part of continuous integration.

NF	Time (mins)
Natasha	15
Maglev	5
VigNAT	4
Bridge	17
Router	0.73
Policer	3
DPDK FW	4
DPDK NAT	6
Katran	32
Cilium filter	0.43
CRAB	0.15
hXDP FW	0.23

Table 5.5: Time taken by LINX to extract the general case interfaces.

The time required to obtain the deployment-specific interface is largely a function of the time required to run the provided workload. In our experiments, we ran PCAP files with 100M packets, and it took LINX <= 5 mins to generate the deployment-specific interface for a given <*workload, HW*> pair from the general-purpose interface, regardless of NF.

Our evaluation thus far supports the belief that LINX is practical: the complexity of extracted interfaces is significantly lower than the NF implementation, their accuracy is high, and the time to extract is reasonable.

5.5.4 Are Interfaces Useful to NF Developers?

In this section, we present two workflows that NF developers can use to understand and debug the latency behavior of their code, respectively.

Flagging latency regressions

Programmers often introduce involuntary latency regressions. Using test suites to catch such regressions is not easy, because they require environment setup, are fragile, and take long to run. We show here how a developer or a tool can instead compare the latency interface before and after a commit to identify latency regressions more quickly, conveniently, and precisely than with a latency test suite.

We wrote a script that retrieves each Katran commit and uses LINX to extract the corresponding instruction-count interface, at resolution=1. For each pair of commits a and b, there is a corresponding pair of interfaces S_a and S_b. The script finds the maximum latency (in terms of LLVM instruction count) predicted by each of the two interfaces and compares the two. We report LLVM (not eBPF bytecode) instructions since LINX builds on KLEE which interprets LLVM IR. Reporting eBPF instructions would require us to build on a tool that interprets eBPF bytecode (e.g., Serval [170])—this is an engineering task we leave to future work. We run LINX on all commits to the eBPF portion of Katran's code.

Table 5.6 shows the commits where a latency regression occurs. Over the past three years, the maximum latency for new flows regressed by 14.6%.

Commit ID	Latency before [LLVM instrns]	Latency after [LLVM instrns]	Latency regression [%]
Orig commit	-	1771	-
873d0501695c	1765	1896	7.42%
39e58b530a8a	1896	1914	0.95%
458aa0907b68	1914	1933	0.99%
15f81d0e7ec6	1930	1946	0.83%
74c3338c2f7e	1952	1983	1.59%
d0790d3a3823	1983	2030	2.37%
All commits	1771	2030	14.62%

Table 5.6: Latency regressions in Katran (handling new flows).

We imagine using this workflow as part of continuous integration (CI) to automatically identify unintended latency regressions. The CI system can present to the developer a before-and-after comparison of latency that directly highlights for which classes of inputs the regression occurs and what the magnitude of the regression is. Compared to performance tests, this workflow consumes less developer time and fewer resources and offers better completeness.

Fixing performance bugs

By helping developers understand the code's latency more quickly and deeply, interfaces can help fix performance bugs. We illustrate this with two examples of performance bugs in the map used by Vigor NFs [151].

The top of Fig. 5.8 shows a snippet of the latency interface of the `contains` operation in libVig's `map`.

```
if map.contains(key): # --- BEFORE ---
  if not(cached(key)):
   # Warning: 2*t integer divides
   return (4*t)*miss_latency + (21*t+27)*CPI
....
```

```
if map.contains(key): # --- AFTER ---
  if not(cached(key)):
   return (1*t)*miss_latency + (18*t+27)*CPI
....
```

Figure 5.8: Interface for `map_contains()` before and after the bug fix. `t` is the LCV for traversals in the hash ring.

The first red flag is the warning issued by LINX itself, based on tracking of expensive x86 instructions that adversely impact CPI. Looking for integer divides in the `map` code, we found that, on each traversal, it uses two costly modulo operations. To fix the issue, we replaced them with one bitwise `and`.

The second red flag is that each traversal requires 4 independent heap accesses (`4*t`). It turns out that `key` metadata is being stored in four distinct arrays of `int` elements. Our fix was to encapsulate `key`'s metadata in a single `struct` and use a single array with elements of this `struct` type. The rest remained unchanged.

Table 5.7 shows the impact of our fixes, based on Vigor's benchmarks: the two fixes, together, improve NF latency by 22% on average, and throughput by 19%.

NF	Throughput [Mpps]			Change	Latency [ns]			Change
	Orig	Fix 1	Fix 2		Orig	Fix 1	Fix 2	
VigNAT	3.88	4.36	4.68	20.62%	317	276	236	25.55%
Bridge	3.05	3.59	3.62	18.69%	410	332	323	21.22%
Maglev	2.58	2.86	3.04	17.83%	482	423	391	18.88%

Table 5.7: Throughput and latency of three NFs using `map`, shown before/after each performance bug fix.

This example shows how latency interfaces can not only flag possible performance issues but also guide the developer in where to look to improve performance.

5.5.5 Are Latency Interfaces Useful to NF Operators?

Operators typically care about how an NF performs in their specific deployment, not in general for everyone's deployment. We show how operators can use latency interfaces to pick the NF variant best suited to their hardware and to do a root-cause diagnosis of deployment-specific performance anomalies.

Which NF variant for my NIC?

Modern NICs provide the ability to offload specific tasks (like checksums and encryption) to specialized hardware. It is therefore useful to know which variant of an NF takes max advantage of the offloads available on a NIC.

Fig. 5.9 shows the interfaces for two variants of a NAT, and the interaction with checksum offload on Mellanox ConnectX-4 [162] and Intel `ixgbe` [122] NICs. The formally verified VigNAT does not do any offloading, whereas DPDK NAT does. The strings in the `if` conditions on lines 3 and 6 are identical to the one used by the NIC driver to identify itself [71]. The interface also shows the difference in latency: `ixgbe` requires the software to compute a pseudo-header checksum, whereas ConnectX-4 allows full offload, so it has lower latency.

```
1 # Snippet from VigNAT interface
2 if flowtable.contains(pkt.flow):
3  return  18*t + 30*c + 518  # No offload
4 else:
5   ....
```

```
1 # Snippet from DPDK NAT interface
2 if flowtable.contains(pkt.flow):
3  if(NIC_family == "net_mlx5"):
4   return  18*t + 30*c + 265 + cksum_offload()
5  else:
6   if(NIC_family == "net_ixgbe"):
7    return 18*t + 30*c + 478 + cksum_offload()
8   else:
9    return 18*t + 30*c + 564
10 else:
11   ....
```

Figure 5.9: Interfaces for VigNAT (top) and DPDK NAT (bottom): VigNAT does checksums in software, while DPDK NAT offloads checksums to the NIC as much as possible.

Based on this latency interface, an operator can make an informed deployment decision: if using `ixgbe` NICs, choosing the verified VigNAT makes sense; else, it's a trade-off to make carefully. The example also illustrates the benefit of performance abstraction in interfaces: even if an operator has access to NF source code, reading an interface that is orders of magnitude simpler helps answer questions more quickly.

Why do I get bad performance?

NFs running in production can face workloads that trigger surprising performance degradation. To address such anomalies, operators must first diagnose the root cause, and this often takes a lot of work.

LINX helps the search for a root cause by providing a list of possible explanations for the observed performance, ranked by likelihood. Given a problematic workload and an NF (or its general case interface), LINX instantiates the LCVs in a deployment-specific manner and then measures the distributions for each LCV and the NF latency. It then ranks the LCVs based on the correlation between the latency distribution and that of the LCV (using least-square fit linear regression).

To illustrate this workflow, we refer to three performance bugs that span both hardware and software root causes, shown in Table 5.8. The first bug occurs due to the uniform random workload causing hash collisions in a widely used hash function [128] used by Bridge; typical workloads with Zipfian distributions do not suffer from hash collisions. The second bug is caused by VigNAT's batches expiry of flows, which results in a latency spike that only becomes evident for traffic with high churn. The third bug occurs when the active flowtable in Maglev overflows the last-level cache of the server; this makes the latency spike highly dependent on LLC configuration.

Bug	**Root cause**	**Identified as most-likely cause?**
Spike in median latency of Bridge for uniform random workload	`hash-collisions`	Yes
Spike in tail latency of VigNAT due to high churn	`expired-flows (batched)`	Yes
Spike in median latency of Maglev on a particular x86 server	`active-flowtable-size`	Yes

Table 5.8: Performance bugs used for root-cause diagnosis.

For each bug, we generated a workload that triggers it and provided the PCAP file to LINX, along with the general case interface of the corresponding NF. For each bug, LINX correctly reported the culprit LCV as the most likely root cause. Of course, LINX can only track bugs that arise from LCVs it accounts for. It would be unable, for instance, to identify the root cause for a latency spike due to LLC evictions caused by a noisy neighboring process, since LINX does not account for contention.

This example illustrates how LINX can help focus the operators' attention on likely explanations for the performance they observe, thereby reducing the amount of work needed to find the root cause.

In conclusion, our evaluation shows that LINX is practical: the complexity of extracted interfaces is significantly lower than the NF implementation, their accuracy is high, and the time taken to extract them is reasonable. Further, NF developers and operators can use these interfaces to identify performance regressions, diagnose and fix performance bugs, and pick the NFs that are best suited to their hardware.

5.6 Does LINX Generalize Beyond NFs?

In this section, we explore how LINX can generalize in two directions: (1) programs other than NFs, that are nevertheless still amenable to ESE; and (2) NFs that are not amenable to ESE. Overall, we find that the design of LINX—split into a modular back-end and front-end that produce general case and deployment-specific interfaces, respectively—enables generalization by adapting just the necessary modules in the LINX pipeline.

Beyond NFs: We have successfully applied LINX to the OpenSSL library, to uncover digital side-channels, and to eBPF extensions for user-space file systems.

Extracting interfaces for finding digital side-channels required modifying only LINX's hardware model (i.e., step 1 in the back-end). Implementing a new model focused on sources of constant-time violations (using the exhaustive list in [8]) took us 2 person months. We ran LINX on 12 cryptographic primitives from OpenSSL 3.0 [181] and found a constant-time violation in the AES cipher unpadding function. This violation was acknowledged by the OpenSSL maintainers [183]. We have submitted a pull request [184] that has undergone multiple rounds of review and is in the final stages of getting merged.

Our experience with OpenSSL reinforced our belief (from §5.5.4) that a tool that automatically extracts latency interfaces would be of great use to developers. For example, we learned that the violation we uncovered had been latent since OpenSSL 1.1.1 because the developer "just reused the code" and had somehow been missed despite the extremely thorough code reviews that OpenSSL goes through. If latency interfaces of the OpenSSL code were extracted regularly, e.g., as part of continuous integration, it is unlikely that this violation would have persisted for this long.

Extracting interfaces for eBPF file system extensions was more straightforward since the code is similar to that of eBPF NFs. Here, we only had to add translation rules (step 2 in the LINX back-end) corresponding to the supported system calls. This took 4 person-days, after which LINX was able to automatically extract interfaces for the extFUSE extensions [27].

Code not amenable to ESE: To evaluate the limits of LINX's ESE-based approach, we used LINX on Snort [213], a popular IDS that independent prior work has shown to not be amenable to ESE [161, 244]. Our results corroborated those from prior work; while LINX did extract latency interfaces for the networking stack and all detection rules that look only at packet headers, attempting to extract a complete interface caused LINX to time out. Extracting an interface from Snort with LINX requires either that we modify its code to cleanly separate the stateful components, or that we replace the symbolic execution engine in the LINX back-end with a manual theorem prover.

5.7 Comparison to Freud

The work closest to ours is Freud [201] which takes as input a binary and a test suite, and outputs an expression of latency as a function of input and global variables. Freud strikes a different generality/accuracy balance than LINX: It is more general, in the sense that it can run on any program—not just NFs that are amenable to ESE—and requires no source code and no data-structure pre-analysis. However, it is less accurate, in two ways: (a) It cannot reason about the performance of execution paths that are not triggered by the test suite (since it does not analyze the source code). (b) It cannot reason about how past inputs affect performance in stateful code (since it does not know anything about the data structures where the state is stored).

Nevertheless, we evaluated Freud to see whether it could indeed extract latency interfaces, especially in the context of NFs. We used the publicly available Freud code [90] at commit ID e6e7a91006.

We used Freud on three classes of programs: (a) A stateless program that spins for a period of time proportional to the input length. (b) Data structures commonly used by NFs: a longest prefix match (LPM) trie and a hash map. (c) NFs: VigNAT (academic prototype), Natasha (production NAT used at ScaleWay), and Maglev (DPDK implementation of Google's load balancer). Natasha comes with an open-source performance test suite [168], making it an ideal fit for Freud. For the remaining programs, we used as test suites the packet traces on which we evaluated LINX.

Table 5.9 summarizes our results, discussed below.

Freud-vanilla: First, we ran Freud on unmodified programs, and it behaved as expected: It successfully characterized the spinning program's runtime as a function of the input length, but it could not produce meaningful performance annotations for the data structures or NFs. This is normal, since, in the latter programs, the latency is a function of implicit variables that capture the interaction between current and past inputs (e.g., number of iterations of `while(bucket[i].is_full ==1)`).

Freud-nf: Next, to compare with LINX more fairly, we explicitly modified our programs to work with Freud: we identified conditions that we knew impacted performance (essentially LCVs) and manually added them as global variables (which Freud tracks). For instance, in the hashmap, we added a global variable to explicitly track the number of collisions; in the LPM trie, we added a global variable to explicitly track the depth traversed.

The results for the data structures were mixed: For the LPM trie, Freud produced an accurate performance annotation. For the hashmap, Freud mistook a correlation for a causation: when a test caused every packet to experience a collision, Freud concluded that runtime was determined by occupancy, as opposed to the number of collisions. We expect that this issue can be resolved at the cost of extra developer effort (to produce a smarter test suite).

For the real NFs, Freud could not produce meaningful performance annotations (despite our

Freud mode	Program	Accurate annotation?
Freud-vanilla	Synthetic stateless NF	Yes
	LPM trie	No
	Hashmap	No
	Real NFs	No
Freud-nf	Synthetic stateless NF	Yes
	LPM trie	Yes
	Hashmap	No
	Real NFs	No

Table 5.9: Summary of our experiments with Freud.

modifications to the NF source code). This is not surprising, given that Freud does not analyze the source code, hence is unable to track how a sequence of state-accessing calls affects runtime. For instance, in Maglev, known client packets that are destined to a now-stale backend-server undergo consistent hashing once again, to pick a new backend. Since Freud does not analyze the source code, it cannot track how this call sequence affects runtime, looking instead to express runtime as a function of individual variables—which does not work. We observed similar scenarios in the other NFs.

Conclusion: In its current form, Freud cannot produce accurate performance annotations for stateful NFs. To do so, it would need to track how a sequence of state-accessing calls affects performance. We think that that would necessarily require (a) some assumption about the structure of the code (akin to our clean state assumption), (b) a nuanced test suite for the NF's data structures to reveal which aspects of state affect performance (which is done, in our approach, with the manual extraction of LCVs during pre-analysis), and (c) leveraging call context. We think that adding these elements to Freud would bring it very close to LINX; we expect it would achieve similar accuracy but at the cost of its current generality.

5.8 Related Work

Performance analysis for SmartNIC-based NFs: Krude et al. [144] use SMT solvers to analyze NF code written for processor-based SmartNICs and provide lower bounds on throughput. Focussing solely on throughput lower bounds results in their approach being limited to analyzing worst-case latency, much like WCET. Clara [197] uses machine learning to analyze NF code written in C to identify "effective porting strategies" that result in low latency when the NF is ported to a SmartNIC. Unlike LINX that focusses on accurately predicting the NF latency, Clara focusses on identifying how the NF implementation can make best use of the SmartNIC hardware (e.g., accelerator usage, NF state placement strategies, etc).

Program analysis for NF code running on commodity hardware: Several instances of prior work have proposed using program analysis to help understand, debug, and verify the semantic behavior of software NFs [37, 38, 67, 138, 192, 218, 252, 255]. LINX builds upon the experience of all of this prior work, but analyzes NF performance.

NF performance monitoring and diagnosis: Several instances of prior work [86, 101, 167, 243] diagnose performance issues such as packet drops or low throughput in NF deployments. Such work is complementary to LINX since it helps diagnose performance issues once they occur in production, while LINX provides a summary of NF performance before the NF is deployed.

5.9 Conclusion

In this chapter, we described LINX, a tool that automatically extracts latency interfaces from NF implementations. and evaluated it on 12 NFs, including several used in production. Our results show that LINX is practical—the complexity of extracted interfaces is significantly lower than the NF implementation, their accuracy is high, and the time to extract them is reasonable. Finally, we show how NF developers and operators can use these interfaces today, to identify performance regressions, diagnose and fix performance bugs, and pick the NFs that are best suited to their hardware.

6 From Latency to Side-Effects: Automatically Reasoning About How Systems Code Uses the CPU Cache

Since a program's semantic interface describes not only its expected output(s) but also any related side effects (modifications to shared state that may lead to differences in externally observed behavior) [210], the program's latency interface must also describe its expected latency side effects in addition to the processing latency (described in Chapter 5).

Latency side effects arise due to shared micro-architectural state. Since all programs running on the same CPU core (e.g., caller and callee, application and operating system) share all core-local micro-architectural resources (e.g., data and instruction caches, TLB, branch predictor, etc.) calling into a piece of code has not only a direct impact on latency (via the execution latency of callee) but also an indirect cost that depends on how the callee perturbs shared micro-architectural state. This indirect cost is a frequently observed source of latency variability. For example, FlexSC [215] showed how a system call can take up to 3× longer depending on the invoking program's micro-architectural resource usage, while the invoking program may run up to 4× slower after the system call, depending on the system call's micro-architectural resource usage. Similarly, Cerebros [254] showed how microservices can spend up to 50% of their total cycles simply stalling on the instruction cache due to gRPC's large instruction footprint.

In this chapter, we focus on a dominant source of micro-architectural side effects, namely the CPU cache. Our goal is to enable developers to answer frequently asked questions about how a piece of systems code interacts with the cache, such as: *How does the code's cache usage vary with workload (e.g., as a function of the number of network connections)*? [24, 68, 85, 219, 226] What is the code's cache hit/miss profile? [45–47, 238, 257] *Which workloads make the working set exceed the cache size*? [145, 188]

To answer the above questions, developers require visibility into *what the code does* to the microarchitecture *as a function of the workload*, i.e., visibility into how the code processes an *abstract* workload. For instance, understanding the cache usage of a network stack requires understanding the layout of various `structs` and how these are accessed as the number of network connections changes.

Existing performance-analysis tools such as profilers and cycle-accurate simulators do not provide such visibility; they can draw conclusions about the *concrete* workloads with which the code is profiled or simulated, but not about *abstract* unseen workloads. Profilers [35, 153, 189, 230] treat the code as a black box and measure its performance for a given workload. Cycle-accurate simulators [28, 33] are similar: they provide greater visibility into micro-architectural events than profilers, but, once again, only for the given workload. So for instance, while they can tell developers which memory accesses result in a cache miss for the given workload, they cannot provide predictions about which workloads will thrash the cache and cause an unacceptable miss rate. As a result, developers are forced to manually reverse-engineer the answer to their key questions. This process is time-consuming and error-prone [115], particularly for code that the developers did not write themselves.

We present CFAR (Cache Footprint AnalyzeR), a tool that processes a piece P of systems code into answers to developers' questions about how that code uses the cache. CFAR's processing consists of two phases: In the former, CFAR takes as input the code and outputs an intermediate representation (a "distillate") that contains all the information on how the code accesses memory. In the latter, developers can write simple programs ("projectors") that use the distillate to compute answers to specific questions they have about P's cache usage. CFAR relies on a combination of static analysis, symbolic execution, and binary instrumentation to automatically extract distillates. We chose these particular program-analysis techniques because, despite their limitations (discussed in §6.3.3), they enable precisely the visibility we seek—reasoning about what the code does to the micro-architecture for abstract workloads.

The current CFAR prototype comes with three projectors that answer frequently asked questions about cache usage: (1) $\mathscr{P}_{\text{scale}}$ shows how the amount of data the code brings into the cache (i.e., the number of unique cache lines touched) varies with workload (e.g., with the number of active network connections). (2) $\mathscr{P}_{\text{h/m}}$ shows whether each memory access will hit or miss in the cache as a function of workload, and (3) $\mathscr{P}_{\text{crypt}}$ flags cryptographic code that branches or accesses memory addresses depending on secret inputs, thereby flagging potential branch and cache-based leakages. $\mathscr{P}_{\text{crypt}}$ in particular, demonstrates the flexibiilty of CFAR's two-phased process: since the distillate contains all information relevant to how the code accesses memory, developers can write projectors to analyze more than just performance properties. We envision developers contributing more such projectors, expanding CFAR's collection as needed to get the answers they seek.

In the rest of the chapter, we motivate CFAR through an example cache-usage question that existing tools cannot answer (§6.1), provide an overview of CFAR (§6.2), describe its design (§6.3), evaluate it experimentally (§6.4), discuss related work (§6.5), and conclude (§6.6).

6.1 Motivation

In this section, we give an example of the kind of questions that systems developers ask about their code's cache usage (§6.1.1), describe why existing tools cannot answer such questions (§6.1.2), and argue that answering them requires reasoning about the code (§6.1.3).

6.1.1 Example

Suppose Alice is building a simple in-memory key-value store that uses a hashtable (for storing the key-value pairs) and runs atop a user-space, kernel-bypass transport stack. Alice has modified an existing hashtable implementation to suit her needs and thus understands that part of the code well. At the same time, she is using an off-the-shelf transport stack [74, 137, 256], of which she understands little beyond the semantic interface it exposes.

In such a system, processing latency (and thus throughput) is typically determined by the number of LLC misses per request [152,219,258]; hence, to optimize her system's throughput, Alice needs to know how the different parts of her code use the cache and how they affect the LLC misses and working set as a function of workload. For example, if her system fails to reach the expected throughput due to persistent LLC misses, which is the predominant cause? that the hashtable code touches too many cache lines per put and/or get request? or that the transport stack's buffer-management code touches too many cache lines per connection [24]? In the former case, Alice should spend her time further optimizing the memory layout of the hashtable [45–47], whereas in the latter, she should port her code to alternative stacks with lower memory footprints [74, 219]. Finally, if both codebases were already highly optimized, she should avoid wasting time on code optimizations and replicate her service across machines [15].

6.1.2 Existing Tools Are Insufficient

Answering this question today is inefficient: Alice would run her system with many workloads, use a profiler [35, 153, 189, 230] or a simulator [28, 33] to measure micro-architectural events and reverse-engineer the predominant cause of LLC misses. In particular, she would try to identify the properties of workloads that led to higher throughput: were they those that led to fewer put/get accesses per request? or those that led to fewer concurrent connections? This is similar to what Google developers do to answer such questions, e.g., they run their code for multiple workloads, use profilers to count the total number of unique cache lines touched, and extrapolate how workload affects their code's cache footprint [18].

This profiling+extrapolation process not only is inefficient but can be incomplete, especially when the analyzed system includes third-party code. Alice (who knows little of the transport stack's implementation) may not even think to run workloads that lead to different numbers of concurrent connections. In general, performance profiling suffers from the "large input problem" [157, 180], i.e., the fact that unexpected performance behaviour often manifests only when input size (e.g., the number of concurrent connections) exceeds some limit that may seem arbitrary to those who are not intimately familiar with the code. So, designing a test suite that completely covers a system's performance behaviors is hard, and developers don't even have well-defined coverage metrics. For example, line coverage is used as a proxy for coverage of semantic behaviors; performance profiling does not even benefit from such an approximate metric.

As a result, developers often fail to identify workload properties that significantly impact cache usage, causing performance cliffs to manifest in production. For instance, initial work on predicting the working set of network functions ignored the impact of different packet sizes [68], while a recent study of Linux's system call performance showed how a newly introduced configuration parameter can destroy spatial locality and lead to increased LLC misses [200]. In practice, to avoid suffering from unexpected throughput degradation due to incomplete performance profiling, Alice would over-approximate her system's cache usage and overprovision resources for her service, for instance by replicating more than necessary. Recent work from Azure showed that most customer VMs over-approximate their memory footprints by $30-50\%$ [92].

6.1.3 An Interface for Cache Usage?

Ultimately, Alice needs to understand how her code uses the cache in order to answer her question. It is first the code (e.g., the layout of different `structs` and how they are accessed), and only then the cache algorithm that determines how many cache lines are touched per unique key and per connection. Hence, tools that treat the code as a black box are fundamentally inefficient in answering questions about how the code uses the cache—or any component of the micro-architecture.

A natural question is whether the general-case latency interface for the program can answer the above questions. The answer is that while the definition of general-case interfaces (Chapter 4) certainly allow for the possibility, the interfaces extracted by LINX cannot. The former is possible because one can think of each question about cache usage as defining a new latency metric (e.g., total number of unique cache lines touched by the program, number of cache lines unique to the connection, etc.), and so a general-case interface that returns that particular metric is precisely what Alice requires. However, LINX-extracted interfaces are not flexible enough to answer this question. This is because LINX, which was designed for NFs that run pinned to CPU cores assumes that the NF always executes in a "steady" micro-architectural state, i.e., processing a packet always leaves the cache in an equivalent state. So, LINX-extracted interfaces can reason about latency without considering the NF's cache usage and thus discard the information required to answer Alice's questions.

The more relevant question is: How do we extract an interface that describes the program's cache usage in terms of the metric that the developer is interested in? Is there an abstraction that retains enough information to answer arbitrary questions about cache usage, or must the program be re-analyzed each time? In the next section, we describe our abstraction (which we call the CFAR distillate) and show how it is flexible enough to answer arbitrary questions about cache usage.

6.2 The CFAR Approach

CFAR analyzes a piece P of systems code (such as a system call in an OS kernel) and produces an intermediate representation that captures rich information on how that code accesses memory. On top of this IR, developers can write simple programs that compute answers to questions they have about how P uses the cache. CFAR provides an engine that produces this intermediate representation and a few example programs that answer questions like how does P's cache footprint scale across a range of inputs, what is the cache hit/miss profile of P, or does P access the cache in a way that depends on secret inputs. We envision developers contributing more such programs, to expand CFAR's collection over time. Eventually, developers will just use whatever ships with CFAR, extending it only when they cannot get the answer they seek.

P can be any well defined part of a system that can be invoked individually, such as a system call in an OS kernel or a function in a library, or even a standalone program. Fig. 6.1 illustrates CFAR's workflow.

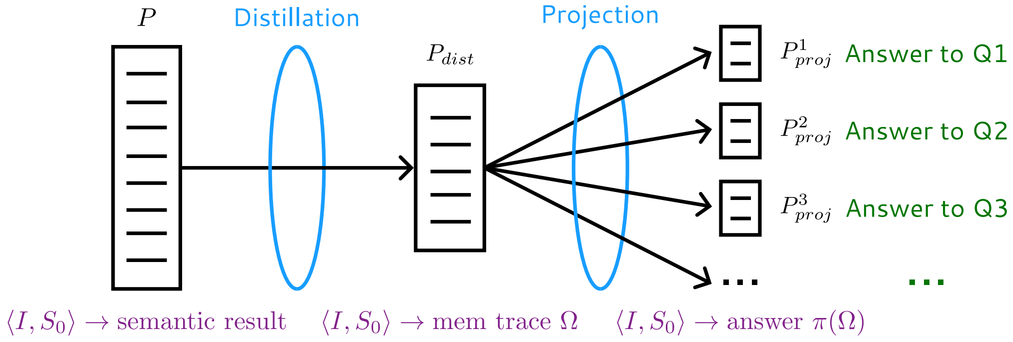

P
Distillation
P_{dist}
Projection
P^1_{proj} Answer to Q1
P^2_{proj} Answer to Q2
P^3_{proj} Answer to Q3
...
...
$\langle I, S_0 \rangle \rightarrow$ semantic result
$\langle I, S_0 \rangle \rightarrow$ mem trace Ω
$\langle I, S_0 \rangle \rightarrow$ answer $\pi(\Omega)$

Figure 6.1: The CFAR workflow.

CFAR's processing of P has two phases: The first phase abstracts the code of P into a P_{dist} (a *distillate*) that contains all information relevant to how P accesses memory and discards everything else. The second phase abstracts P_{dist} into P_{proj} (a *projection*) that contains only information relative to the developer's specific question and nothing else. P_{dist} may contain too much low-level detail for human consumption, so we consider it an intermediate representation. P_{proj}, which we call a projection, answers a developer question, and so must be human-readable.

Splitting the processing into two phases offers flexibility and customizability. For any given P, there exists a single P_{dist}, uniquely determined by P, but there can be as many P_{proj} as there are questions about P's cache usage. For example, one projection could compute the number of cache lines touched by P, while another projection could determine whether P's cache access pattern is influenced by secrets, and so on. Most importantly, developers themselves can write programs (*projectors*) that generate projections, and CFAR essentially offers a framework for programmatically answering any number of cache-usage questions. This enables CFAR to be expanded with little effort, becoming more useful over time.

CFAR's approach is similar to that of LINX in two notable ways.
First, both analyze the input program separately from the environment in which it runs. For any given P, they first extract a representation that is uniquely determined by P—general-case interfaces and distillates, respectively—and then analyze how that representation interacts with its environment (e.g., different underlying hardware, different cache architectures, etc.). This split analysis ensures flexibility and allows the general-case interface to be tailored to arbitrary deployment environments and the distillate to answer arbitrary questions about the program's cache usage. Second, both represent the property of interest—processing latency and answers

about cache usage, respectively—as programs that accept the same input(s) as P. Like latency interfaces, both P_{dist} and P_{proj} are programs. P_{proj} in particular, can be thought of as being an interface that describes the latency of P in terms of the metric defined by the developer's specific question.

```c
int sys_create(int fd, fn_t fn, uint64_t type, uint64_t value, uint64_t omode) {

    struct file *file;
    if (type == FD_NONE)
        return -EINVAL;
    if (!is_fd_valid(fd))
        return -EBADF;
    if (get_fd(current, fd) != 0)
        return -EINVAL;
    if (!is_fn_valid(fn))
        return -EINVAL;
    file = get_file(fn);
    if (file->refcnt != 0)
        return -EINVAL;

    file->type = type;
    file->value = value;
    file->omode = omode;
    file->refcnt = file->offset = 0;
    set_fd(current, fd, fn);
    return 0;
}
```

```python
def sys_create_dcache(fd, fn, type, value, omode):
    # State: pid, proc_table, filetable

    if type == FD_NONE: #6 accesses
      return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]

    if not(fd >=0 and fd < NOFILE): #6 accesses
        return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]

    if [proc_table+256*pid+64+8*fd]: #7 accesses
        return [(w,rsp-8),(w,rsp-16),..,(r,proc_table+256*pid+64+8*fd),..,(r,rsp-8)]

    if not(fn >=0 and fn < NOFILE): #7 accesses
        return [(w,rsp-8),(w,rsp-16),..,(r,proc_table+256*pid+64+8*fd),..,(r,rsp-8)]

    if [filetable+40*fn+8]: #9 accesses
        return [(w,rsp-8),(w,rsp-16),..,(r,proc_table+256*pid+64+8*fd)..,(r,filetable+40*fn+8),..,(r,rsp-8)]

    # Succesful create. 17 accesses
    return [(w,rsp-8),(w,rsp-16),..,(r,proc_table+256*pid+64+8*fd),..,(r,filetable+40*fn+8),(w,filetable+40*fn),(w,filetable+40*fn+16),..,(w,proc_table+256*pid+64+8*fd),..,(r,rsp-8)]
```

Figure 6.2: Example P on left (Hyperkernel's `sys_create` system call that creates a new file) and the corresponding P_{dist}^{data} distillate.

.

```python
def sys_create_icache(fd, fn, type, value, omode):
    # State: pid, proctable, filetable
    # sys_create abbreviated as s

    if type == FD_NONE: #10 insns
      return [(r,s),..,(r,s+168),..,(r,s+176)]

    #Error paths elided for representation
    ......

    # Succesful create. 45 insns
    return [(r,s),(r,s+8),..,(r,s+160),(r,s+168),(r,s+176)]
```

Figure 6.3: P_{dist}^{instr} distillate for `sys_create`.

6.2.1 Phase 1: Distillation

For a program (or function, or method) P that takes input I and whose state is S_0 at the time of invocation, the distillate P_{dist} is another (simpler) program that also takes input I but returns an ordered sequence Ω of P's memory accesses. Ω is expressed as a function of I and S_0 (where S_0 is the value of P's memory objects in the heap and the stack up to `%esp`). Accessing data vs. instructions exhibits distinct patterns so we distinguish a data-accesses distillate P_{dist}^{data} and an instruction-accesses distillate P_{dist}^{instr}. The former describes the sequence of data-memory accesses that would be observed if executing P with input I starting from state S_0, while P_{dist}^{instr} describes the instruction-memory accesses. A distillate is essentially a program that computes function $<I, S_0> \rightarrow \Omega$.

Each entry in Ω is a memory access $<type, addr>$ where *type* can be either read (`r`) or write (`w`), and *addr* is a memory address. Fig. 6.2 and Fig. 6.3 give two examples, where P is the `sys_create` system call in the Hyperkernel [171]. For a data-accesses distillate (Fig. 6.2), addresses are described in terms of standard state components, like the stack pointer `rsp`, as well as state components specific to P: line 11 describes accesses that are a function of `proc_table`, `pid`, and `fd`, caused by the `&proc_table[pid]->ofile[fd]` lookup done by `sys_create`. If an address is independent of I and S_0 (not shown in the example), such as that of a struct allocated by P in the heap and then freed before returning, the corresponding entry in Ω has a named constant, e.g., `mallocRetVal@file.c:342`. For an instruction-accesses distillate (Fig. 6.3), the addresses are given as aligned offsets relative to the memory address of the first instruction in P. In this example, the compiler inlines all helper functions, hence there is only one base address `s`.

The P_{dist} distillate is a precise and complete representation of P's memory usage. It is *precise* because it correctly predicts the actual memory usage of P during an execution. The symbolic expressions for data- and instruction-memory accesses as a function of I and S_0 are precise by construction, and therefore correct for any concrete instantiation of I and S_0. The distillate is *complete* in that it contains all information on P's memory usage that can be found in P. No matter what the concrete values of I and S_0, how the address space is randomized [2], or where in memory the code is loaded, a distillate will always be able to produce the exact sequence of memory accesses that P makes when executing from S_0 with input I.

6.2.2 Phase 2: Projection

A projection P_{proj} is a program similar to P_{dist} but that, instead of returning Ω, returns a function $\pi(\Omega)$. For example, a simple projection might replace Ω in P_{dist} with $|\Omega|$ to compute the number of memory accesses performed by P. Another projection might replace Ω with $|\{\lambda(r) = r/64 : r \in \Omega\}|$ to compute the number of unique 64-byte cache lines accessed by P. Where a distillate computes function $<I, S_0> \rightarrow \Omega$, a projection computes $<I, S_0> \rightarrow \pi(\Omega)$

The π function can generalize to take additional input, beyond just Ω. For example, a projection can use a $\pi(\Omega, \$M)$ function to combine the access sequence Ω with a cache model $\$M$ and produce a program P_{proj} that computes the number of hits and misses incurred in the L1 data-cache by code P.

In conclusion, CFAR's two-phase approach—P_{dist} as an intermediate representation and projections P_{proj} layered on top of it— offers the flexibility to answer any number of questions about P's cache usage. The key is that the $P \to P_{dist}$ abstraction captures correctly and fully all of P's memory access information, and offers an intermediate representation that can easily be projected into answers to developers' questions.

6.3 CFAR Design

CFAR abstracts $P \to P_{dist} \to P_{proj}$. In the first phase—distillation—CFAR processes the code P into a distillate P_{dist} in four steps (shown in Fig. 6.4): it (i) enumerates all feasible executions paths in P; then (ii) obtains a binary execution trace for each such path; then (iii) based on the two outputs, it prepares an execution tree for the distillate; and lastly (iv) optimizes this tree and produces P_{dist}. This distillate is a program that describes symbolically P's memory trace Ω for every possible input to P. The second phase—projection—transforms P_{dist} into P_{proj} that directly answers the developer's specific cache-usage question. It does so by summarizing Ω and possibly also transforming the control flow of the P_{dist} program, producing a much simpler program that is better able to answer the posed question.

The four steps in CFAR's distillation phase are similar to those in the LINX back-end (Fig. 5.2), and our implementation reused large parts of the LINX codebase. However, there are two important differences. First, as we will show, extracting the distillate requires extracting additional information during the path enumeration stage. Second, since CFAR is not designed for NFs in particular, and aims to be fully automatic, it cannot rely on manual analysis to extract loop summaries using LCVs. To overcome this, CFAR performs best-effort loop summarization in the code synthesis stage.

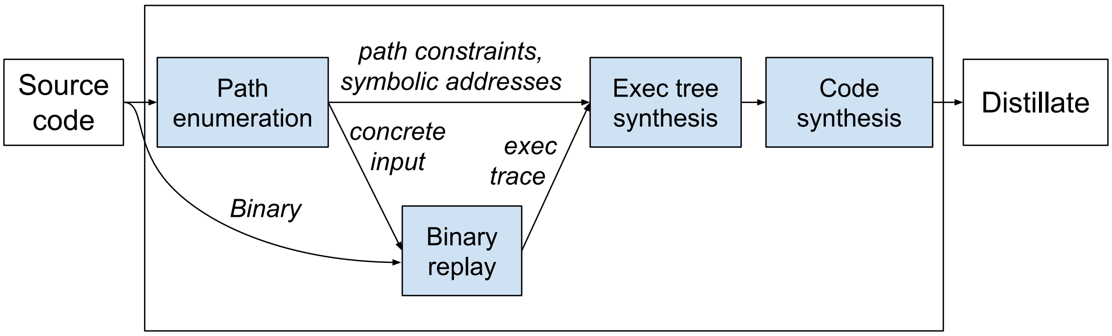


Figure 6.4: The four components of CFAR's analysis.

6.3.1 Distilling P into the P_{dist} representation

A distillate P_{dist} is a program that returns, for every relevant case, the corresponding memory trace Ω of P. The control flow of P_{dist} reflects the different cases that would influence Ω, and so

P_{dist} will naturally have P's control flow. The examples in Fig. 6.2 and Fig. 6.3 show what the output of this phase looks like.

Step 1: Enumerating all paths in P

To obtain all the paths in P, CFAR uses *exhaustive* symbolic execution to enumerate them. Symbolic execution [36, 98, 140, 206] is a program analysis technique that automatically traverses the feasible execution paths of a body of code, enabling a comprehensive analysis of its control flow. The technique is powerful, but also faces challenges related to loops and pointers, which we discuss in §6.3.3. We use an exhaustive form of this technique, which yields *all* feasible paths in P.

For each enumerated path Π, CFAR saves four key pieces of information: (1) the precise path constraint C_Π that uniquely defines this path, i.e., C_Π is the conjunction of the outcomes of evaluating each `if` predicate along Π; (2) a concrete input I_Π that exercises this path, obtained by feeding C_Π to an SMT constraint solver and asking for a satisfying assignment to the input variables in C_Π; (3) for each data/instruction memory location accessed, the symbolic expression corresponding to the address, as a function of the inputs and/or P state; (4) for each memory operation, a corresponding `filename:linenum` identifier, to be used later. The sequence of these symbolic expressions is ω_Π.

Step 2: Obtaining the binary execution trace

What actually executes on the hardware is not the source code or the IR. Compiler optimizations, such as link-time optimization, cause the executing machine code to not directly correspond to what is in the IR. Furthermore, many IRs are Static Single Assignment (SSA), in which each variable is assigned exactly once. This makes the data flow and dependencies among variables more explicit and easier for the compiler to analyze, but it also implies an infinite register file. However, processors do not have infinite register files, so during an actual execution register values often need to be spilled to the stack. But P in the IR form does not push or pop the stack, so the corresponding memory accesses will not appear in ω_Π.

Therefore, CFAR replays an *instrumented* version of the P binary for each I_Π, to obtain the corresponding concrete execution trace X_Π. For each machine instruction executed in X_Π, CFAR saves: (1) the program counter; (2) the instruction opcode, such as `push` or `pop`; (3) the concrete memory addresses accessed; (4) the corresponding `filename:linenum` debug information inserted into the binary by the instrumentation.

We deliberately split the analysis into a source-based and a binary-based step. It is easier to extract symbolic expressions for memory operations by analyzing the source or the IR. On the other hand, analyzing the binary enables CFAR to be fully precise with respect to compiler optimizations and which instructions lead to memory accesses and do not merely manipulate CPU registers. In theory, these two steps could be combined into a single one by directly symbolically executing the binary. To answer with certainty, one would need to assess how CFAR is affected by the loss of type information when going from source code to binaries, and find a good way to deal with the lack of debug symbols in production binaries.

Step 3: Synthesizing P_{dist}'s execution tree

In this step, CFAR combines the information extracted in the previous two steps. For each path Π in P, it combines the symbolic memory trace ω_Π with the corresponding binary execution trace X_Π. To produce a *data-memory access trace*, CFAR takes the sequence of concrete addresses from X_Π and replaces (using debug information) all input- and state-dependent accesses with the corresponding symbolic expressions from ω_Π, thus producing Ω_Π^{data}. To produce an *instruction-memory trace*, CFAR uses the program counter values and the call stack in X_Π to compute the symbolic offset of each instruction from the start of P (e.g., its entry point, if it is a function or a system call) and produce Ω_Π^{instr}. The call stack gives CFAR information on which function the instruction belongs to, so that it can compute the function-specific offset.

Next, CFAR assembles an execution tree out of the path constraints C_Π. It arranges all the paths into a tree based on their common prefixes; for every path Π there exists a path from tree to leaf in the tree, and vice versa. Each internal node n in the tree contains the predicate corresponding to the original branch in P. The conjunction of the predicates for all internal nodes along a root-to-leaf path forms the corresponding path constraint C_Π.

Step 4: Producing the P_{dist} distillate

The final step consists of best-effort summarization of loop-related memory access patterns and other, more minor, improvements for human readability of P_{dist}. Symbolic execution, by default, unrolls loops and thus produces a different execution path for each loop iteration. This leads to bloated distillates that contain redundant information and are hard to read, particularly if the access pattern of the code does not change across loop iterations.

Automatically summarizing loops in general is undecidable [93], but fortunately studies have shown that there exist four common categories of loops that relate to data locality issues in systems code [139]. Therefore, CFAR contains loop-summary templates for these four categories of loops—two that traverse array-like data structures, and two that traverse pointer-chasing data structures (e.g., linked lists, trees). All four categories of loops require the loop body to not branch on the precise value of the iteration counter, and for the loop to have a maximum of two termination predicates, one in the loop definition and at most one `break` in the body. In case CFAR is unable to infer that this holds, the distillate presents the unrolled loop.

Finally, CFAR transforms the optimized tree into a program that represents P_{dist}. This program takes the same input as P. Every internal tree node n leads to an `if` statement in the program, branching on the predicate contained in that node. Each path through the program/tree ends with a `return` of the corresponding Ω_{Π}—depending on the memory-type of the distillate, this is either Ω_{Π}^{instr} or Ω_{Π}^{data}.

Our CFAR prototype uses Python to represent distillates, because it is one of the most widely used languages [179] and has an easy-to-understand syntax.

Fig. 6.5 illustrates the loop-summarization optimization with a snippet for `memcmp`'s P_{dist}^{data} distillate. The corresponding loop belongs to the first category mentioned above. Our CFAR prototype uses first-order logic to summarize loops with primitives from Z3's Python API [250]. The predicate that starts on line 3 identifies the smallest index `i` at which the two strings differ. The distillate then states (starting at line 8) that the memory accessed corresponds to every element of the two arrays up to `i`.

```
1  def memcmp_dcache(s1,s2,len):
2
3      if Exists(i,And(0<=i<len,
4                      [s1+i]!=[s2+i],
5                      ForAll(j, Implies(0<=j<i),
6                          [s1+j]==[s2+j])))):
7
8        return ForAll(k, Implies(0<=k<=i),
9                          [(r,s1+k),(r,s2+k)])
10
11     return ForAll(k, Implies(0<=k<=len),
12                         [(r,s1+k),(r,s2+k)])
```

Figure 6.5: P_{dist}^{data} for `memcmp`.

6.3.2 Projecting the P_{dist} IR into P_{proj} Answers

The distillate produced by the previous phase is a precise and complete description of P's memory-access behavior. While the answers to developers' cache-usage questions can be found in the distillate, they are buried in details that may not be relevant to the specific question being asked. The projection phase turns distillates into actual answers.

Defining Projectors

With CFAR, developers can define projectors, programs that take as input Ω (as a Python list [195] in our prototype) and compute specific properties related to cache usage. Computing these properties is equivalent to computing function $\pi(\Omega)$. CFAR then puts together the specific properties returned by the distillate for each path, into a program that we call a projection (P_{proj}). This program has the same control flow as P_{dist}, but returns $\pi(\Omega)$ instead of Ω. As described in §6.2.2, a simple projector might compute the number of memory accesses or the number of unique cache lines accessed by P.

A projector is free to take in additional parameters, not just P_{dist}. For example, it can often be useful to pass in a cache model, in order to answer questions like what is the number of cache hits and misses incurred for each class of inputs, or when does P's working set fit in L1 vs. not.

Advanced developers can produce arbitrarily sophisticated P_{proj}. A program-specific projector could focus on just a subset of the input classes by conditioning $\pi(\Omega)$ on program-specific predicates. For example, a version of `sys_create_dcache` (Fig. 6.2) focused on only successful `sys_create` calls would have a single `if` statement with the conjunctive negation of the first 5 predicates in Fig. 6.2, followed by line 20.

Program-specific projectors help developers reason succinctly about groups of inputs to stateful code with loops, by specifying aggregate abstract state that the inputs are likely to encounter. For instance, consider a hash table implemented using linked lists for which the number of memory accesses per call is a linear function of the number of hash collisions, which itself is expressed as a first-order logic predicate. A precise program-independent sum across a group of N inputs would have to be expressed as a sum of N such predicates, which is hard for both humans and solvers to reason about. Instead, by constraining the number of collisions to particular values, developers can focus on how the footprint scales as a function of N. Such constraining is reasonable because developers care about different properties for individual inputs vs groups of inputs. For the former, they are interested in how many collisions the code can suffer, which they reason about using individual predicates. For the latter, they are typically happy to reason about aggregate statistics that the group of inputs encounter [123].

CFAR-provided projectors

CFAR comes with three example projectors: (1) $\mathscr{P}_{\text{scale}}$ produces a projection P_{proj}^{scale} that shows how the cache footprint scales across an entire range of previously unseen inputs (e.g., how it varies with the number of active network connections). (2) $\mathscr{P}_{\text{h/m}}$ produces a projection $P_{proj}^{h/m}$ that shows the cache hit and miss profiles per class of input as opposed to per specific input; and (3) $\mathscr{P}_{\text{crypt}}$ produces a projection P_{proj}^{crypto} that flags cryptographic code that accesses the cache in a way that depends on secret inputs; this can be used to flag potential vulnerabilities.

To compute how the footprint scales for each input to P, $\mathscr{P}_{\text{scale}}$ first determines the number of symbolic addresses in the list that would differ if only the value of that input changed. It then checks the alignment of those bytes using a solver query, to check the number of unique cache lines touched by these addresses. It presents the results to the user as formulae. For instance, the output of $\mathscr{P}_{\text{scale}}$ for `sys_create` is: `8*fd + 32*fn` which is precisely what Alice wanted to know for keys and connections in §6.1. Developers can use this information to estimate when their working set overflows the cache.

$\mathscr{P}_{\text{h/m}}$ allows developers to go a step further and reason about the possible cache misses that their code might incur. $\mathscr{P}_{\text{h/m}}$ takes in three inputs: trace of memory accesses Ω, a cache model, and an input set size. $\mathscr{P}_{\text{h/m}}$'s default cache model is a 3-level inclusive cache with a next-line prefetcher and the sizes and set associativity of each level being configurable parameters. The input set size refers to the number of unique inputs (e.g., number of active connections) that the program expects to receive and is used to warm up the cache. $P_{proj}^{h/m}$ passes the input set size to P_{proj}^{scale}, obtains the total number of cache lines it must account for, and inserts a corresponding number of symbolic addresses into the cache in random order. Finally, it runs the memory trace with symbols corresponding to a random input from the input set and measures the number of hits and misses. To ensure that the effects of the random selection are properly accounted for, $P_{proj}^{h/m}$ repeats the last step multiple times until the set of possible misses stabilizes.

$\mathscr{P}_{\mathrm{h/m}}$ can be thought of as a symbolic, trace-based cache simulator that allows developers to study cache events just like traditional trace-based memory simulation supported by all cycle-accurate simulators but without having to write concrete benchmarks to initialize cache state. The only difference is that $\mathscr{P}_{\mathrm{h/m}}$ is forced to deal with symbolic addresses, and so cannot accurately compute set-associativity conflicts in the cache. Instead, $\mathscr{P}_{\mathrm{h/m}}$ allocates unconstrained memory addresses to a random set in the cache, and ensures that addresses computed using that symbol as a base are assigned according to their least-significant bits. In our evaluation, we show how this approximation works reasonably well in comparison to real hardware.

$\mathscr{P}_{\mathrm{crypt}}$ produces a P_{proj}^{crypto} that requires the developer to define the program inputs that are secrets, and then uses a Z3 [64] solver query to determine whether any `if` conditions (program branches) or memory addresses in the distillate are functions of secrets. If this is the case, it returns debug information `filename:linenum` corresponding to the branch/memory-access as well as the path constraints under which it occurs. Note, $\mathscr{P}_{\mathrm{crypt}}$ does not account for all cache-based leakages, it only accounts for leaks due to secret-dependent branches or memory accesses.

In our experimental evaluation (§6.4) we use these three projectors to evaluate CFAR.

6.3.3 Limitations and Assumptions

Symbolic execution: CFAR's reliance on symbolic execution (SE) makes it subject to SE's own limitations. Depending on which SE engine is used, certain kinds of loops, or symbolic pointers, or multi-threading could prevent obtaining all execution paths [30]. However, there is active research on this topic, and recent SE engines have brought various enhancements that overcome these challenges, such as state merging [146], loop-extended symbolic execution [203], loop summaries [99, 247], loop invariants [127], and symbolic abstract transformers [143].

A CFAR prototype will ultimately be as powerful as its underlying SE engine. Since our prototype relies on KLEE, code whose loops do not have statically computable bounds, or that is multi-threaded, or that has arbitrary symbolic pointers is not an ideal match because path exploration may take too long.

Input constraints: One way to use CFAR on code that cannot be exhaustively symbexed is by constraining the input space. A reasonable approach is to constrain it to inputs that are meant to trigger the "fast path" through the code, since that is a common object of performance analysis. For instance, if the target code is an IP forwarding function, it is reasonable to constrain the input space to packets without IP options; this reduces dramatically the size of the execution tree (because it eliminates the part of the code that loops through the variable-length IP options), and it still yields practically useful results (because performance-sensitive traffic does not typically carry IP options). Of course, this approach has a drawback: a distillate computed this way is complete only for inputs that belong to the selected input space. Nevertheless, we followed this approach to use CFAR on TCP stacks, and our evaluation demonstrates that the results can be useful despite the constrained input space.

Time Limit: Another way to use CFAR on code that cannot be exhaustively symbexed is to allow the developer to specify a time limit for the analysis; when the time limit expires, CFAR outputs a partial distillate that iss precise but incomplete: it returns the exact sequence of memory accesses performed by the code along the explored execution paths. Our CFAR prototype offers this capability, however, we did not use it for any of the results shown in our evaluation.

Binary Instrumentation: CFAR employs binary instrumentation to obtain an execution trace. Unfortunately, such instrumentation can only reveal instructions that the processor finished executing (retired); it does not reveal instructions that were executed as a result of incorrect speculation (e.g., a mispredicted branch). Such instructions nevertheless could impact the cache and, since CFAR does not see them, the answers computed by projectors may not be fully accurate. We are not aware of any tool that can precisely report such mis-speculated instructions during an execution; existing work that does so relies on micro-architectural simulators and models [110, 232, 248]. Future work may however remove this limitation.

Preemption: In a similar vein, we assume that P is small enough to not have its execution interrupted by preemption. In the absence of this assumption, the distillate produced by CFAR (i.e., P_{dist}) is still correct, but the projection result might not be, because it could be missing third-party cache accesses that occurred during the preemption. In other words, in the presence of preemption, when a projector looks at the symbolic memory trace, it may not get a fully accurate picture of all cache accesses.

6.4 Evaluation

In this section, we evaluate the CFAR prototype by answering two main questions:

- **Does it work?** We show that CFAR extracts 100%-accurate data- and instruction-cache distillates, and that this extraction completes in minutes for various kinds of systems code (§6.4.1).

- **Is it useful** to system developers? We describe use cases that demonstrate how CFAR provides developers with visibility into cache usage in a way that profilers and simulators cannot (§6.4.2).

Evaluated programs. We used CFAR to analyze: the fast path of the transport layer of 3 TCP stacks—the Linux stack (v6.0)[1], a TCP stack used by recently proposed kernel-bypass OSes [24], and the lwIP TCP stack for embedded systems [74]—as well as 2 hash table implementations [192, 251], all 51 of the Hyperkernel's system calls [171], and 7 algorithms from OpenSSL 3.0.0 [181]. All of this code is publicly available and unless specified, we analyze the latest stable version. Note, the kernel-bypass stack uses the lwIP stack as a starting point, but heavily modifies the internal datastructures and timer management.

Put together, these programs challenge CFAR's automated program analysis approach to different extents. At one end of the spectrum are the Hyperkernel and OpenSSL, both of which are amenable to automated program analysis. The hash table implementations occupy the middle of the spectrum, since they are both amenable to *manual* (not automated) program analysis. Finally, at the other end are the three transport layer implementations, which were not written to be amenable to any form of program analysis.

Despite these challenges, we demonstrate that CFAR can provide actionable information about cache usage for each of these programs, by constraining the input space (as discussed in §6.3.3). To analyze the transport layer of the three TCP stacks, we constrained CFAR to only explore execution paths corresponding to packets processed in the TCP fast path, i.e., packets that belong to an established TCP connection, are received in order, and do not suffer hash collisions with packets from other connections. We picked this particular packet class because it represents a large fraction of packets processed by TCP stack and is the path for which performance matters the most; so, even though the resulting distillate is not complete, we think that it is still useful to developers building atop the TCP stack. To analyze the hash tables, we had to fix the maximum capacity of the table to a concrete value (we picked 65536)—so, our conclusions about this code hold only for this maximum capacity. Given that the implementations take in the capacity as a configurable parameter, we are certain that the memory access pattern is independent of the capacity; we just cannot prove it using symbolic execution.

Setup. We ran all our analysis on an Intel Xeon E5-2690 v2 CPU, clocked at 3.30GHz and provisioned with 25.6MB of LLC and 252GB of DRAM.

6.4.1 Does CFAR Work?

There are two key aspects to determining whether a tool like CFAR works and is practical: does it obtain an accurate intermediate representation of performance out of the code, and is the time to do so reasonable.

[1] The Linux TCP fast path was analyzed after the thesis was defended

Accuracy of Extracted distillates

To measure the accuracy of our prototype's distillates, we randomly picked 50% of the execution paths of each program, constructed inputs that exercised each path, counted the number of instructions and memory accesses executed while running each program with each input, and compared this number to the one predicted by the program's d-cache and i-cache distillates. In particular, the number of instructions counted during execution should be equal to the number of memory accesses predicted by the i-cache distillate (for the given input), while the number of memory accesses counted during execution should be equal to the number of memory accesses predicted by the d-cache distillate.

The error was always **zero**, across programs and inputs: CFAR's distillates correctly predict every single instruction and memory access executed by the code. This is not surprising: CFAR does not rely on models to predict the sequence of instructions and memory accesses, it measures them by replaying the binary. Since CFAR does not modify the binary in any way, the value measured during replay is identical to the one measured in production.

Time to Extract Distillates

Table 6.1 lists how long our prototype takes to extract distillates: for all programs, the analysis completes within 30 mins. The programs that take longest (and the only ones that take more than 15 min) are the Vigor hash table and the echde key-generation algorithm in OpenSSL; this is because in both, symbolic execution needed to unroll long loops that iterate over the data structure and compute co-prime numbers respectively. For all programs, the binary replay, execution tree synthesis, and code synthesis take approximately 2-3 mins in total; as expected, the dominant component—and the one that varies across programs—is symbolic execution.

Program	**Extraction time (mins)**
Linux TCP ingress	11
Linux TCP egress	14
Kernel-bypass TCP ingress	5
Kernel-bypass TCP egress	7
lwIP TCP ingress	4
lwIP TCP egress	5
Hyperkernel syscalls (51 total)	Avg: 4, Max: 7
OpenSSL primitives (7 total)	Avg: 9, Max: 22
Vigor hash table	28
Klint hash table	12

Table 6.1: Time taken by CFAR to extract distillates.

6.4.2 Is CFAR useful for system developers?

We demonstrate usefulness by presenting four use cases of CFAR answering important questions that developers cannot readily answer with the state of the art: How does the code's working set change with workload? How does the layout of data structures I call interact with my code? Does my code lead to inefficient memory access patterns ? Can I prove/disprove the absence of secret-dependent memory accesses?

How Does the Working Set Change with Workload?

We used the $\mathscr{P}_{\text{scale}}$ and $\mathscr{P}_{\text{h/m}}$ projectors to analyze the cache usage of the fast path of the transport layer of the 3 TCP stacks. This is the kind of analysis that Alice would do to (partly) answer her question in §6.1: is the predominant cause of persistent LLC misses the TCP stack?

First, we used $\mathscr{P}_{\text{scale}}$ to predict the number of unique cache lines touched by the TCP fast path assuming symbolic packet contents. The answer was 4,5, and 19 unique cache lines for the lwIP, kernel-bypass and Linux stack, respectively. Note, this is not information that one can glean by merely observing the size of the connection-specific `struct`. For instance, in Linux this `struct` (`tcp_sock`) occupies 44 cache lines in total, but only 19 of them are accessed on the fast path.

We then passed this information to $\mathscr{P}_{\text{h/m}}$ and asked it to predict when incoming packets were likely to suffer persistent cache misses due to the working set overflowing the LLC (25.6MB on our machine). The answer was that this would occur at approximately 91k, 76k and 18k concurrent connections for the lwIP, kernel-bypass and Linux stack, respectively. The slight differences in these predictions as compared to simple capacity-based calculations (e.g., 25.6M/(64*4) = 100k connections for lwIP) is due to $\mathscr{P}_{\text{h/m}}$ taking into account conflict misses in addition to capacity misses.

To verify these predictions, we ran a set of experiments where the transport layer receives and sends packets from/to a fixed set of established connections, and we varied the number of connections across experiments. To isolate just the transport layer (which is the code we analyzed), we wrote simple shims for the application and IP layers ourselves. In each experiment, we measured the average latency incurred by packets within the transport layer.

Fig. 6.6 plots latency as a function of the number of connections. For each of the three stacks, we see a clear shift around the number of connections predicted by $\mathscr{P}_{\text{h/m}}$. For instance, the latency for the Linux stack increases by only 45ns from 1k to 16k connections, but increases by 183ns from 16k to 32k connections. Likewise—although less visible in the graph due to Linux's dominant latency—the latency for the lwIP stack increases by only 13ns from 1k to 86k connections, it increases by 50ns from 86k to 125k connections. Note, the shift does not occur exactly at the predicted number of connections, but very close to it; compared to the predicted values of 18,76k, and 91k, we observed the shifts at 16k, 72k, and 86k, respectively. This is expected, since cache-mapping policies are proprietary and $\mathscr{P}_{\text{h/m}}$ only approximates them.

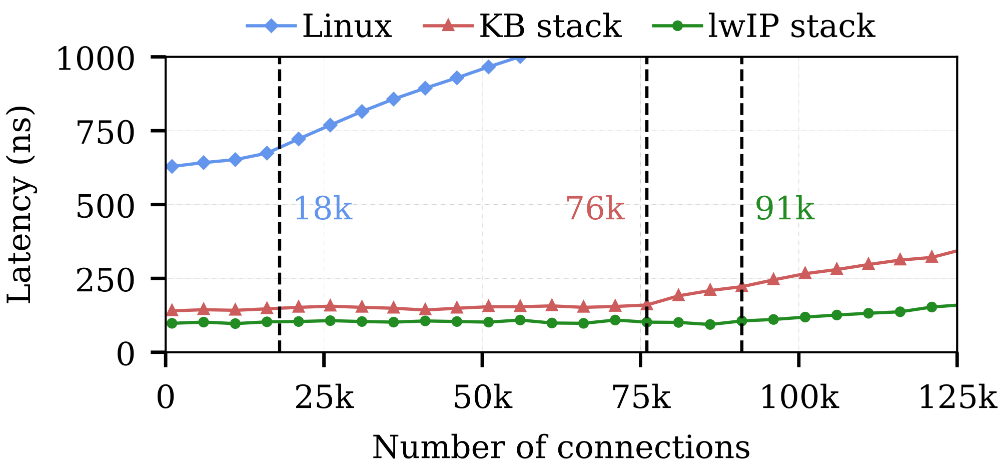


Figure 6.6: Measured latency for TCP packet processing as a function of the number of connections. 18k, 76k and 91k are the number of connections at which CFAR predicts persistent LLC misses for Linux, the kernel-bypass stack and the lwIP stack, respectively.

Conclusion: Based on these results, we conclude that CFAR's $\mathscr{P}_{\text{scale}}$ and $\mathscr{P}_{\text{h/m}}$ projectors enable developers to accurately identify how the working set of third-party or their own code changes as a function of the workload. Given that CFAR can extract distillates in < 30 mins, we envision such extraction and analysis of cache distillates to be a part of the regular development cycle (e.g., continuous integration), enabling developers to identify surprising performance behavior without having to write elaborate test suites.

How Does Data Structure Layout in Code I Call Interact with My Code?

Finally, we used the $\mathscr{P}_{\text{scale}}$ and $\mathscr{P}_{\text{h/m}}$ projectors to analyze the cache usage of the hash table implementations from Vigor [251] and Klint [192]. We did not write this code, but we read it and thought we understood it fairly well. This is the kind of analysis that Alice would do to answer the other part of her question in §6.1: is the hash table the predominant cause of persistent LLC misses?

The projections proved our expectations about the performance of the two tables wrong: The two hash tables organize keys, values, and 4 metadata fields in slightly different ways: Vigor stores them as 6 distinct arrays, while Klint encapsulates all 6 fields into a single 64B `struct` and maintains a single array with elements of this `struct` type. At first glance, it appears—and did to us too—that the latter always leads to better locality and improved performance. However, it turned out that this is not always true.

Applying $\mathscr{P}_{\text{scale}}$ and $\mathscr{P}_{\text{h/m}}$ on the `put`, `get`, and `delete` operations of the two implementations predicts the following: For a `put()` or `get()` call, both implementations bring 64B of data into the cache, but while Klint does so in one cache line, Vigor does so across 6 cache lines. When the table does not fit in the LLC, Klint suffers one LLC miss, while Vigor suffers 6. On the other

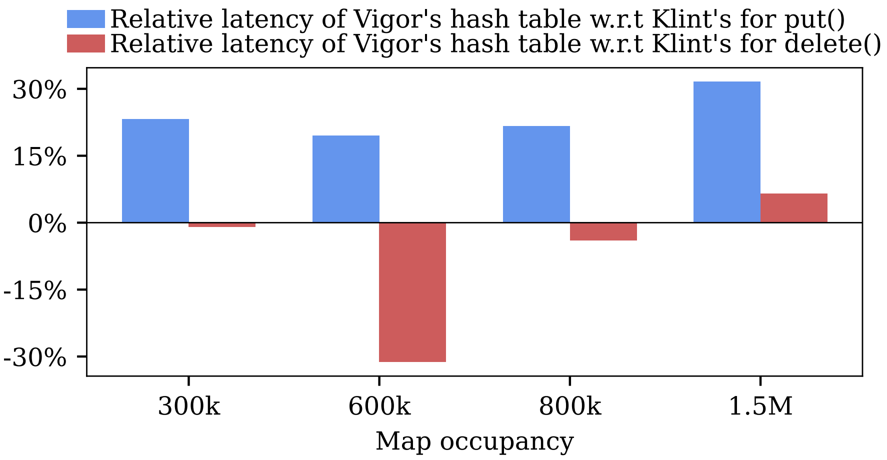


Figure 6.7: Relative latency (measured) of the Vigor hash table as compared to Klint's for `put()` and `delete()` calls. Positive (negative) numbers indicate that the Vigor table is slower (faster).

hand, for a `delete()` call, both implementations touch 32B, but while Vigor brings only these 32B into the LLC, Klint brings in 64B. On closer inspection, this is because Klint must bring in at least one entire cache-line-aligned struct (it cannot bring in half a cache line); hence, it always brings in the value and 2 other metadata fields, even though it never accesses them. As a result, for a range of table occupancies, Klint overflows the LLC and suffers 1 miss, while Vigor fits in the LLC and suffers none. $\mathscr{P}_{\mathrm{h/m}}$ predicts that this range begins at approximately 400k keys and ends at approximately 800k keys, at which point both implementations overflow the LLC.

To verify these predictions, we measured the latency and LLC misses incurred by the `put` and `delete` calls of the two implementations, for different map occupancies. Fig. 6.7 plots Vigor's latency overhead relative to Klint, as a function of map occupancy. As predicted: Klint `put` is consistently faster due to better locality. For occupancies of 400-800k keys, Vigor `delete` incurs 30% lower latency than Klint; moreover, Vigor incurs no misses, while Klint incurs 1. There was one discrepancy between $\mathscr{P}_{\mathrm{h/m}}$'s predictions and the outcome of our experiments: for occupancies above 860k, Vigor continued to incur 1 miss, whereas $\mathscr{P}_{\mathrm{h/m}}$ predicted 3. We believe that this is due to Intel's stride prefetcher, which our current cache model does not consider.

Conclusion: Data-structure developers often tailor their memory layout to different workloads [45–47]; CFAR's projections can make such subtle differences accessible to data-structure users, allowing them to pick the implementation best suited for their expected workoad without elaborate benchmarking.

Does My Code Lead To Inefficient Memory Access Patterns?

We now describe how CFAR's projections helped us uncover two inefficient cache access patterns in the kernel-bypass stack and the Hyperkernel's `mmap()` syscall, respectively.

Kernel-bypass stack: The inefficient access pattern in the kernel-bypass stack was due to the organization of the TCP process control block (PCB). The TCP PCB stores per-connection state and corresponds to the 4,5, and 19 cache lines touched per-connection. To understand what data was being accessed within the PCB for each stack, we wrote a simple projector that returned the offset (in cache lines) of each access within the PCB from the base of the PCB. Fig. 6.8 shows the projector.

```
def pcb_offset(seq):
    pcb = sympy.Symbol('pcb')
    # if address is an offset from only the PCB,
    # return (address-PCB)/64
    return [(x-pcb)//64 for x in seq if sympy.is_constant(x-pcb)]
```

Figure 6.8: Projector that returns offsets accessed within the TCP Process Control Block (PCB).

Applying this projector to the fast path's `rcv` and `snd` calls, made us realize that there was only a single access to the 5^{th} cache line in the PCB. Fig. 6.9 shows the list of cache-line accesses returned for the above calls:

```
# Receive fast path: KB stack
# Only one access to 5th cache line
[1,1,0,0,2,2,3,4,1,2,2,3]

# Send fast path: KB stack
# No access to 5th cache line
[2,3,3,1,1,3,3,3,3,1,2,3,2,2,1,1,1,1,0,0,2,1,2,2,1,0,2]
```

Figure 6.9: Offsets accessed within the PCB for the kernel-bypass stack's fast path.

Using the `filename:linenum` information that CFAR logs during symbolic execution (§6.3), we realized that the field being accessed was `keep_cnt_sent`, which was being updated on the `rcv` path to indicate that the connection was still live. We used this information to re-organize the PCB struct—we moved `keep_cnt_sent` into the first 4 cache lines and moved some of the timer fields (primarily used during retransmissions) to the 5^{th} line. Fig. 6.10 shows the list returned by the PCB offset-computing projector after the change, which confirmed that the fast path only accessed the first 4 cache lines.

```
# Receive fast path: KB stack
# No access to 5th cache line
[0,0,0,0,1,1,2,1,0,1,1,2]

# Send fast path: KB stack (updated)
# No access to 5th cache line
[1,2,2,0,0,2,2,2,2,0,1,2,1,1,3,3,3,3,3,3,3,1,3,1,1,0,0,1]
```

Figure 6.10: Offsets accessed within the PCB for the kernel-bypass stack's fast path after our fix.

We evaluated the impact of this change by running the same experiment we ran previously, where we measured the latency of the fast path as a function of the number of connections; Fig. 6.11 illustrates the results. We see that our fix has a significant impact on the fast path's connection scalability: touching one less cache line ensures that the stack can support 88k concurrent connections (as opposed to 72k) before suffering from a latency spike due to LLC misses.

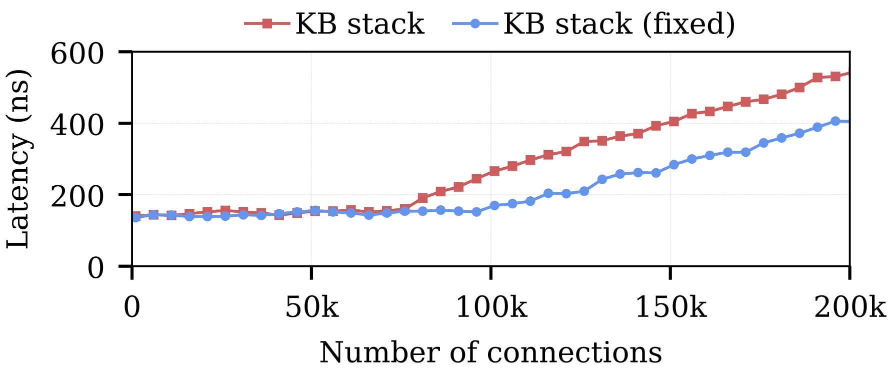


Figure 6.11: Measured latency as a function of the number of connections for the kernel-bypass stack, before and after our fix.

Hyperkernel `mmap()`: CFAR's $\mathscr{P}_{\text{scale}}$ projector enabled us to uncover and fix a subtle performance bug in Hyperkernel's `mmap()` implementation. The `mmap` code performs a four-level page walk, checking for permissions only before it allocates the final page. So, if it is called with invalid permissions, it does not exhibit incorrect behavior (it does not allocate the final page), but it still performs significant unnecessary work (allocates and zeroes out up to 3 new page-table pages, depending on where the walk fails). This brings up to $12KB$ of data into the L1 cache; given that most servers today have an L1 cache of 32KB, this code unnecessarily pollutes up to roughly 40% of the cache.

Figs. 6.12 and 6.13 show parts of `mmap`'s projections before and after we fixed the above behavior. Consider Fig. 6.12: Line 6 corresponds to the scenario where the walk fails at level 1 (no page-table page is allocated at that level for the target address), and the permissions are invalid; line 12 corresponds to the scenario where the code fails at level 2, and the permissions are valid. In the former case, the projection returns "201," while in the second, "202." So, the code touches almost the same number of cache lines, even though it should be accessing very different amounts of memory: in the former case it doesn't need to allocate any pages, whereas in the latter case it needs to start allocating at level 2. Now consider Fig. 6.13: Line 4 corresponds to the scenario where the permissions are invalid; in this case, the projection always returns "3," i.e., the code touches only 3 cache lines.

For the developer of this code, a read of the projection (before the fix) would immediately reveal that there is a performance problem: code paths that should be accessing very different amounts of memory touch almost the same number of cache lines.

Conclusion: The CFAR distillate, coupled with simple projectors enables developers to quickly inspect their code and identify inefficient cache access patterns without having to run elaborate benchmarks.

```
def mmap_dcache(va,perm):
    #State: pid, proctable, pages

    if [pages + [proctable+320*pid+16]*4096 + 8*((va>>39)&511)]:
        if not (perm & PTE_PERM_MASK):
            return 201
        return 265

    if [pages + [proctable+320*pid+16]*4096 + 8*((va>>30)&511)]:
        if not (perm & PTE_PERM_MASK):
            return 138
        return 202
    ....
```

Figure 6.12: Projection for buggy `mmap` code showing number of unique cache lines.

```
def mmap_optimized_dcache(va,perm):
    #State: pid, proctable, pages

    if not (perm & PTE_PERM_MASK):
        return 3

    if [pages + [proctable+320*pid+16]*4096 + 8*((va>>39)&511)]:
        return 265

    if [pages + [proctable+320*pid+16]*4096 + 8*((va>>30)&511)]:
        return 202
    .....
```

Figure 6.13: `mmap` projection after fix.

Can I Prove/Disprove the Absence of Leaks Due to Secret-Dependent Memory Accesses?

We used CFAR's $\mathscr{P}_{\text{crypt}}$ projector to analyze the 8 OpenSSL algorithms listed in Table 6.2. The first 7 are the ones mentioned in the beginning of §6.4, while the last one is from a previous version of OpenSSL (1.1). We included the latter because it is known to exhibit cache-based leakage (CVE-2018-0737 [182]), and we wanted to demonstrate CFAR's capability to identify this behavior (and none of the algorithms that we analyzed from the latest version of OpenSSL exhibit it).

Program	**Remarks**
OpenSSL 3.0 AES	Identified previously unknown branch-based leak
OpenSSL 3.0 ChaCha	Verified absence of secret-dependent branches, memory accesses
OpenSSL 3.0 ECDHE	Verified absence of secret-dependent branches, memory accesses
OpenSSL 3.0 MD5	Verified absence of secret-dependent branches, memory accesses
OpenSSL 3.0 MD4	Verified absence of secret-dependent branches, memory accesses
OpenSSL 3.0 Poly1305	Verified absence of secret-dependent branches, memory accesses
OpenSSL 3.0 SHA-256	Verified absence of secret-dependent branches, memory accesses
OpenSSL 1.1 RSA (CVE-2018-0737)	Reproduced known cache-based leak

Table 6.2: OpenSSL programs analyzed using CFAR's $\mathscr{P}_{\text{crypt}}$.

```
def ossl_cipher_unpadblock(buffer, buffer_length, block_size):

  if buffer.padding_length == 0:
    return 44
  if buffer.padding_length > block_size:
    return 48
  return 57 + 19*buffer.padding_length
```

Figure 6.14: Projection showing how instruction count for AES's cipher unpadding function is a function of `buffer.padding_length`, which must remain secret.

```
def ossl_cipher_unpadblock(buffer, buffer_length, block_size):
  return 2985
```

Figure 6.15: Projection showing instruction count for AES's cipher unpadding function after our fix.

The $\mathscr{P}_{\text{crypt}}$ projector enabled us to confirm the cache-based leakage in OpenSSL 1.1, and also uncover a previously-unknown branch-based leakage in OpenSSL 3.0.0. In particular, the output projection revealed that the cipher-block unpadding function used by AES branches depending on secret input. To further investigate, we wrote another projector that shows the number of executed instructions; this revealed that the number of instructions executed by the function in question depends on the length of the input buffer's padding, making the code vulnerable to padding oracles. We reported this leakage to the maintainers who confirmed it [183], and we have submitted a pull request [184] that has undergone multiple rounds of review and is in the final stages of getting merged.

Figs. 6.14, 6.15 show the output projection of the latter projector for the unpadding function before and after the fix. The former clearly indicates that the number of instructions depends on padding length, whereas the latter returns the same value independent of the input.

Our experience with OpenSSL suggests that incorporating CFAR and its projectors into the development cycle could be useful to developers. For instance, we learnt that the specific branch leakage had been latent since OpenSSL 1.1.1 (released in 2019) because the developer "just reused the code," and it had been missed despite the thorough code reviews that OpenSSL undergoes. Yet for the developer, a quick glance at the projection (before the fix) would have immediately revealed the problem. So, perhaps if distillates and projections were extracted regularly, e.g., as part of continuous integration, branch and cache leakage would be detected before making their way into production.

Conclusion: Since the distillate captures all information relevant to how a piece of code accesses memory, CFAR can help developers efficiently reason about more than just performance properties; $\mathscr{P}_{\text{crypt}}$ can help identify both branch- and cache-based leakage in cryptographic code (or prove their absence).

6.5 Related Work

Given the ever growing gap between processor and memory speeds, understanding how systems code uses the cache has been extensively studied. However, we are not aware of any tool that (like CFAR) possesses predictive power across workloads. All prior tools we know of are limited to providing insights only about the workloads that the tool was run on.

We drew significant inspiration from work in the early 90s on abstract execution [148] and memory tracing [78]. Both these systems were designed to be able to replay the memory trace of a piece of systems code (just like CFAR's distillates), but only for concrete inputs. This is because their goal was to avoid having to store large memory traces required for computer architecture simulations, instead they sought to generate this trace on the fly. CFAR's distillate thus represents a generalized version of their work, that builds on advancements in automated program analysis.

More recent work has focussed on building better profilers [35, 59, 139, 153, 158, 189, 230] to help developers fix performance issues that they observe due to poor cache utilization. The key trade-off that such systems explore is between ease of use, performance overhead and the level of detail at which they can analyze the execution of the given input workload. The most detailed memory profiler we know of is Memspy [158]. MemSpy uses a system simulator to execute an application which allows it to interpose on all memory accesses and build a complete map of the cache. Thus, MemSpy can account for and explain every single cache miss and using a processor accurate model can approximate memory access latencies. However, Memspy requires applications to be ported to its simulator which can be a painstaking task. Additionally its high performance overhead ensures that it can only be used to profile a limited number of input workloads. At the other end of the spectrum are profilers such as DMon [139] which work-off-the-shelf for almost all systems code, and have very low-enough overheads to run continuously in production. The downside is that they can only be used to monitor a very specific subset of events and cannot provide the visibility that MemSpy does. We see profilers as complementary to CFAR. CFAR's distillate and projectors allow developers to quickly understand which workloads might be of interest and cause unexpected cache behavior. Once they narrow this search space, they can use state-of-the-art profilers to study these workloads in great detail.

Lastly, DTrace [39], eBPF [56] and DeBox [202] allow developers to not only profile their applications, but also the kernel to observe, among other information, its cache usage behavior. DTrace and eBPF achieve this by allowing developers to write their own instrumentation code that is loaded in the kernel. DeBox takes a different approach and has the kernel expose performance information (including cache usage) after each system call is completed in a `DeBoxInfo struct`, just like the kernel exposes semantic information via error codes today.

6.6 Conclusion

In this chapter, we presented CFAR a tool that enables developers to scrutinize and answer arbitrary questions about the CPU cache usage of their own, as well as third party code. CFAR's key contribution is the CFAR distillate: a precise intermediate representation that contains all information about how the code in question accesses memory, and discards everything else. CFAR allows developers to write simple programs (projectors) to query the CFAR distillate and answer specific questions about cache usage. This enables them to understand, among other things, the micro-architectural side effects of invoking a piece of code, and thus gain greater visibility into the latency impact of using third-party code.

CFAR's approach is similar to that of LINX in two notable ways. First, both analyze the input program separately from the environment in which it runs. For any given P, they first extract a representation that is uniquely determined by P—general-case interfaces and distillates, respectively—and then analyze how that representation interacts with its environment (e.g., different underlying hardware, different cache architectures, etc.). This split analysis ensures flexibility and allows the general-case interface to be tailored to arbitrary deployment environments and the distillate to answer arbitrary questions about the program's cache usage.

Second, both represent the property of interest—processing latency and answers about cache usage, respectively—as programs that accept the same input(s) as P. Like latency interfaces, both P_{dist} and P_{proj} are programs. P_{proj} in particular, can be thought of as being an interface that describes the latency of P in terms of the metric defined by the developer's specific question. Thus, our work on CFAR reinforces our belief that simple, executable programs are best suited to summarizing the latency behavior of systems code in a manner that is simultaneously readable and accurate.

Wrapping Up Part IV

7 Future Work

Both LINX and CFAR faced two constant challenges. First, the unpredictability introduced by general-purpose hardware, which makes it challenging for LINX to reason about NF tail latency and introduces inaccuracies in CFAR's cache projections. Second, the limitations of automated program analysis which impact the scalability of both LINX and CFAR and often cause them to timeout while analyzing complex systems code.

We now describe two directions for future work aimed at overcoming each of the above challenges.

7.0.1 Performance Interfaces for Hardware Accelerators

With the decline of Moore's law, systems developers are increasingly reliant on hardware accelerators for performance improvements. From datacenters to hand-held devices, hardware accelerators are used to speed up a wide variety of applications such as machine learning [16, 132, 133, 173], video processing [84, 198], compression, encryption [48, 118] and systems infrastructure tasks [17, 89, 102, 136].

While the ship has arguably sailed with respect to precise performance interfaces for general-purpose hardware [82, 105], we are optimistic that such interfaces are feasible for accelerators for two reasons: First, accelerators are much simpler than today's general-purpose servers, and so we can rely on a plethora of high-fidelity models from the 80s-90s (summarized in [77]) to analyze them. Second, initial discussions with accelerator builders indicated that they have an intuitive understanding of the factors that impact performance, and for them, writing a performance interface for their hardware is on par (in terms of difficulty) with a software developer writing a semantic interface for their software. So we see hardware accelerators providing a golden opportunity for researchers to rethink performance modularity from the ground up and provide firm foundations for the design of systems with predictable performance.

That said, performance interfaces for hardware accelerators will likely need a rethink of what the right representation for performance behavior is. This is because hardware, unlike software, has an inherently parallel execution model, with multiple circuits (e.g., different pipeline stages) executing in parallel to the same system clock. It is likely that to accurately predict performance for such a model, the interface will need to reflect this parallelism which raises the question of whether programs (which are easiest to read when sequential) remain the right representation.

7.0.2 Latency Verification Tools That Cut across the Stack

To make latency interfaces immediately useful, we focused on being able to extract them automatically from systems code and relied on automated program analysis techniques to do so. However, this comes with the caveat of being unable to scale to complex systems (e.g., Linux) which limits the applicability of our techniques.

We are particularly interested in augmenting state-of-the-art proof assistants (e.g., Coq [55], Isabelle [121]) with the ability to reason about the latency in machine instructions of systems code to enable latency interfaces to scale to more complex code. That is, we wanted developers to be able to interactively verify post-conditions about the execution latency, just as they do today for semantics. We are optimistic that this will help us scale our latency interfaces to more complex code since proof assistants are increasingly scaling to complex systems code such as file systems [43] and even key-value stores similar to the ones we used in Concord [109].

The key challenge here is that of abstraction. Proof assistants currently do not possess visibility into the executing binary. This is because they do not require it; they can reason about more high-level representations of the program (e.g. the abstract syntax tree) which provide richer semantic information, safe in the knowledge that the semantics will remain unchanged as the code is translated to lower-level representations. In contrast, performance properties are rarely preserved as the code is translated to lower-level representations, so being able to verify performance properties will require a verification tool that can reason about multiple representations of the code simultaneously.

8 Conclusion

In this thesis, we presented a three-part approach to enable developers to reason more precisely about the latency behavior of systems code.

First, we worked within the confines of existing representations for system latency and studied the problem of ensuring that datacenter applications meet their microsecond-scale tail-latency SLOs. We showed how this problem necessitated low-overhead scheduling mechanisms and proposed Concord: an efficient scheduling runtime for microsecond-scale datacenter applications. Concord demonstrates that careful approximation (as opposed to canonical implementation) of theoretically optimal scheduling policies enables new microsecond-scale mechanisms that provide significant throughput benefits while ensuring the applications continue to meet the same tail latency SLO. Concord introduced two novel mechanisms: (1) compiler-enforced cooperation which enables preemptive scheduling at 4× lower overhead than the state of the art, and (2) a work-conserving dispatcher which enables the dispatcher thread (previously responsible only for scheduling requests) to contribute to application goodput.

We then advocated that systems code must have a latency interface that describes its latency behavior and related side effects for all inputs, just like the code's semantic interface describes its functionality and related side effects.

We proposed that the latency interface for a system be represented as a program that accepts the same input(s) as the system and returns its processing latency. We introduced three key ideas that enable latency interfaces to summarize latency in a manner that is simultaneously accurate and readable: (1) Latency-critical variables (LCVs) that succinctly summarize the impact of latency of all factors other than the current input (e.g., prior inputs, system state, configuration, and runtime environment), (2) Latency resolution which provides readers of the interface with explicit control over the trade-off between accuracy and readability, and (3) Deployment-specific interfaces that enable developers unfamiliar with the system's implementation details to reason about its latency behavior in their specific use-case scenario.

Finally, since a latency interface is incomplete without a description of side effects, we studied the problem of helping developers reason about the micro-architectural (specifically CPU cache) side effects of systems code. We present CFAR (Cache Footprint AnalyzeR), a tool that processes a piece P of systems code into answers to developers' questions about how that code uses the cache. CFAR's processing consists of two phases: In the former, CFAR takes as input the code and outputs an intermediate representation (a "distillate") that contains all the information on how the code accesses memory. In the latter, developers can write simple programs ("projectors") that use the distillate to compute answers ("projections") to specific questions about P's cache usage.

CFAR's approach is similar to that of LINX in two notable ways.
First, both analyze the input program separately from the environment in which it runs. For any given P, they first extract a representation that is uniquely determined by P—general-case

interfaces and distillates, respectively—and then analyze how that representation interacts with its environment (e.g., different underlying hardware, different cache architectures, etc.). This split analysis ensures flexibility and allows the general-case interface to be tailored to arbitrary deployment environments and the distillate to answer arbitrary questions about the program's cache usage.

Second, both represent the property of interest—processing latency and answers about cache usage, respectively—as programs that accept the same input(s) as P. Like latency interfaces, both P_{dist} and P_{proj} are programs. P_{proj} in particular, can be thought of as being an interface that describes the latency of P in terms of the metric defined by the developer's specific question. Thus, our work on CFAR reinforces our belief that simple, executable programs are best suited to summarizing the latency behavior of systems code in a manner that is simultaneously readable and accurate.

We are optimistic about what the future holds for latency interfaces. We believe that the widespread adoption of such interfaces could be the first step towards a future where we can build systems with well-understood performance, just like types and object-oriented programming enabled us to build programs that were orders of magnitude bigger, better, yet safer than any that came before.

Bibliography

[1] The Biggest Thing Amazon Got Right: The Platform. https://old.gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/. [Last accessed on 2023-06-26].

[2] Address Space Layout Randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization. [Last accessed on 2023-06-26].

[3] Afek, Y., Bremler-Barr, A., Harchol, Y., Hay, D., and Koral, Y. Making DPI Engines Resilient to Algorithmic Complexity Attacks. *Transactions on Networking* (2016).

[4] Akamai Online Retail Performance Report: Milliseconds Are Critical. https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report. [Last accessed on 2023-06-26].

[5] Albert, E., Arenas, P., Genaim, S., and Puebla, G. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *International Symposium on Static Analysis* (2008).

[6] Albert, E., Arenas, P., Genaim, S., Puebla, G., and Zanardini, D. Cost Analysis of Java Bytecode. In *European Symposium on Programming* (2007).

[7] Aldinucci, M., Danelutto, M., Kilpatrick, P., and Torquati, M. Fastflow: High-Level and Efficient Streaming on Multicore. In *Journal on Programming Multi-Core and Many-Core Computing Systems* (2017).

[8] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. Verifying constant-time implementations. In *USENIX Security Symposium* (2016).

[9] AMD SEV-SNP. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. [Last accessed on 2023-06-26].

[10] Amvrosiadis, G., Park, J. W., Ganger, G. R., Gibson, G. A., Baseman, E., and DeBardeleben, N. On the Diversity of Cluster Workloads and its Impact on Research Results. In *USENIX Annual Technical Conference* (2018).

[11] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Symposium on Operating Systems Principles* (1991).

[12] Arashloo, M. T., Lavrov, A., Ghobadi, M., Rexford, J., Walker, D., and Wentzlaff, D. Enabling Programmable Transport Protocols in High-Speed NICs. In *Symposium on Networked Systems Design and Implementation* (2020).

[13] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Conference* (2012).

[14] Atre, N., Sadok, H., Chiang, E., Wang, W., and Sherry, J. SurgeProtector: Mitigating Temporal Algorithmic Complexity Attacks Using Adversarial Scheduling. In *ACM SIGCOMM Conference* (2022).

[15] AWS Elasticache. https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/AutoScaling.html. [Last accessed on 2023-06-26].

[16] AWS Inferentia Accelerators for Deep Learning Inference. https://aws.amazon.com/machine-learning/inferentia. [Last accessed on 2023-06-26].

[17] AWS Nitro System. https://aws.amazon.com/ec2/nitro. [Last accessed on 2023-06-26].

[18] Ayers, G., Nagendra, N. P., August, D. I., Cho, H. K., Kanev, S., Kozyrakis, C., Krishnamurthy, T., Litz, H., Moseley, T., and Ranganathan, P. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *International Symposium on Computer Architecture* (2019).

[19] Bansal, D., DeGrace, G., Tewari, R., Zygmunt, M., Grantham, J., Gai, S., Baldi, M., Doddapaneni, K., Selvarajan, A., Arumugam, A., Raman, B., Gupta, A., Jain, S., Jagasia, D., Langlais, E., Srivastava, P., Hazarika, R., Motwani, N., Tiwari, S., Grant, S., Chandra, R., and Kandula, S. Disaggregating Stateful Network Functions. In *Symposium on Networked Systems Design and Implementation* (2023).

[20] Barbette, T., Soldani, C., and Mathy, L. Fast Userspace Packet Processing. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2015).

[21] Barroso, L. A., Dean, J., and Hölzle, U. Web Search for a Planet: The Google Cluster Architecture. In *IEEE Micro* (2003).

[22] Barroso, L. A., Marty, M., Patterson, D. A., and Ranganathan, P. Attack of the Killer Microseconds. *Communications of the ACM* (2017).

[23] Basu, N., Montanari, C., and Eriksson, J. Frequent Background Polling on a Shared Thread, Using Light-Weight Compiler Interrupts. In *International Conference on Programming Language Design and Implementation* (2021).

[24] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Symposium on Operating Systems Design and Implementation* (2014).

[25] Benson, T., Akella, A., and Maltz, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *Internet Measurement Conference* (2010).

[26] The Big O Notation. https://en.wikipedia.org/wiki/Big_O_notation. [Last accessed on 2023-06-26].

[27] Bijlani, A., and Ramachandran, U. Extension Framework for File Systems in User space. In *USENIX Annual Technical Conference* (2019).

[28] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. In *ACM SIGARCH Computer Architecture News* (2011).

[29] Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A., and Heiser, G. Timing Analysis of a Protected Operating System Kernel. In *Real-Time Systems Symposium* (2011).

[30] Boonstoppel, P., Cadar, C., and Engler, D. R. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[31] Boucher, S., Kalia, A., Andersen, D. G., and Kaminsky, M. Lightweight Preemptible Functions. In *USENIX Annual Technical Conference* (2020).

[32] Brunella, M. S., Belocchi, G., Bonola, M., Pontarelli, S., Siracusano, G., Bianchi, G., Cammarano, A., Palumbo, A., Petrucci, L., and Bifulco, R. hXDP: Efficient Software Packet Processing on FPGA NICs. In *Symposium on Operating Systems Design and Implementation* (2020).

[33] Burger, D., and Austin, T. M. The simplescalar tool set, version 2.0. In *ACM SIGARCH Computer Architecture News* (1997).

[34] Burnim, J., Juvekar, S., and Sen, K. WISE: Automated Test Generation for Worst-Case Complexity. In *International Conference on Software Engineering* (2009).

[35] Cachegrind: A Cache and Branch-Prediction Profiler. https://valgrind.org/docs/manual/cg-manual.html. [Last accessed on 2023-06-26].

[36] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation* (2008).

[37] Canini, M., Kostic, D., Rexford, J., and Venzano, D. Automating the Testing of OpenFlow Applications. In *International Workshop on Rigorous Protocol Engineering* (2011).

[38] Canini, M., Venzano, D., Perešíni, P., Kostić, D., and Rexford, J. A NICE Way to Test OpenFlow Applications. In *Symposium on Networked Systems Design and Implementation* (2012).

[39] Cantrill, B., Shapiro, M. W., and Leventhal, A. H. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference* (2004).

[40] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. C. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX Conference on File and Storage Technologies* (2020).

[41] Cao, Z., Tarasov, V., Raman, H. P., Hildebrand, D., and Zadok, E. On the Performance Variation in Modern Storage Stacks. In *USENIX Conference on File and Storage Technologies* (2017).

[42] Chen, B., Liu, Y., and Le, W. Generating Performance Distributions via Probabilistic Symbolic Execution. In *International Conference on Software Engineering* (2016).

[43] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Symposium on Operating Systems Principles* (2015).

[44] Chen, S., Delimitrou, C., and Martinez, J. F. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[45] Chilimbi, T. M., Davidson, B., and Larus, J. R. Cache-Conscious Structure Definition. In *International Conference on Programming Language Design and Implementation* (1999).

[46] Chilimbi, T. M., Hill, M. D., and Larus, J. R. Cache-Conscious Structure Layout. In *International Conference on Programming Language Design and Implementation* (1999).

[47] Chilimbi, T. M., and Larus, J. R. Using Generational Garbage Collection To Implement Cache-Conscious Data Placement. In *International Symposium on Memory Management* (1998).

[48] Chiosa, M., Maschi, F., Müller, I., Alonso, G., and May, N. Hardware Acceleration of Compression and Encryption in SAP HANA. In *International Conference on Very Large Databases* (2022).

[49] Commits to the eBPF Code in the Cilium Project. https://github.com/cilium/cilium/commits/master/bpf. [Last accessed on 2023-06-26].

[50] Cilium Project. https://cilium.io. [Last accessed on 2023-06-26].

[51] Cloudlab. https://cloudlab.us. [Last accessed on 2023-06-26].

[52] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Symposium on Cloud Computing* (2010).

[53] Coppa, E., Demetrescu, C., and Finocchi, I. Input-Sensitive Profiling. In *International Conference on Programming Language Design and Implementation* (2012).

[54] Coppa, E., Demetrescu, C., Finocchi, I., and Marotta, R. Estimating the Empirical Cost Function of Routines with Dynamic Workloads. In *International Symposium on Code Generation and Optimization* (2014).

[55] Coq Proof Assistant. https://coq.inria.fr. [Last accessed on 2023-06-26].

[56] Corbet, J. The BPF system call API, version 14. https://lwn.net/Articles/612878. [Last accessed on 2023-06-26].

[57] Cousot, P. Abstract interpretation. In *ACM Computing Surveys* (1996).

[58] Crosby, S. A., and Wallach, D. S. Denial of Service via Algorithmic Complexity Attacks. In *USENIX Security Symposium* (2003).

[59] Curtsinger, C., and Berger, E. D. Coz: Finding Code that Counts with Causal Profiling. *Commun. ACM* (2018).

[60] Daglis, A., Sutherland, M., and Falsafi, B. RPCValet: NI-Driven Tail-Aware Balancing of μs-Scale RPCs. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[61] Dalton, M., Schultz, D., Arefin, A., Docauer, A., Gupta, A., Fahs, B. M., Rubinstein, D., Zermeno, E. C., Rubow, E., Adriaens, J., Alpert, J. L., Ai, J., Olson, J., DeCabooter, K. P., de Kruijf, M. A., Hua, N., Lewis, N., Kasinadhuni, N., Crepaldi, R., Krishnan, S., Venkata, S., Richter, Y., Naik, U., and Vahdat, A. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *Symposium on Networked Systems Design and Implementation* (2018).

[62] David, T., Guerraoui, R., and Trigonakis, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Symposium on Operating Systems Principles* (2013).

[63] DAWNBench: An End-to-End Deep Learning Benchmark and Competition. https://dawn.cs.stanford.edu/benchmark. [Last accessed on 2023-06-26].

[64] de Moura, L. M., and Bjørner, N. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[65] Dean, J., and Barroso, L. A. The Tail at Scale. In *Communications of the ACM* (2013).

[66] Demoulin, H. M., Fried, J., Pedisich, I., Kogias, M., Loo, B. T., Phan, L. T. X., and Zhang, I. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Symposium on Operating Systems Principles* (2021).

[67] Dobrescu, M., and Argyraki, K. Software Dataplane Verification. In *Symposium on Networked Systems Design and Implementation* (2014).

[68] Dobrescu, M., Argyraki, K., and Ratnasamy, S. Toward Predictable Performance in Software Packet-Processing Platforms. In *Symposium on Networked Systems Design and Implementation* (2012).

[69] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *Symposium on Operating Systems Principles* (2009).

[70] DPDK: Data Plane Development Kit. https://dpdk.org. [Last accessed on 2023-06-26].

[71] Ethtool Driver Identifier. https://docs.huihoo.com/doxygen/linux/kernel/3.7/include_2uapi_2linux_2ethtool_8h_source.html#l00085. [Last accessed on 2023-06-26].

[72] Drumond, M., Daglis, A., Mirzadeh, N. S., Ustiugov, D., Picorel, J., Falsafi, B., Grot, B., and Pnevmatikatos, D. N. The Mondrian Data Engine. In *International Symposium on Computer Architecture* (2017).

[73] Drumond, M., Daglis, A., Mirzadeh, N. S., Ustiugov, D., Picorel, J., Falsafi, B., Grot, B., and Pnevmatikatos, D. N. Algorithm/Architecture Co-Design for Near-Memory Processing. In *Operating Systems Review* (2018).

[74] Dunkels, A. Design and Implementation of the lwIP TCP/IP Stack. Tech. Rep. 2:77, Swedish Institute of Computer Science, 2001.

[75] Commits to eBPF Maps in the Linux Kernel. https://github.com/torvalds/linux/commits/master/kernel/bpf. [Last accessed on 2023-06-26].

[76] eBPF maps. https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf_maps.html. [Last accessed on 2023-06-26].

[77] Eeckhout, L. Computer Architecture Performance Evaluation Methods. In *Synthesis Lectures on Computer Architecture* (2010).

[78] Eggers, S. J., Keppel, D. R., Koldinger, E. J., and Levy, H. M. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. In *ACM SIGMETRICS Conference* (1990).

[79] Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W., and Hosein, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *Symposium on Networked Systems Design and Implementation* (2016).

[80] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., and Carle, G. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference* (2015).

[81] Erik Stenman. The Beam Book. https://blog.stenmans.org/theBeamBook. [Last accessed on 2023-06-26].

[82] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. A Performance Counter Architecture for Computing Accurate CPI Components. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2006).

[83] Facebook, Google and Apple hit by Unusual Outages. https://www.wsj.com/articles/facebook-and-instagram-suffer-lengthy-outages-11552539752. [Last accessed on 2023-06-26].

[84] Facebook: Video transcoding with Mount Shasta. https://engineering.fb.com/2019/03/14/data-center-engineering/accelerating-infrastructure.

[85] Farshin, A., Roozbeh, A., Jr., G. Q. M., and Kostic, D. Make the Most out of Last Level Cache in Intel Processors. In *ACM European Conference on Computer Systems* (2019).

[86] Fayazbakhsh, S. K., Chiang, L., Sekar, V., Yu, M., and Mogul, J. C. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Symposium on Networked Systems Design and Implementation* (2014).

[87] Filliâtre, J., Gondelman, L., and Paskevich, A. The Spirit of Ghost Code. In *Formal Methods in System Design* (2016).

[88] Firestone, D. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *Symposium on Networked Systems Design and Implementation* (2017).

[89] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A. M., Chung, E. S., Chandrappa, H. K., Chaturmohta, S., Humphrey, M., Lavier, J., Lam, N., Liu, F., Ovtcharov, K., Padhye, J., Popuri, G., Raindel, S., Sapre, T., Shaw, M., Silva, G., Sivakumar, M., Srivastava, N., Verma, A., Zuhair, Q., Bansal, D., Burger, D., Vaid, K., Maltz, D. A., and Greenberg, A. G. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Symposium on Networked Systems Design and Implementation* (2018).

[90] Freud Source Code Repository. https://github.com/usi-systems/freud. [Last accessed on 2023-06-26].

[91] Fried, J., Ruan, Z., Ousterhout, A., and Belay, A. Caladan: Mitigating Interference at Microsecond Timescales. In *Symposium on Operating Systems Design and Implementation* (2020).

[92] Fuerst, A., Novakovic, S., Goiri, I., Chaudhry, G. I., Sharma, P., Arya, K., Broas, K., Bak, E., Iyigun, M., and Bianchini, R. Memory-Harvesting VMs in Cloud Platforms. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2022).

[93] Furia, C. A., Meyer, B., and Velder, S. Loop Invariants: Analysis, Classification, and Examples. *ACM Computing Survey* (2014).

[94] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., and Delimitrou, C. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software

Implications for Cloud & Edge Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).

[95] Ganesh, V., and Dill, D. L. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification* (2007).

[96] Ghost Variables in Software Verification. http://whiley.org/2014/06/20/understanding-ghost-variables-in-software-verification. [Last accessed on 2023-06-26].

[97] Giacomoni, J., Moseley, T., and Vachharajani, M. FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue. In *Symposium on Principles and Practice of Parallel Programming* (2008).

[98] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. In *International Conference on Programming Language Design and Implementation* (2005).

[99] Godefroid, P., and Luchaup, D. Automatic Partial Loop Summarization in Dynamic Test Generation. In *International Symposium on Software Testing and Analysis* (2011).

[100] Goldsmith, S., Aiken, A., and Wilkerson, D. S. Measuring empirical computational complexity. In *Symposium on the Foundations of Software Engineering* (2007).

[101] Gong, J., Li, Y., Anwer, B., Shaikh, A., and Yu, M. Microscope: Queue-based Performance Diagnosis for Network Functions. In *ACM SIGCOMM Conference* (2020).

[102] Google-Intel Infrastructure Processing Unit (IPU). https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html. [Last accessed on 2023-06-26].

[103] Google Cloud Storage Incident. https://status.cloud.google.com/incident/storage/19002. [Last accessed on 2023-06-26].

[104] Goroutines. https://go.dev/tour/concurrency. [Last accessed on 2023-06-26].

[105] Guarnieri, M., Köpf, B., Reineke, J., and Vila, P. Hardware-Software Contracts for Secure Speculation. In *IEEE Symposium on Security and Privacy* (2021).

[106] Gulwani, S. SPEED: Symbolic Complexity Bound Analysis. In *International Conference on Computer Aided Verification* (2009).

[107] Gulwani, S., Mehra, K. K., and Chilimbi, T. M. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Symposium on Principles of Programming Languages* (2009).

[108] Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V., and Satria, A. D. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Symposium on Cloud Computing* (2014).

[109] Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., and Parno, B. Storage Systems are Distributed Systems (So Verify Them That Way!). In *Symposium on Operating Systems Design and Implementation* (2020).

[110] He, Z., Hu, G., and Lee, R. B. New Models for Understanding and Reasoning about Speculative Execution Attacks. In *International Symposium on High-Performance Computer Architecture* (2021).

[111] Hoffmann, J., and Jost, S. Two Decades of Automatic Amortized Resource Analysis. In *Mathematical Structures in Computer Science* (2022).

[112] Hu, Y., Huang, G., and Huang, P. Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution. In *Symposium on Operating Systems Design and Implementation* (2020).

[113] Humphries, J. T., Kaffes, K., Mazières, D., and Kozyrakis, C. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *ACM Workshop on Hot Topics in Networks* (2019).

[114] Humphries, J. T., Natu, N., Chaugule, A., Weisse, O., Rhoden, B., Don, J., Rizzo, L., Rombakh, O., Turner, P., and Kozyrakis, C. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Symposium on Operating Systems Principles* (2021).

[115] Hundt, R., Raman, E., Thuresson, M., and Vachharajani, N. MAO — An Extensible Micro-Architectural Optimizer. In *International Symposium on Code Generation and Optimization* (2011).

[116] Hunt, G., and Larus, J. Singularity: Rethinking the Software Stack. *Operating Systems Review* (2007).

[117] Ibanez, S., Mallery, A., Arslan, S., Jepsen, T., Shahbaz, M., Kim, C., and McKeown, N. The nanoPU: A Nanosecond Network Stack for Datacenters. In *Symposium on Operating Systems Design and Implementation* (2021).

[118] Intel QAT: Accelerating Data Compression and Encryption. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-quick-assist-technology-overview.html.

[119] Intel TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. [Last accessed on 2023-06-26].

[120] 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. [Last accessed on 2023-06-26].

[121] Isabelle Proof Assistant. https://isabelle.in.tum.de. [Last accessed on 2023-06-26].

[122] Intel Network Adapter Driver for PCIe. Intel 10 Gigabit Ethernet Network Connections under Linux. https://downloadcenter.intel.com/download/14687.

[123] Iyer, R., Argyraki, K., and Candea, G. Performance Interfaces for Network Functions. In *Symposium on Networked Systems Design and Implementation* (2022).

[124] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K., and Candea, G. Performance Contracts for Software Network Functions. In *Symposium on Networked Systems Design and Implementation* (2019).

[125] Iyer, R. R., Ma, J., Argyraki, K. J., Candea, G., and Ratnasamy, S. The Case for Performance Interfaces for Hardware Accelerators. In *Workshop on Hot Topics in Operating Systems* (2023).

[126] Iyer, R. R., Unal, M., Kogias, M., and Candea, G. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Symposium on Operating Systems Principles* (2023).

[127] Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. TRACER: A Symbolic Execution Tool for Verification. In *International Conference on Computer Aided Verification* (2012).

[128] Java String hashCode. https://docs.oracle.com/javase/6/docs/api/java/lang/String.html#hashCode().

[129] Jetstream Benchmark for Javascript Engines. https://www.browserbench.org/JetStream/. [Last accessed on 2023-06-26].

[130] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. In *International Conference on Programming Language Design and Implementation* (2012).

[131] Jost, S., Loidl, H., Hammond, K., Scaife, N., and Hofmann, M. "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In *World Congress on Formal Methods* (2009).

[132] Jouppi, N. P., Yoon, D. H., Ashcraft, M., Gottscho, M., Jablin, T. B., Kurian, G., Laudon, J., Li, S., Ma, P. C., Ma, X., Norrie, T., Patil, N., Prasad, S., Young, C., Zhou, Z., and Patterson, D. A. Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In *International Symposium on Computer Architecture* (2021).

[133] Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A.,

Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *International Symposium on Computer Architecture* (2017).

[134] Kaffes, K., Chong, T., Humphries, J. T., Belay, A., Mazières, D., and Kozyrakis, C. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *Symposium on Networked Systems Design and Implementation* (2019).

[135] Kalia, A., Kaminsky, M., and Andersen, D. G. Datacenter RPCs can be General and Fast. In *Symposium on Networked Systems Design and Implementation* (2019).

[136] Karandikar, S., Leary, C., Kennelly, C., Zhao, J., Parimi, D., Nikolic, B., Asanovic, K., and Ranganathan, P. A Hardware Accelerator for Protocol Buffers. In *International Symposium on Microarchitecture* (2021).

[137] Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. E. TAS: TCP acceleration as an OS service. In *ACM European Conference on Computer Systems* (2019).

[138] Khalid, J., Gember-Jacobson, A., Michael, R., Abhashkumar, A., and Akella, A. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *Symposium on Networked Systems Design and Implementation* (2016).

[139] Khan, T. A., Neal, I., Pokam, G., Mozafari, B., and Kasikci, B. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *Symposium on Operating Systems Design and Implementation* (2021).

[140] King, J. C. Symbolic Execution and Program Testing. *Journal of the ACM 19*, 7 (1976).

[141] Kogias, M., Iyer, R., and Bugnion, E. Bypassing the Load Balancer without Regrets. In *Symposium on Cloud Computing* (2020).

[142] Kogias, M., Prekas, G., Ghosn, A., Fietz, J., and Bugnion, E. R2P2: Making RPCs First-Class Datacenter Citizens. In *USENIX Annual Technical Conference* (2019).

[143] Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., and Wintersteiger, C. M. Loop Summarization Using Abstract Transformers. In *Automated Technology for Verification and Analysis* (2008).

[144] Krude, J., Rüth, J., Schemmel, D., Rath, F., Folbort, I., and Wehrle, K. Determination of Throughput Guarantees for Processor-Based SmartNICs. In *International Conference on Emerging Networking Experiments and Technologies* (2021).

[145] Kumar, P., Dukkipati, N., Lewis, N., Cui, Y., Wang, Y., Li, C., Valancius, V., Adriaens, J., Gribble, S., Foster, N., and Vahdat, A. PicNIC: Predictable Virtualized NIC. In *ACM SIGCOMM Conference* (2019).

[146] Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. Efficient State Merging in Symbolic Execution. In *International Conference on Programming Language Design and Implementation* (2012).

[147] Lampson, B. Hints and Principles for Computer System Design (Updated). https://www.microsoft.com/en-us/research/uploads/prod/2019/09/Hints-and-Principles-v1-full.pdf, 2020. [Last accessed on 2023-06-26].

[148] Larus, J. R. Abstract Execution: A Technique for Efficiently Tracing Programs. *Softw. Pract. Exp.* (1990).

[149] Lemieux, C., Padhye, R., Sen, K., and Song, D. PerfFuzz: Automatically Generating Pathological Inputs. In *International Symposium on Software Testing and Analysis* (2018).

[150] LevelDB. https://github.com/google/leveldb. [Last accessed on 2023-06-26].

[151] libVig Source Code . https://github.com/vigor-nf/vigor/tree/master/libvig/verified. [Last accessed on 2023-06-26].

[152] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Symposium on Networked Systems Design and Implementation* (2014).

[153] The Linux Perf Tool. https://perf.wiki.kernel.org. [Last accessed on 2023-06-26].

[154] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. Heracles: Improving Resource Efficiency at Scale. In *International Symposium on Computer Architecture* (2015).

[155] Luke Cheeseman and Matthew J. Parkinson and Sylvan Clebsch and Marios Kogias and Sophia Drossoupolou and David Chisnall and Tobias Wrigstad and Paul Lietar. When Concurrency Matters: Behaviour-Oriented Concurrency. In *ACM Conference on Object-oriented Programming, Systems, Languages, and Applications* (2023).

[156] Manousis, A., Sharma, R. A., Sekar, V., and Sherry, J. Contention-Aware Performance Prediction For Virtualized Network Functions. In *ACM SIGCOMM Conference* (2020).

[157] Marinov, D., and Khurshid, S. TestEra: A Novel Framework for Automated Testing of Java Programs. In *International Conference on Automated Software Engineering* (2001).

[158] Martonosi, M., Gupta, A., and Anderson, T. E. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *ACM SIGMETRICS Conference* (1992).

[159] Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S. D., Kidd, N., Kononov, R., Kumar, G., Mauer, C., Musick, E., Olson, L. E., Rubow, E., Ryan, M., Springborn, K., Turner, P., Valancius, V., Wang, X., and Vahdat, A. Snap: A Microkernel Approach to Host Networking. In *Symposium on Operating Systems Principles* (2019).

[160] Mauro, T. Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices, 2015. [Last accessed on 2023-06-26].

[161] Mehrotra, P., and Goswami, S. Analyzing Snort. Tech. rep., University of British Columbia, 2018.

[162] Mellanox ConnectX-4 Network Adapter Cards. https://downloadcenter.intel.com/download/14687. [Last accessed on 2023-06-26].

[163] Microscope Survey Form and Results. https://www.dropbox.com/s/66cp4k3wl8zm0q5/survey.pdf?dl=0. [Last accessed on 2023-06-26].

[164] Misra, P. A., Borge, M. F., Goiri, I., Lebeck, A. R., Zwaenepoel, W., and Bianchini, R. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *ACM European Conference on Computer Systems* (2019).

[165] Mogul, J. C., and Wilkes, J. Nines are Not Enough: Meaningful Metrics for Clouds. In *Workshop on Hot Topics in Operating Systems* (2019).

[166] Moon, S.-J., Helt, J., Yuan, Y., Bieri, Y., Banerjee, S., Sekar, V., Wu, W., Yannakakis, M., and Zhang, Y. Alembic: Automated Model Inference for Stateful Network Functions. In *Symposium on Networked Systems Design and Implementation* (2019).

[167] Naik, P., Shaw, D. K., and Vutukuru, M. NFVPerf: Online Performance Monitoring and Bottleneck Detection for NFV. In *IEEE Conference on Network Function Virtualization and Software Defined Networks* (2016).

[168] Performance Tests for Natasha. https://github.com/scaleway/natasha/tree/master/test/perf. [Last accessed on 2023-06-26].

[169] Scaleway Natasha. https://github.com/scaleway/natasha. [Last accessed on 2023-06-26].

[170] Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., and Wang, X. Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval. In *Symposium on Operating Systems Principles* (2019).

[171] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *Symposium on Operating Systems Principles* (2017).

[172] Network Functions Virtualization: Introductory White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf. [Last accessed on 2023-06-26].

[173] Norrie, T., Patil, N., Yoon, D. H., Kurian, G., Li, S., Laudon, J., Young, C., Jouppi, N. P., and Patterson, D. A. Google's Training Chips Revealed: TPUv2 and TPUv3. In *IEEE Hot Chips Symposium* (2020).

[174] Number of Daily E-Commerce Shipments. https://earlymoves.com/2017/03/21/amazon-3-million-packages-a-day-alibaba-12-million-packages-a-day. [Last accessed on 2023-06-26].

[175] Number of Sent and Received Emails per Day. https://www.statista.com/statistics/456500/daily-number-of-e-mails-worldwide. [Last accessed on 2023-06-26].

[176] Number of Daily Google Queries. https://www.internetlivestats.com/google-search-statistics. [Last accessed on 2023-06-26].

[177] Number of Active Internet Users. https://www.statista.com/statistics/617136/digital-population-worldwide. [Last accessed on 2023-06-26].

[178] Octane Benchmark for Javascript Engines. https://chromium.github.io/octane/. [Last accessed on 2023-06-26].

[179] Github: State of the Octoverse 2022 - Programming Languages. https://octoverse.github.com/2022/top-programming-languages. [Last accessed on 2023-06-26].

[180] Olivo, O., Dillig, I., and Lin, C. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *International Conference on Programming Language Design and Implementation* (2015).

[181] OpenSSL. https://github.com/openssl/openssl. [Last accessed on 2023-06-26].

[182] OpenSSL CVE-2018-0737. https://github.com/advisories/GHSA-rj52-j648-hww8. [Last accessed on 2023-06-26].

[183] Github issue raising constant-time violation in OpenSSL's Cipherblock Unpadding. https://github.com/openssl/openssl/issues/16230. [Last accessed on 2023-06-26].

[184] Pull request to fix constant-time violation in OpenSSL's Cipherblock Unpadding. https://github.com/openssl/openssl/pull/16323. [Last accessed on 2023-06-26].

[185] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Symposium on Networked Systems Design and Implementation* (2019).

[186] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., and Shenker, S. NetBricks: Taking the V out of NFV. In *Symposium on Operating Systems Design and Implementation* (2016).

[187] The PARSEC benchmark suite. https://parsec.cs.princeton.edu. [Last accessed on 2023-06-26].

[188] Pedrosa, L., Iyer, R., Zaostrovnykh, A., Fietz, J., and Argyraki, K. Automated Synthesis of Adversarial Workloads for Network Functions. In *ACM SIGCOMM Conference* (2018).

[189] Pesterev, A., Zeldovich, N., and Morris, R. T. Locating Cache Performance Bottlenecks Using Data Profiling. In *ACM European Conference on Computer Systems* (2010).

[190] Petsios, T., Zhao, J., Keromytis, A. D., and Jana, S. Slowfuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Conference on Computer and Communication Security* (2017).

[191] PHOENIX Benchmark Suite. https://github.com/kozyraki/phoenix. [Last accessed on 2023-06-26].

[192] Pirelli, S., Valentukonytė, A., Argyraki, K., and Candea, G. Automated Verification of Network Function Binaries. In *Symposium on Networked Systems Design and Implementation* (2022).

[193] Prekas, G., Kogias, M., and Bugnion, E. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Symposium on Operating Systems Principles* (2017).

[194] Primorac, M., Argyraki, K., and Bugnion, E. How to Measure the Killer Microsecond. In *SIGCOMM Workshop on Kernel-Bypass Networks* (2017).

[195] Python3 Documentation: Lists. https://docs.python.org/3/tutorial/datastructures.html. [Last accessed on 2023-06-26].

[196] Qin, H., Li, Q., Speiser, J., Kraft, P., and Ousterhout, J. K. Arachne: Core-Aware Thread Management. In *Symposium on Operating Systems Design and Implementation* (2018).

[197] Qiu, Y., Xing, J., Hsu, K.-F., Kang, Q., Liu, M., Narayana, S., and Chen, A. Automated SmartNIC Offloading Insights for Network Functions. In *Symposium on Operating Systems Principles* (2021).

[198] Ranganathan, P., Stodolsky, D., Calow, J., Dorfman, J., Hechtman, M. G., Smullen, C., Kuusela, A., Laursen, A. J., Ramirez, A., Wijaya, A. A., Salek, A., Cheung, A., Gelb, B., Fosco, B., Kyaw, C. M., He, D., Munday, D. A., Wickeraad, D., Persaud, D., Stark, D., Walton, D., Indupalli, E., Perkins-Argueta, E., Lou, F., Wu, H. K., Chong, I. S., Jayaram, I., Feng, J., Maaninen, J., Lucke, K. A., Mahony, M., Wachsler, M. S., Tan, M., Penukonda, N., Dasharathi, N., Kongetira, P., Chauhan, P., Balasubramanian, R., Macias, R., Ho, R., Springer, R., Huffman, R. W., Foss, S., Bhatia, S., Gwin, S. J., Sekar, S. K., Sokolov, S. N., Muroor, S., Rautio, V.-M., Ripley, Y., Hase, Y., and Li, Y. Warehouse-Scale Video Acceleration: Co-design and Deployment in the Wild. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2021).

[199] Ranney, M. Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories. http://highscalability.com/blog/2016/10/12/lessons-learned-from-scaling-uber-to-2000-engineers-1000-ser.html. [Last accessed on 2023-06-26].

[200] Ren, X. J., Rodrigues, K., Chen, L., Vega, C., Stumm, M., and Yuan, D. An Analysis of Performance Evolution of Linux's Core Operations. In *Symposium on Operating Systems Principles* (2019).

[201] Rogora, D., Carzaniga, A., Diwan, A., Hauswirth, M., and Soulé, R. Analyzing System Performance with Probabilistic Performance Annotations. In *ACM European Conference on Computer Systems* (2020).

[202] Ruan, Y., and Pai, V. S. Making the "Box" Transparent: System Call Performance as a First-Class Result. In *USENIX Annual Technical Conference* (2004).

[203] Saxena, P., Poosankam, P., McCamant, S., and Song, D. Loop-Extended Symbolic Execution on Binary Programs. In *International Symposium on Software Testing and Analysis* (2009).

[204] Schwartz, E. J., Avgerinos, T., and Brumley, D. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy* (2010).

[205] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Symposium on Networked Systems Design and Implementation* (2012).

[206] Sen, K., Marinov, D., and Agha, G. CUTE: a Concolic Unit Testing Engine for C. In *Symposium on the Foundations of Software Engineering* (2005).

[207] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. Making Middleboxes Someone Else's Problem: Network Processing as A Cloud Service. In *ACM SIGCOMM Conference* (2012).

[208] Shirokov, N., and Dasineni, R. Open-Sourcing Katran, a Scalable Network Load Balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer, 2018. [Last accessed on 2023-06-26].

[209] Shortest Remaining Processing Time (SRPT) Scheduling. https://en.wikipedia.org/wiki/Shortest_remaining_time. [Last accessed on 2023-06-26].

[210] Side Effects. https://en.wikipedia.org/wiki/Side_effect_(computer_science). [Last accessed on 2023-06-26].

[211] Service Level Objectives. https://sre.google/sre-book/service-level-objectives. [Last accessed on 2023-06-26].

[212] Smith, R., Estan, C., and Jha, S. Backtracking algorithmic complexity attacks against a NIDS. In *Annual Computer Security Applications Conference* (2006).

[213] Snort. https://www.snort.org. [Last accessed on 2023-06-26].

[214] Snowflake vs Databricks 2022 Benchmark Wars. https://www.linkedin.com/pulse/snowflake-vs-databricks-tpcs-ds-benchmark-wars-who-cares-jacobs. [Last accessed on 2023-06-26].

[215] Soares, L., and Stumm, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Symposium on Operating Systems Design and Implementation* (2010).

[216] Souyris, J., Pavec, E. L., Himbert, G., Borios, G., Jégu, V., and Heckmann, R. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Workshop on Worst-Case Execution Time Analysis* (2007).

[217] SPLASH-2 Benchmark Suite. https://github.com/staceyson/splash2. [Last accessed on 2023-06-26].

[218] Stoenescu, R., Popovici, M., Negreanu, L., and Raiciu, C. SymNet: Scalable Symbolic Execution for Modern Networks. In *ACM SIGCOMM Conference* (2016).

[219] Sutherland, M., Gupta, S., Falsafi, B., Marathe, V., Pnevmatikatos, D., and Daglis, A. The NeBuLa RPC-Optimized Architecture. In *International Symposium on Computer Architecture* (2020).

[220] Sutherland, M., Gupta, S., Falsafi, B., Marathe, V. J., Pnevmatikatos, D. N., and Daglis, A. The NEBULA RPC-Optimized Architecture. In *International Symposium on Computer Architecture* (2020).

[221] Tarjan, R. E. Amortized Computational Complexity. In *SIAM Journal on Algebraic Discrete Methods* (1985).

[222] TCP Offload Engine (TOE). https://www.chelsio.com/nic/tcp-offload-engine. [Last accessed on 2023-06-26].

[223] Terpstra, D., Jagode, H., You, H., and Dongarra, J. J. Collecting Performance Data with PAPI-C. In *Workshop on Parallel Tools for High Performance Computing* (2009).

[224] The Halting Problem in Computer Science. https://en.wikipedia.org/wiki/Halting_problem. [Last accessed on 2023-06-26].

[225] The Cost of Latency. https://perspectives.mvdirona.com/2009/10/the-cost-of-latency. [Last accessed on 2023-06-26].

[226] Tootoonchian, A., Panda, A., Lan, C., Walls, M., Argyraki, K. J., Ratnasamy, S., and Shenker, S. ResQ: Enabling SLOs in Network Function Virtualization. In *Symposium on Networked Systems Design and Implementation* (2018).

[227] The TPC-C OLTP benchmark. http://www.tpc.org/tpcc. [Last accessed on 2023-06-26].

[228] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy Transactions in Multicore In-Memory Databases. In *Symposium on Operating Systems Principles* (2013).

[229] Benchmarking Methodology for Networking Interconnect Devices. http://www.rfc-editor.org/rfc/rfc2647.txt. [Last accessed on 2023-06-26].

[230] Valgrind. https://valgrind.org. [Last accessed on 2023-06-26].

[231] Vanderwaart, C. J. Static Enforcement of Timing Policies Using Code Certification. http://reports-archive.adm.cs.cmu.edu/anon/2006/abstracts/06-143.html, 2006. [Last accessed on 2023-06-26].

[232] Wang, G., Chattopadhyay, S., Biswas, A. K., Mitra, T., and Roychoudhury, A. KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *Transactions on Software Engineering Methodology* (2020).

[233] Watson, A. H., and McCabe, T. J. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. Computer Systems Laboratory, National Institute of Standards and Technology, 1996.

[234] Worst Case Execution Time Analysis for Automotive Software. https://www.rapitasystems.com/automotive-software-testing. [Last accessed on 2023-06-26].

[235] MTU uses Worst Case Execution Time Tool aiT to Demonstrate Correctness of Control Software for Emergency Power Generators in Power Plants. https://www.absint.com/ait/slides/15.htm. [Last accessed on 2023-06-26].

[236] Software Considerations in Airborne Systems. https://www.rapitasystems.com/do178. [Last accessed on 2023-06-26].

[237] Wegbreit, B. Mechanical Program Analysis. In *Communications of the ACM* (1975).

[238] Wei, H., Yu, J. X., Lu, C., and Lin, X. Speedup Graph Processing by Graph Ordering. In *ACM SIGMOD Conference* (2016).

[239] Why Brands are Fighting over Milliseconds. https://www.forbes.com/sites/steveolenski/2016/11/10/why-brands-are-fighting-over-milliseconds. [Last accessed on 2023-06-26].

[240] Wierman, A., and Zwart, B. Is Tail-Optimal Scheduling Possible? In *Journal of Operating Research* (2012).

[241] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems* (2008).

[242] Wu, L., Barker, R. J., Kim, M. A., and Ross, K. A. Navigating Big Data with High-Throughput, Energy-Efficient Data Partitioning. In *International Symposium on Computer Architecture* (2013).

[243] Wu, W., He, K., and Akella, A. PerfSight: Performance Diagnosis for Software Dataplanes. In *Internet Measurement Conference* (2015).

[244] Wu, W., Zhang, Y., and Banerjee, S. Automatic Synthesis of NF Models by Program Analysis. In *ACM Workshop on Hot Topics in Networks* (2016).

[245] x86 Support for User Interrupts. https://lwn.net/Articles/869140/. [Last accessed on 2023-06-26].

[246] Express data path. https://prototype-kernel.readthedocs.io/en/latest/networking/XDP. [Last accessed on 2023-06-26].

[247] Xie, X., Chen, B., Liu, Y., Le, W., and Li, X. Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. In *Symposium on the Foundations of Software Engineering* (2016).

[248] Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C. W., and Torrellas, J. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy . In *International Symposium on Microarchitecture* (2019).

[249] Youtube Video Streaming Statistics. https://www.comparitech.com/tv-streaming/youtube-statistics. [Last accessed on 2023-06-26].

[250] Z3 Python API. https://github.com/Z3Prover/z3/blob/master/src/api/python/z3/z3.py. [Last accessed on 2023-06-26].

[251] Zaostrovnykh, A., Pirelli, S., Iyer, R. R., Rizzo, M., Pedrosa, L., Argyraki, K. J., and Candea, G. Verifying Software Network Functions with No Verification Expertise. In *Symposium on Operating Systems Principles* (2019).

[252] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K., and Candea, G. A Formally Verified NAT. In *ACM SIGCOMM Conference* (2017).

[253] Zaparanuks, D., and Hauswirth, M. Algorithmic Profiling. In *International Conference on Programming Language Design and Implementation* (2012).

[254] Zarandi, A. P., Sutherland, M., Daglis, A., and Falsafi, B. Cerebros: Evading the RPC Tax in Datacenters. In *International Symposium on Microarchitecture* (2021).

[255] Zeng, H., Kazemian, P., Varghese, G., and McKeown, N. Automatic Test Packet Generation. In *International Conference on Emerging Networking Experiments and Technologies* (2012).

[256] Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O. S. N., Martinez, A., Liu, J., Simpson, A. K., Jayakar, S., Penna, P. H., Demoulin, M., Choudhury, P., and Badam, A. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Symposium on Operating Systems Principles* (2021).

[257] Zhong, Y., Orlovich, M., Shen, X., and Ding, C. Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity. In *International Conference on Programming Language Design and Implementation* (2004).

[258] Zhou, D., Yu, H., Kaminsky, M., and Andersen, D. G. Fast Software Cache Design for Network Appliances. In *USENIX Annual Technical Conference* (2020).