Enabling Hyperscale Web Services

by

Akshitha Sriraman

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2021

Doctoral Committee:

Professor Thomas F. Wenisch, Chair
Professor David Brooks, Harvard University
Assistant Professor Jean-Baptiste Jeannin
Assistant Professor Baris Kasikci
Professor Margo I. Seltzer, University of British Columbia

Akshitha Sriraman

akshitha@umich.edu

ORCID iD: 0000-0003-4780-9483

To the ones who raised me to believe that anything is possible, Sriraman and Rajalakshmi.

To the one who made many things possible, Akshay.

And to the one who makes everything possible, Amrit.

ACKNOWLEDGEMENTS

My Ph.D. experience was a roller coaster: ups and downs, twists and turns. But boy, was it a ride! For all the marvelous times when I built large systems that somehow just worked, had papers accepted, and won fellowships, there were just as many, if not more, challenging times when I built large systems that somehow just *didn't* work, floundered with no clue about what I was doing, and struggled with imposter syndrome. I express my deepest gratitude to the village of phenomenal people who made the challenging times conquerable and the marvelous memorable, *enabling a Ph.D.* that I didn't think would happen.

`AMMA & APPA.` Starting with the people who have *always* been there, I thank my parents, my Amma and Appa, **Rajalakshmi Sriraman** and **P.R. Sriraman**. Rather, I will *attempt* to thank them, because I know that words will always fall short in this case.

My father aspired to be an electrical engineer, but his chance of realizing his dreams perished when he lost his father at a young age. I thank you, Appa, for teaching me the gratification of sincere efforts—you never gave up, endured a job that kept you far from home, and pursued an advanced degree well into adulthood. Learning to persevere like you has helped me come a long way. Appa, I thank you for the seemingly small things that made the biggest difference—for setting alarms to wake up at odd hours to then wake *me* up with a coffee in hand before my deadlines, for staying up on Skype for hours on end to “keep me company” when I was working and was feeling homesick, for always reading every word I write, and for texting me *every single day* to tell me you love me. Thank you for your value of education and for always encouraging me to push boundaries. Because of you, I have become the person I always dreamed of being. I hope that makes you proud. I thank you for being my inspiration—wanting to make you proud is the fire that fuels me to revive the dreams you once dreamed, by living them. *Appa, this Ph.D. is for you.*

As my first teacher, I thank you, Amma, for instilling in me from an early age the idea that anything is possible, any dream attainable. Today, several of my dreams have come true. And I want you to know that it is because of you. I thank you for believing in me even at times when I did not believe in myself. I will forever be indebted to you for all that I am today. You told me that you do not understand the meaning behind several technical details in my dissertation. While that may be true, I want you to know that this dissertation is *meaningless* without you. *And for this reason, Amma, I dedicate this dissertation to you.*

அம்மா மற்றும் அப்பா, நான் இந்த ஆய்வுக் கட்டுரையை அன்போடு உங்களுக்கு அர்ப்பணிக்கிறேன்.

AKSHAY SRIRAMAN. I thank the best brother I could ask for, Akshay Sriraman. In the past, when I had trouble figuring out who I wanted to be, I simply followed Akshay without question. I followed him when I decided to pursue engineering and I followed him again when I decided to study electronics. I knew that if I did what he did, I would do great.

I thank Akshay for putting me before himself *so* many times. He sacrificed his goals of getting a Master’s degree because he knew that I wanted to pursue one. So, instead, he took up a job to monetarily back my (extremely expensive) Master’s education at UPenn[1]. These sacrifices, like a skyscraper’s foundation, are the kinds of things that seemingly go unseen, but actually form the very cornerstone of a person’s achievements. I don’t think I

[1] As the picture suggests, Akshay did eventually get a Master’s degree, also from the University of Pennsylvania. So, perhaps, *he* followed *me* this time around?

would be writing this dissertation if not for him. How else could I have traveled to the US? Pursued further education? Received my first exposure to research and decided on a Ph.D.? Recognized that my dream is to become a professor? And later go on to achieve that dream? I start at Carnegie Mellon University in January 2022, and I am forever indebted to Akshay for enabling the cascading set of events that brought me to this point.

JOSEPH DEVIETTI. I will forever be grateful to Joe Devietti, my Master's advisor at UPenn, who started it all. Joe took me on when *nobody* would have given me a second glance, and also despite me not doing great in his course[2]. I thank him for believing that I could do research; his belief made a world of difference. I recall telling him he was making a mistake—although I wanted to try doing research, I didn't think I could. He said that I was hard working, motivated, and passionate, and those are the only things I would need. Thank you, Joe, for showing me what *truly* matters—every time I felt overwhelmed due to my lack of prior CS education, your words kept me going. Your faith in me made me push myself harder every day—I didn't want you to be wrong and I just didn't want to let you down.

When I struggled with visualizing software, Joe pointed out how I could use my strength of networking with people to learn from others about how *they* might build the software I was working on. Joe, Chapter III of this dissertation, my biggest software effort, is for you. Thank you for identifying my strengths; knowing, and using them to my advantage has made me come a long way. But most importantly, I thank you for giving meaning to my entire life and career. You are the reason I dreamed of becoming a professor—if I can pass on what you did for me to at least one other student, I shall declare my life *wildly* successful.

THOMAS WENISCH. When I moved from Penn to Michigan, I didn't initially work with my Ph.D. advisor, Tom. I thank Tom for taking me on from another lab (resulting in me naming our research group "The Sanctuary Lab"). That single decision changed the entire course of my Ph.D., making it into a more fabulous experience that I ever dreamed possible.

Tom is famous for (amongst other things, e.g., a Maurice Wilkes award) saying the

[2] This happened to be a graduate course on *computer architecture*. Ironic, isn't it? Goes to show that grades don't really define the career you can go on to have.

words: *“I already have a PhD. I don’t have a need to get another PhD via you”*. Although these words might seem like a joke at first blush, I realize that they summarize how Tom shaped me as a researcher. Tom did everything to make me an independent researcher, so that I could define my Ph.D. in a way that was uniquely me. From guiding me every step of the way when he knew I was utterly lost, to taking a back seat when he knew I was ready, I thank Tom for doing it all in the most beautiful way imaginable.

Tom knows me as a researcher even better than I know myself. He encouraged me to independently see a project through and write a paper about it (see Chapter VI); *“We are taking the training wheels off,”* he said. Although I felt unprepared initially, I emerged much more confident in my abilities and much closer towards winning my battle against imposter syndrome. I will always maintain that “Tom knows everything about everything.” Tom, I know that if I become at least half as great an advisor as you, I will be *ridiculously* happy.

DOCTORAL COMMITTEE MEMBERS. **Margo Seltzer**, for taking me under her wing and being such a fabulous role model to me. Having Margo as my mentor exemplifies the notion that sometimes the most incredible things in life happen by accident. She leads by example and has always gone above and beyond whenever I needed help, be it technical, professional, or personal. I thank Margo for reading every single word of this dissertation and I thank her for always accepting nothing less than my absolute best. I am incredibly lucky to have Margo as my mentor. *Everyone needs a Margo in their life.*

Baris Kasikci, for being such a constant pillar of support. The past couple of years of working with Baris and EFES Lab members have truly helped me grow and thrive as a researcher. I am so grateful to Baris for everything he does: from letting me know whether an email draft looks okay, to helping me with the job search process, Baris has been there for it all. I thank him for being the perfect mix of a mentor who is also a friend.

David Brooks, for giving an opportunity to mentor Ph.D. students when I was a Ph.D. student myself. And **Jean-Baptiste Jeannin**, for the valuable feedback on my work.

MENTORS and COLLABORATORS. I thank **Sarita Adve** for being my biggest role model. I thank her for leading by example in teaching me how to be a tough cookie at times when I encounter bias. I am grateful to **Peter Chen** for always insisting that I think about a research problem from first principles and **Christos Kozyrakis** for teaching me how to write grant proposals. I thank all three of them for helping me land my dream job.

I thank **Reetu Das** for reaching out to me at a time when I was thinking about quitting the Ph.D. program, for her kindness, and for our "lunch dates" every semester. I thank **Kim Hazelwood** for recruiting me for a Facebook internship that opened a world of research possibilities, and for helping me land my dream job. When I was concerned about how my research approach didn't always fit the mould (e.g., introducing humour, or rather my version of humour, in my talks), Kim told me something that I will always remember: *"You are a unicorn. Don't let people chop off your horn and convert you into yet another sheep."*

I am grateful to **Abhishek Dhanotia**, my Facebook internship mentor, for his research advice during multiple Facebook internships. I thank him for showing me how much fun research can be when one is not constantly worrying about the next paper deadline. My time at Facebook was always intellectually enriching because of stimulating conversations with so many people who were merely a ping away. I especially thank **Vijay Balakrishnan**, **Carlos Torres**, **Pallab Bhattacharya**, **Hsien-Hsin Lee**, **Carole-Jean Wu**, **Xiaodong Wang**, **Jonathan Haslam**, **Denis Sheahan**, **Jay Kamat**, and **Shobhit Kanaujia** for helping me understand production systems. I thank Facebook for granting me access to real-world data centers, significantly enhancing my dissertation's contributions. I also thank Facebook for the free food during my internships (especially for the best boba and ramen I've ever had)!

I thank **Ed Nightingale** and **Galen Hunt**, my MSR internship mentors, for teaching me to never take no for an answer because there will always be a workaround if I looked hard enough. I am grateful to many MSR researchers who helped me create a concrete research vision at a time when I was floundering. I especially thank **Ricardo Bianchini** for helping me realize μTune; *"The moment you have an inflection point, you have a story,"* he said.

Gilles Pokam and **Rajeev Alur**, for helping me get into one of the best Ph.D. programs. **Rajagopalan Nadathur**, without whom I couldn't have moved to the US and started at Penn. **James Mickens**, **Abhishek Bhattacherjee**, **Brandon Reagen**, and my **WICArch** friends for all their advice. In loving memory of my English teacher, **Ms. Stephens**, without whom I couldn't have authored my papers. I wish I could share them with her.

SANCTUARY LAB. A huge shout-out to **Aasheesh Kolli** and **Neha Agarwal** for always encouraging me. I thank **Amlan Nayak**, **Brendan West**, **Hossein Golestani**, **Harini Muthukrishnan**, **Steve Zekeny**, **Ofir Weisse**, and **Amir Mirhosseini** for all the technical discussions, but more importantly, for being my partners in crime when pulling unicorn-themed pranks on Tom. (Remember the time we TP'd Tom's office with prancing unicorn-themed toilet paper?) All of you made graduate school memorable and fun.

But, this Ph.D. wouldn't have happened without **Vaibhav Gogte**. From teaching me how to write code (yes, that indeed happened since I entered a Ph.D. program in *CS* without prior programming education—smh), to spending long hours entertaining my (half-baked) ideas, Vaibhav has been my greatest source of inspiration. I thank him for being my go-to (inside joke: more like "gogo") person for *everything*. Vaibhav exemplified an ideal Ph.D. student by living his own quote: *"Your Ph.D. is just another job. It is not a higher calling."* His words stuck with me and made graduate school so much more than just about publishing papers. I thank him for making graduate school the most valuable experience of my life.

EFES LAB. A huge shout-out to **Tanvir Khan**, **Kevin Loughlin**, **Andrew Quinn**, **Marina Minkin**, **Sara Shahri**, **Ian Neal**, **Jiacheng Ma**, and **Gefei Zuo**—for tirelessly attending my many (bad) practice talks. I thank them for making me one of their own and for being my new research home. I thank Tanvir for making me look forward to being a professor.

`MICHIGAN CSE.` I thank **Manos Kapritsos** for caring so deeply about every student. **Mahdi Cheraghchi**, **Mark Guzdial**, **John Laird**, and **Emily Provost**, for helping me at key phases. And **Karen Liska**, **Ashley Andrea**, **Stephen Reger**, **Jamie Goldsmith**, **Laura Fink**, **Sylvia Galaty**, **Cindy Estell**, and **Erika Hauff**, for being CSE's true heroes.

`GRANTS.` My work was supported by (1) the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA and (2) NSF Grant IIS1539011 and gifts from Intel. I also thank Facebook for a fellowship that allowed me to freely explore new research directions.

`FRIENDS.` I thank my best friend, **Amritha Varshini**, for being the sister I never had.

Thank you for always having my back. Thank you for being the loving, encouraging, and nonjudgemental person that you are, all wrapped into one small, amazing package. When I started at Penn, I had no idea that I would come away with the greatest gift of my life.

A big shout-out to the Juwaris group—**Kumar Aanjaneya**, **Aditi Kulkarni**, **Sneha Joshi**, **Shriya Sethuraman**, **Poorwa Shekhar**, **Ripudaman Singh**, **Niket Prakash**, **Harshad Dharmatti**, **Kunal Garg**, **Saurabh Mahajan**, **Bharadwaj Mantha**, and **Keval Ramani**—for the boisterous board game nights that helped me stay sane during the dark times. Meeting **Zahra Tarkhani** at MSR made me realize that soulmates are real—our similarities are too strong to be a coincidence; but, perhaps, building a bare-metal hypervisor from scratch, together, does that to you? Thank you, Zahra, for the laughs and the crazy, impromptu drives. I thank **Nilmini Abeyratne**, **Sai Gouravajhala**, **Amir Rahmati**, and **Earlence Fernandes** for helping me move to Sanctuary Lab. **Animesh Jain**, for the walk along the Sammamish river, where μTune took a concrete form for the first time. **Vidushi Goyal**, for all our "discussions" over chai. **Caroline Trippel**, for being such a trailblazer and passing on advice on how to become one. **Tumpa Chakraborty**, for all her support.

RELATIVES. In loving memory of my grandpa, **Thathappa**, who was the first person to tell me that I should do a Ph.D., introducing the word to me. I miss you, Thathappa, and I wish I could share this dissertation with you. I thank my grandma, **Momma**, for being the sweetest person I've ever met. In loving memory of my grandma, **Radha paatti**. I hope she would be proud. I never got to meet my grandpa, but I hope he would be proud too. I thank my extended parents, **Anandhi** and **Gopal**, for all their love and for always being there; they are my life's most beautiful gifts. I thank **Shruthi** and **Abhinaya** for their love.

When I first moved to the US, I was terribly homesick and wanted to move back to India in a grand total of three days. I thank **Ranga uncle**, **Anu aunty**, **Sumun**, **Pranav**, and **Leo** for everything they did to make Philly my home away from home. I am also very grateful to **Krishnan uncle** and **Shankar uncle** for helping me navigate the US academic system.

PO GOPAL. I thank Po, my oh-so-adorable puppy. Po is my lucky charm: my dissertation work took off only after he entered my life. Po sat with me (more like slept near me) through every deadline and was my rubber duck (dog, rather) for debugging. Thank you, Po, for being fluffy, for being silly, and for your (literally) warm presence that kept me company.

AMRIT GOPAL. Saving the best for last, to my husband, Amrit, a few private words addressed to you in public. Thank you for your love, patience, and unwavering support

towards my growth—you have always been my rock. From spending numerous hours critiquing my work, to squandering your vacation days traveling to conferences so that you could sit in the front row and record my talk (like some kind of a proud parent), you have done it all. Amrit, you are a true partner, and I feel truly blessed that I get to have you by my side. During the past six years, I felt a lot like I did just the front-end UI development work, when you were the back-end data center holding the fort together.

Amrit, this Ph.D. is as much yours as it is mine.

And finally, Amrit, Amma, and Appa, you know the past few years were not easy:

When I felt so lost, like I'd lost my way,
When I didn't know what to do or say,
When I didn't know if my research was sound,
When I was floundering all around,
When my research vision was rare to see,
When I didn't know what I'd grow to be,
When paper acceptances were rare,
When Reviewer C just didn't care,
When I had no clue and felt very blue,
I knew I could always turn to you.
I knew you'd always be there.
I knew you'd always care.
All I'd do was close my eyes and think of you
And I knew I'd come through.

"The city of Pittsburgh gleaming suddenly before her . . .
so startling in its vastness and its beauty that she had gasped and slowed,
afraid of losing control of the car."
Kim Edwards

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES xvi

LIST OF TABLES xxi

ABSTRACT xxii

CHAPTER

I. Introduction 1

1.1 Motivation 1
1.1.1 Web Services Powered by Data Centers are Here, There, and Everywhere 1
1.1.2 Radical Shift in Hyperscale Computing 3
1.1.3 Decline in Technology Trends that Drive Processor Performance Scaling 4
1.1.4 Consequences on the Software and Hardware Research Landscape 5
1.2 Research Challenges and Goals 12
1.2.1 Enabling the Study of Modern Web Services 14
1.2.2 Redesigning Software Based on Underlying Data Center Hardware Constraints 16
1.2.3 Architecting Commodity Hardware for New Web Service Software Paradigms 18
1.2.4 Architecting Custom Hardware for New Web Service Software Paradigms 22
1.3 Dissertation Contributions 25
1.4 Dissertation Outline 29

II. A Benchmark Suite for Microservices 31

2.1 Prior Work 34
2.2 μSuite: Benchmarks Description 35
2.2.1 HDSearch 36
2.2.2 Router 40
2.2.3 Set Algebra 43
2.2.4 Recommend. 45
2.3 μSuite: Framework Design 49
2.4 Methodology 51
2.5 Results 52
2.5.1 Saturation Throughput 52
2.5.2 End-to-end Response Latency 53
2.5.3 OS and Network Overheads 54
2.6 Long-Term Impact Potential 59
2.7 Chapter Summary 61

III. Auto-Tuned Software Threading for Microservices 63

3.1 Motivation 66
3.1.1 The Need for a Threading Model Taxonomy 66
3.1.2 The Need for Automatic Load Adaptation 67
3.1.3 A Microservice Framework 68
3.2 A Taxonomy of Threading Models 68
3.2.1 Key Dimensions 69
3.2.2 Synchronous Models 71
3.2.3 Asynchronous Models 73
3.3 μTune: System Design 75
3.3.1 Framework 76
3.3.2 Automatic Load Adaptation 76
3.4 Implementation 78
3.5 Experimental Setup 79
3.6 Evaluation 80
3.6.1 Threading Model Characterization 81
3.6.2 Load Adaptation 89
3.7 Discussion 96
3.8 Related Work 97
3.9 Long-Term Impact Potential 98
3.10 Follow-On Research 101
3.11 Chapter Summary 102

IV. Optimizing Commodity Server Architectures for Microservice Diversity at Hyperscale 105

4.1 Understanding Microservice Performance 108
4.1.1 The Production Microservices 109

4.1.2 Characterization Approach 110
4.1.3 System-Level Characterization 111
4.1.4 Architectural Characterization 116
4.2 "Soft" SKU 124
4.3 μSKU: System Design 126
4.4 Methodology 128
4.5 Evaluation 131
4.5.1 Knob Characterization 131
4.5.2 Soft SKU Performance 136
4.6 Discussion 137
4.7 Related Work 138
4.8 Long-Term Impact Potential 139
4.9 Follow-On Research 142
4.10 Chapter Summary 145

V. Redesigning Commodity Server Architectures for Efficient Event Notification at Hyperscale 148

5.1 Why do Widely-Used Notification Paradigms Fall Short? 151
5.1.1 Microservice Requirements 151
5.1.2 Interrupts 152
5.1.3 Spin-polling 155
5.1.4 MWAIT 158
5.2 The μNotify Paradigm 159
5.2.1 Design Goals 159
5.2.2 System Model 160
5.2.3 μNotify Overview 161
5.2.4 μNotify's Back-end Microarchitecture 163
5.2.5 μNotify's Front-end Programming Model 165
5.3 Evaluation 168
5.3.1 Experimental Setup 168
5.3.2 Microservices 169
5.3.3 Peak Throughput 170
5.3.4 Queue Scalability 172
5.3.5 Median and Tail Latency 173
5.3.6 Overheads 174
5.4 Discussion 174
5.5 Related Work 175
5.6 Chapter Summary 176

VI. Understanding Hardware Customization Opportunities at Hyperscale 179

6.1 Understanding Microservice Overheads 183
6.1.1 The Production Microservices 183
6.1.2 Characterization Approach 184

6.1.3 Leaf Function Characterization 185
6.1.4 Service Functionality Characterization 193
6.2 The Accelerometer Model 196
6.2.1 Acceleration Strategies 197
6.2.2 System Abstraction 198
6.2.3 Parameter Definition 199
6.2.4 Modeling Diverse Threading Designs 199
6.2.5 Accelerometer Use Cases 204
6.3 Validating the Accelerometer Model 205
6.3.1 Validation Methodology 205
6.3.2 Experimental Setup 205
6.3.3 Case Study 1: AES-NI for Cache1 206
6.3.4 Case Study 2: Encryption for Cache3 208
6.3.5 Case Study 3: Inference for Ads1 209
6.4 Applying the Accelerometer Model 210
6.4.1 Compression 211
6.4.2 Memory Copy 213
6.4.3 Memory Allocation 213
6.5 Related Work 213
6.6 Long-Term Impact Potential 215
6.7 Chapter Summary 218

VII. Future Work and Conclusions 221

7.1 Future Directions 221
7.1.1 Enabling Cross-Stack Designs for Emerging Web Service Paradigms and Application Domains 221
7.1.2 Rethinking Hardware-Software Co-Design for System Overheads that Arise at Hyperscale 222
7.1.3 Mitigating the Killer Microsecond Problem in Modern Web Services 223
7.1.4 Redesigning Software Stacks for Emerging Hardware Accelerators 223
7.1.5 Designing Systems to Support Emerging Device Technologies 224
7.1.6 Using Machine Learning to Self-Navigate the Hyperscale Design Space 225
7.1.7 Designing Energy-Efficient Data Centers 225
7.1.8 Making Intersectionality, Equity, and Fairness as First-Order System Design Metrics 226
7.2 Dissertation Conclusions 227

BIBLIOGRAPHY 233

LIST OF FIGURES

Figure

1.1 A timeline of the work presented in this dissertation, organized horizontally according to the years when portions of the individual dissertation chapters were published. The timeline is divided into this dissertation's software and hardware thrusts to show a bird's eye view of the software and hardware design space studied in this dissertation. Each work of research is annotated with the dissertation chapter in which it is covered and the publication venue where portions of that chapter were published. The background color of the box representing each work of research is color coordinated with the levels of the systems stack (shown on the right) that the technologies presented in that chapter cover. 25

2.1 A typical web application fan-out: Modern web services are composed of a complex web of microservices that interact via RPCs. 33

2.2 HDSearch: Front-end presentation microservice. 37

2.3 HDSearch: Back-end request and response pipelines. 38

2.4 An example of HDSearch's request (left) and 1-NN response (right): Response's highlighted circular segment illustrates why the images match. 40

2.5 Router: Back-end request and response pipelines. 42

2.6 Set Algebra: Back-end request and response pipelines. 45

2.7 Recommend: Back-end request and response pipelines. 48

2.8 μSuite's mid-tier microservice design: μSuite's mid-tier microservices *block* on the front-end network socket, *dispatch* processing of front-end requests, and *asynchronously* communicate with leaf microservices. . . . 49

2.9 Saturation throughput (QPS): μSuite is similar to real-world web services. 52

2.10 End-to-end response latency across different loads for each benchmark: Median latency is higher at low load. 53

2.11 HDSearch's counts of OS system call invocations per QPS: The *futex* system call is predominantly invoked. 54

2.12 Router's counts of OS system call invocations per QPS: The *futex* system call is predominantly invoked. 55

2.13 Set Algebra's counts of OS system call invocations per QPS: The *futex* system call is predominantly invoked. 55

2.14 Recommend's counts of OS system call invocations per QPS: The *futex* system call is predominantly invoked. 55
2.15 HDSearch's breakdown of OS overheads: Time to switch a thread from the *active* to the *running* state is high. 56
2.16 Router's breakdown of OS overheads: Time to switch a thread from the *active* to the *running* state is high. 57
2.17 Set Algebra's breakdown of OS overheads: Time to switch a thread from the *active* to the *running* state is high. 58
2.18 Recommend's breakdown of OS overheads: Time to switch a thread from the *active* to the *running* state is high. 59
2.19 Context switches (CS) and thread contention (HITM) incurred (in millions) for each benchmark across diverse loads: Thread contention is a significant overhead. 59
3.1 A typical web application fan-out. 64
3.2 $99^{th}\%$ tail latency for an RPC handled by a block-based and poll-based model: Poll-based model reduces latency by 1.35x at low load, and saturates at high load. 67
3.3 Execution of an RPC by (a) SIB/SIP (b) SDB/SDP. 72
3.4 Execution of an RPC by (a) AIB/AIP (b) ADB/ADP. 74
3.5 μTune: System design. 75
3.6 Synchronous vs. asynchronous model's saturation throughput: The asynchronous model performs better by 42% on average. 81
3.7 Ratio of the best synchronous model's latency to the best asynchronous model's latency: The best asynchronous model is faster by a mean 12% at the loads that are achievable by the best synchronous model, and infinitely faster at higher loads. 82
3.8 Graph: Latency vs. load trade-off for `HDSearch`'s synchronous models. Table: Latencies at each load normalized to the best latency for that load: No threading model is always the best. 83
3.9 `HDSearch` synchronous thread wakeups at 64 QPS: Block incurs more wakeups. 85
3.10 Relative frequency of synchronous contention, context switches, and cache misses at 10K QPS: SIP performs the worst. 86
3.11 Graph: Latency vs. load for `Set Algebra`'s asynchronous models. Table: The latency at each load level normalized to the best latency for that load: No threading model is always the best. 87
3.12 Asynchronous thread pools for best tail latency: Big thread pools content. 88
3.13 Asynchronous `Set Algebra`'s relative frequency of contention, context switches, and cache misses over the best asynchronous model at peak load: AIP performs worst. 88
3.14 Synchronous (top) and asynchronous (bottom) steady-state adaptation. . . 92
3.15 Synchronous *μTune*'s instruction overhead for steady-state loads: Less than 5% mean overhead incurred. 93
4.1 Variation in system-level & architectural traits across microservices: Facebook's microservices face extremely diverse bottlenecks. 106

4.2 (a) A single request's latency breakdown for each μservice: Few μservices block for a long time, (b) Web's request latency breakdown: Thread oversubscription causes scheduling delays. 113
4.3 Max. achievable CPU utilization in user- and kernel-mode across μservices: Utilization can be low to avoid QoS violations. 114
4.4 Fraction of a second spent context switching (range): Cache1 & Cache2 can benefit from context switch optimizations. 115
4.5 Instruction type breakdown across microservices: Instruction mix ratios vary substantially across microservices. 116
4.6 Per-core IPC across Facebook's microservices and prior work (IPC measured on other platforms): Facebook's microservices have a high IPC diversity. 117
4.7 Top-down bottleneck breakdown: Several of Facebook's microservices face high front-end stalls. 118
4.8 L1 & L2 code & data MPKI: Facebook's microservices typically have higher L1 MPKI than comparison applications. 120
4.9 LLC code & data MPKI: LLC data MPKI is high across microservices and Web incurs a high code LLC MPKI. 121
4.10 LLC code and data MPKI vs. LLC size: Some microservices may benefit from trading LLC capacity for more cores. 121
4.11 I-TLB & D-TLB (load & store) MPKI breakdown: Some microservices can benefit from huge page support. 122
4.12 Memory bandwidth vs. latency: Microservices under-utilize memory bandwidth to avoid latency penalties. 123
4.13 *μSKU*: System design. 125
4.14 Performance trend with (a) core frequency scaling, (b) uncore frequency scaling: The maximum frequency offers the best performance. 131
4.15 Performance trend with core count scaling: Web is core-bound. 133
4.16 Performance trend with CDP scaling: (a) Web (Skylake) & Ads1 benefit due to lower code MPKI (b) Web (Broadwell) has no gains. 133
4.17 Performance trends with varied prefetcher configurations: Turning off prefetchers can improve bandwidth utilization in Web (Broadwell). 134
4.18 Performance trends with varied (a) THP: Web (Skylake) benefits from THP ON, (b) SHP: There is a sweet spot in optimal SHP count. 135
4.19 Performance gain with *μSKU* over stock and hand-tuned servers: *μSKU* outperforms even hand-tuned production servers. 136
5.1 (a) Epoll latency with increasing load: Interrupts face μs-scale context switches & thread wakeups. (b) Thread wakeups at low & high load: Low-load wakeups are costlier. 154
5.2 (a) Peak throughput when spin-polling many queues: Throughput reduces due to empty queue checks. (b) Latency when polling many queues: Long loop traversals dominate. 157
5.3 High-level system model. 160
5.4 High-level diagram of *μNotify*'s operations. 161
5.5 High-level cache coherence controller with *μNotify*. 164

5.6 Peak throughput achieved by *μNotify* compared to spin-polling and interrupts across different queue counts and traffic shapes for LSH: *μNotify* consistently achieves a higher throughput compared to state-of-the-art paradigms. 170
5.7 Peak throughput achieved by *μNotify* compared to spin-polling and interrupts across different queue counts and traffic shapes for McRouter: *μNotify* consistently achieves a higher throughput compared to state-of-the-art paradigms. 170
5.8 Peak throughput achieved by *μNotify* compared to spin-polling and interrupts across different queue counts and traffic shapes for Word Stemming: *μNotify* consistently achieves a higher throughput compared to state-of-the-art paradigms. 171
5.9 Peak throughput achieved by *μNotify* compared to spin-polling and interrupts across different queue counts and traffic shapes for Recommend: *μNotify* consistently achieves a higher throughput compared to state-of-the-art paradigms. 171
5.10 *μNotify*'s latency under light traffic with increasing queues (Y-axis is log-scale): *μNotify* achieves lower latency. 172
5.11 *μNotify*'s latency with increasing load compared to state-of-the-art: *μNotify* sustains higher load with low latency. 174
6.1 Breakdown of cycles spent in core application logic vs. orchestration work: Orchestration overheads can significantly dominate. 181
6.2 Breakdown of cycles spent in leaf functions: Memory functions consume a significant portion of total cycles. 186
6.3 Breakdown of cycles spent in memory leaf functions as a fraction of total cycles: Memory copy, allocation, & free consume significant cycles. . . . 188
6.4 Breakdown of service functionalities that invoke memory copies: There is significant diversity in dominant functionalities that perform copies. . . . 189
6.5 Breakdown of cycles spent in various kernel leaf functions: Kernel scheduler, event handling, and network overheads can be high. 190
6.6 Breakdown of CPU cycles spent in synchronization functions: `Cache` frequently uses spin locks to avoid thread wakeup delays. 191
6.7 Breakdown of CPU cycles spent in C libraries: ML services perform several vector operations while dealing with large feature vectors. 192
6.8 Cache1's IPC scaling across three CPU generations for key leaf functions: Kernel IPC is typically low & scales poorly. 193
6.9 Breakdown of CPU cycles spent in various microservice functionalities: Orchestration overheads are significant & fairly common. 194
6.10 Cache1's IPC scaling across three CPU generations for key functionality categories: A low I/O IPC is primarily due to a low kernel IPC. 196
6.11 Example timeline of host & accelerator. 198
6.12 Modeling `Sync` C_S and C_L for one offload. 200
6.13 Modeling `Sync-OS` C_S and C_L for one offload. 202
6.14 Modeling `Async` C_S and C_L for one offload. 203
6.15 CDF of bytes encrypted in Cache1: <512B are frequently encrypted. . . 207

6.16 Breakdown of cycles spent in Cache1's functionalities for both the no-AES-NI (unaccelerated) & with-AES-NI (accelerated) cases: 12.8% of cycles are freed up with AES-NI. 207

6.17 Breakdown of cycles spent in Cache3's functionalities when encryption is accelerated vs. not: Secure IO calls are optimized with acceleration. . . . 208

6.18 Breakdown of cycles spent in Ads1's functionalities for both the inference unaccelerated & accelerated cases: All inference cycles are freed up. . . . 209

6.19 CDF of bytes compressed in `Feed1` and `Cache1`: `Feed1` often compresses large granularities. 211

6.20 *Accelerometer*-estimated speedup for key overheads we identified: Performance bounds from accelerator offload limit achievable speedup. 212

6.21 CDF of memory copies across microservices: Most microservices frequently copy small granularities. 212

6.22 CDF of memory allocations across microservices: Most microservices frequently allocate small granularities. 213

LIST OF TABLES

Table

1.1 Timeline of the application layer's evolution in response to the unprecedented growth in hyperscale web service trends and today's hardware reality: There has been a shift from monolithic web application architectures to more granular architectures such as microservices. 6

1.2 Timeline of the evolution of software abstraction layers in response to the unprecedented growth in hyperscale web service trends and today's hardware reality: There has been a shift from heavy-weight abstractions (e.g., virtualization) to light-weight abstraction layers (e.g., containers). . 9

1.3 Timeline of the hardware layer's evolution in response to the unprecedented growth in hyperscale web service trends and today's hardware reality: There has been a shift towards building specialized hardware for various "killer" web applications. 11

2.1 Summary of a comparison of μSuite with prior works: Unlike prior works, μSuite is open-source, has web services composed of microservices, and enables the study of mid-tier microservices. 35

2.2 Mid-tier microservice processor specification. 51

3.1 Mid-tier microservice processor specification. 80

3.2 99^{th}% tail latency (ms) for load transients. 95

4.1 `Skylake18`, `Skylake20`, `Broadwell16`'s key attributes. 110

4.2 Average request throughput, request latency, & path length across microservices: We observe great diversity across services. 112

4.3 Summary of findings and suggestions for future optimizations. 124

6.1 GenA, GenB, and GenC CPU platforms' attributes. 184

6.2 Categorization of leaf functions. 185

6.3 Categorization of microservice functionalities. 193

6.4 Summary of findings and suggestions for future optimizations. 197

6.5 Description of the Accelerometer analytical model parameters. 198

6.6 Model parameters used to compare *Accelerometer*-estimated speedup with measured speedup on production systems. 206

6.7 Parameters used to model speedup and latency reduction for a few acceleration recommendations from Table 6.4. 211

ABSTRACT

Modern web services such as social media, online messaging, web search, video streaming, and online banking often support billions of users, requiring data centers that scale to hundreds of thousands of servers, i.e., *hyperscale*. In fact, the world continues to expect hyperscale computing to drive more futuristic, complex applications such as virtual reality, self-driving cars, conversational AI, and the Internet of Things. This dissertation presents technologies that will enable tomorrow's web services to meet the world's expectations.

The key challenge in enabling hyperscale web services arises from two important trends. First, over the past few years, there has been a radical shift in hyperscale computing due to an unprecedented growth in data, users, and web service software functionality. Second, modern hardware can no longer support this growth in hyperscale trends due to a steady decline in hardware performance scaling. To enable this new hyperscale era, hardware architects must become more aware of hyperscale software requirements and software researchers can no longer expect unlimited hardware performance scaling. In short, systems researchers can no longer follow the traditional approach of building each layer of the systems stack separately. Instead, they must rethink the synergy between the software and hardware worlds from the ground up. This dissertation establishes such a synergy to enable futuristic hyperscale web services.

This dissertation bridges the software and hardware worlds, demonstrating the importance of that bridge in realizing efficient hyperscale web services via solutions that span the systems stack. This dissertation's specific goal is to (1) design software that is aware of new hardware constraints and (2) architect hardware that efficiently supports new hyperscale software requirements. To this end, this dissertation spans two broad thrusts: (1) a software and (2) a hardware thrust to analyze the complex software and hardware hyperscale design space and use insights from these analyses to design efficient cross-stack solutions for hyperscale computation.

In the software thrust, this dissertation contributes *μSuite*, the first open-source benchmark suite of modern web services built with a new hyperscale software paradigm. *μSuite* facilitates future research and is being used in academia and industry to study hyperscale behaviors. Next, this dissertation uses *μSuite* to study software threading design implications in light of today's hardware reality and identifies new insights in the age-old research area of software threading. Driven by these insights, this dissertation demonstrates how software threading models must be redesigned at hyperscale by presenting an automated approach and tool, *μTune*, that makes intelligent threading decisions during system runtime.

In the hardware thrust, this dissertation architects both commodity and custom hardware to efficiently support hyperscale software requirements. First, this dissertation characterizes the shortcomings in *commodity hardware* running hyperscale web services, revealing insights that influenced commercial CPU designs. Based on these insights, this dissertation presents a design approach and tool, *SoftSKU*, that enables cheap commodity hardware to efficiently support new hyperscale software paradigms, improving the efficiency of real-world web services that serve billions of users, saving millions of dollars, and meaningfully reducing the global carbon footprint. This dissertation also presents a hardware-software co-design system, *μNotify*, that redesigns commodity hardware with minimal modifications by using existing hardware mechanisms more intelligently to overcome new hyperscale overheads.

Next, this dissertation presents a systematic characterization of how *custom hardware* must be designed at hyperscale, resulting in industry-academia joint benchmarking efforts, commercial hardware changes, and improved software development. Based on this characterization's insights, this dissertation presents *Accelerometer*, an analytical model that estimates realistic gains from hardware customization. Multiple hyperscale enterprises and hardware vendors have adopted *Accelerometer* to make well-informed hardware decisions.

CHAPTER I

Introduction

Bridging the software and hardware worlds via efficient solutions that span the systems stack enables hyperscale web services.

1.1 Motivation

1.1.1 Web Services Powered by Data Centers are Here, There, and Everywhere

The phrase "data center" is a presumption, a misnomer at best. It recalls an era when an enterprise's compute infrastructure was mostly devoted to data storage resources that were cobbled together in a basement or a coat closet [100]. For example, the first semblance to "large-scale" data processing came about in 1946 when the Electronic Numerical Integrator and Computer (ENIAC) was built for the U.S. Army to primarily store artillery firing codes and was dubbed as the first mainframe general-purpose digital computer [237][1]. At the time, data centers, like a sewer system or a highway's foundation beneath the potholes, were not something meant to be seen or paid attention to.

Over the course of time, the original assumptions about a data center started to evolve. In the 1960s, IBM developed commercial mainframe computers that required their own mainframe rooms in dedicated free-standing buildings [49]. The 1980s saw the launch of

[1] Fun fact: The institutions I attended for graduate school, the University of Pennsylvania and the University of Michigan, are two of the nine institutions that hold pieces of the ENIAC. So, perhaps it is no surprise that I decided to focus my dissertation research on making large-scale computation more efficient.

Personal Computers (PCs) that were commonly networked with remote servers, allowing a user on a PC to access files over a network [100]. By the time the internet became widely available in the 1990s, internet exchange buildings had sprung up in key international cities, leading to much larger facilities that housed hundreds or thousands of distributed servers. The “data center as a service” model became popular at this time, with these internet exchange buildings becoming the most important data centers of their time. Today, a data center is a multi-billion dollar warehouse-scale building that houses hundreds of thousands of servers to serve billions of users around the world [149].

This drastic evolution of a data center is fueled by the fact that the world has been undergoing a technological revolution in which web applications or services are growing in variety and complexity [102], dealing with exponentially-increasing data [108], and serving billions of users [31]. For example, modern web services such as social media, online messaging, web search, video/movie streaming, and online banking are becoming ubiquitous[2], performing increasingly new and sophisticated operations.

Modern web services require data centers that scale to hundreds of thousands of servers, i.e., *hyperscale* [445]. While at face value, hyperscale web services seem instantaneously available at the touch of a button, they, in fact, barely meet performance requirements despite running on prohibitively expensive data centers that consume enough power to light up entire countries [446]. As hyperscale computing grows to drive more futuristic applications such as virtual reality, self-driving cars, conversational AI, and the Internet of Things, existing hyperscale systems will face greater efficiency challenges due to these more complex tasks. More specifically, we have reached a new inflection point in web service complexity where, unless we improve the efficiency of web services and the data centers they run in, we cannot realize futuristic hyperscale web services. This dissertation enables the hyperscale web services of tomorrow by designing efficient system stacks for hyperscale computation.

[2] How often do you, dear reader, find yourself subconsciously reaching for your phone to skim through email or scroll through social media? The answer to this rhetorical question might implicitly motivate this subsection much better than I explicitly could.

1.1.2 Radical Shift in Hyperscale Computing

Over the past few years, there has been a radical shift in hyperscale computing due to an unprecedented growth in data [108], users [31], and web service functionality [102]. Data centers must now be able to handle this rapid growth in hyperscale trends.

Growth in data. Web services must increasingly process and store exponentially growing data [108]. In 2006, Clive Humby, a data scientist and mathematician, first coined the catchphrase "data is the new oil" [61]. At the time, the total global data recorded amounted to an estimated 160 exabytes [109]. Since then, data has exploded in volume, growing exponentially to approximately 33 zettabytes by 2018 [109]. As we move from an oil-driven era to a data-driven age that is shaped by the digital revolution (also known as the "Fourth Industrial Revolution"), data is increasingly becoming voluminous and varied in type, growing at a breakneck speed. In 2018, a white paper "Data Age 2025", predicted that data volumes would increase from 33 zettabytes in 2018 to 175 zettabytes by 2025 [28].

Growth in users. Web service usage has seen tremendous growth where, from 2000 to 2009, the number of users globally rose from 394 million to 1.9 billion [419]. By 2010, 22% of the world's population had access to computers with 1 billion Google searches every day, 2 billion daily YouTube views, and 300 million users reading blogs [59]. In 2014, web service users surpassed 3 billion, i.e., 44% of the world's population, with most users belonging to the world's richest countries [31]. As more countries enter the scene, hyperscale enterprises must cope with more and more users accessing their web services.

Growth in web service functionality. Initially, web services were primarily text-based, where a user was limited to reading information provided by content producers (e.g., news articles). There was no option for the user to communicate back since services were built statically. As web services evolved to facilitate more interaction between users, there was a sudden growth in the "social web", with the emergence of social media platforms such as Facebook and Twitter that found newer ways to enable and engage users [101]. The modern web service has grown to interpret user-generated data in more meaningful ways with the help of Artificial Intelligence (AI) and Machine Learning (ML) [102]. As user needs grow, web services must also become richer and more sophisticated, tailoring content to each user's specific needs. In the future, data centers must support futuristic applications such as self-driving cars, virtual reality, conversational AI, and the Internet of Things.

This rapid growth in web service functionality imposes an urgent need on hyperscale enterprises to enable futuristic web services while still making voluminous, multi-dimensional data/content instantaneously available to billions of users. In the next subsection, I explore whether hyperscale enterprises can rely on the hardware to rise up to meet these new, unprecedented web service expectations.

1.1.3 Decline in Technology Trends that Drive Processor Performance Scaling

The early data center server processors enjoyed the promise of two significant technology design trends that sustained fifty years of exponential computing advances. The first, Moore's Law, was the 1965 forecast by Intel co-founder Gordon Moore that transistor densities on integrated circuits would double about every two years [373]. The second, Dennard scaling, was the 1974 observation by Dennard et al. that transistor power densities would remain constant as transistors sizes are scaled down [203].

With Moore's Law and Dennard scaling working in cohesion, transistor sizes scaled down in each technology generation, and processor clock frequency increased at the same power consumption, resulting in faster circuits. As a result, hardware architects leveraged doubling transistor densities to create complex hardware features that further enhanced performance [252], while paying minimal attention to their designs' energy efficiency. These hardware enhancements enabled hyperscale enterprises to primarily focus on cranking up web service performance, to cater content instantaneously to the end user, improving end-user experience, maintaining service availability, and increasing revenue of operation.

Although performance and power scaling weathered several technology challenges throughout the history of Moore's Law and Dennard scaling, the last ten to fifteen years have posed particularly formidable challenges [480, 167, 469]. In particular, one challenge encountered around 2005, was caused by the breakdown of Dennard scaling, resulting in a computing power wall that made further compute improvements power-limited [167, 294, 437]. More recently, the decline of Moore's Law has resulted in computing performance not scaling as expected [480]. Hence, whereas processor performance scaled almost exponentially from the 1980s to around 2005, resulting in a 1000x performance increase, there has been only around a 5x performance improvement since 2005 [196]. These inflection points in the history of computer architecture have marked the end of almost half of a century of exponential growth in single-core processor performance [469].

Due to a decline in these hardware technology design trends, major hyperscale enterprises have reported that successive server generations running hyperscale web services exhibit diminishing performance returns [444]. Today, hyperscale data centers face a bleak situation where the hardware no longer rises to meet modern web service requirements, especially at a time when hyperscale web services have been facing an unprecedented growth in data, users, and functionality (detailed in Subsection 1.1.2). To enable futuristic hyperscale web services, there is now an urgent need to redesign the compute stack to efficiently deal with the rapid growth in web service data, users, and functionality, when the hardware does not scale as well as it used to.

1.1.4 Consequences on the Software and Hardware Research Landscape

To enable hyperscale computing in light of the unprecedented growth in web service trends and the decline in hardware performance scaling, there is a critical need to holistically design the systems stack to support these emerging trends. In other words, hardware architects must design the hardware layer to become more aware of hyperscale software needs and software researchers must design software layers to cope with the decline in hardware performance scaling. However, we find that systems researchers typically continue the traditional approach of building each layer of the systems stack separately. We now highlight how individual systems stack layers, i.e., the application layer, software abstraction layers, and the hardware layer, have evolved in response to the unprecedented growth in hyperscale web service trends and today's hardware reality.

1.1.4.1 Application layer: The shift towards a granular application architecture

Table 1.1: Timeline of the application layer's evolution in response to the unprecedented growth in hyperscale web service trends and today's hardware reality: There has been a shift from monolithic web application architectures to more granular architectures such as microservices.

1997	IBM releases Enterprise Java Bean to provide a "small" service that works with web-related software components [2]
1999	Microsoft introduces the Simple Object Access Protocol to utilize object methods using Hypertext Transfer Protocol (HTTP) [99] Web services start being built with Service-Oriented Architectures [107]
2005	Dr. Peter Rodgers used the term "Micro-Web-Services" during a presentation on cloud computing [2, 432]
2011	Several companies such as Netflix and Gilt adopt the microservice architecture [7, 94]
2014	Amazon introduces AWS Lambda, popularizing the serverless computing model [96]
2015	Kanev et al. use the term "microservice" for the first time in an ISCA paper (appears as a footnote on page 2) [285]
2018	Sriraman et al. introduce μSuite, the first benchmark suite of end-to-end web services composed of microservices [448]
2019	Gan et al. release the DeathStarBench microservice benchmark suite [230] Facebook reveals its adoption of the microservice architecture [443]
2020	Microsoft studies the wide-spread adoption effects of building web services using the serverless paradigm [429]

Table 1.1 references a selected timeline of events that influenced the application layer in response to the unprecedented growth in hyperscale trends and today's hardware reality.

1995 - 2005: The era of monolithic service architectures. In the early days, web services were built with monolithic architectures, where a service's different functionality components were developed as a single program that was run on a single hardware platform [477]. For example, when a video streaming service is built as a monolith, its various operations such as authorizing a user, fetching cached videos, recommending related videos, and displaying advertisements, are all performed by the same application binary running on a single server. However, with the unprecedented growth in data, users, and service functionality, building web services as monoliths is no longer a sustainable approach due to challenges in development, deployment, reliability, and scalability [60]. To overcome these challenges, web services are starting to be built in a more distributed and granular manner.

2005 - Present (2021): The era of granular web service architectures. The idea of a granular web service architecture is not entirely new. As early as 1997, IBM released the Enterprise Java Bean, one of the earliest efforts to provide a "small" service that interacts with web-related software components [2]. The limitation of working only with Java, brought about the solution known as Service-Oriented Architecture (SOA) which became the next evolutionary step for building web services in a granular manner [107]. SOA is an enterprise-wide approach to software development of application components that takes advantage of reusable software components and services. In SOA, each service has the code and data integrations required to execute a specific business function (e.g., authenticating a user).

As web services continued to grow, there emerged a need to develop services granules in a way that was not enterprise-wide, i.e., there was a need to build individual application granules that perform a specific service functionality in a way that is more agile, scalable, and resilient. An example of this need is reflected in an event that occurred in 2008, where a single missing semicolon brought down the entire Netflix website for several hours [74]. Hyperscale enterprises realized that having an entire system that is a single point of failure leads to stringent governance processes, long development cycles, and scalability issues. These challenges resulted in the evolution of "microservices"[3] that have become the mainstream web service architecture today.

Today, modern web services are composed of numerous independent, specialized, dis-

[3] In the "Profiling a warehouse-scale computer" paper, the term "microservice" appears for the first time in a systems/architecture venue as a footnote on page 2 [285].

tributed microservices [286, 207, 378] such as HTTP connection termination, key-value serving [223], query rewriting [144], click tracking, access-control management, and protocol routing [69]. Unlike the catastrophe in 2008, today, hundreds of microservices give Netflix the availability, scale, and speed needed to handle growing user numbers [7]. In fact, several other companies, such as Amazon [17], Gilt [94], LinkedIn [42], and SoundCloud [21], adopted microservice architectures to improve service development and scalability [477].

Microservices are reusable and interoperable as they are composed via standardized service interfaces such as Remote Procedure Calls (RPC) (e.g., Google's Stubby and gRPC [45] or Facebook/Apache's Thrift [37]). Of late, with the evolution of Amazon's AWS Lambda [96], several microservices are being built with the more granular serverless paradigm that allows allocating server resources on demand. This dissertation is one of the first works to comprehensively study hyperscale microservices' system-level implications [448, 449, 443, 446, 450, 444, 445], facilitating future research in this space [230, 367].

1.1.4.2 Software abstraction layers: The shift towards light-weight abstractions

Table 1.2 references a selected timeline of events that influenced software abstractions in response to the unprecedented growth in hyperscale trends and today's hardware reality.

1960 - 2007: The era of virtualization. In the 1960s, IBM came up with the concept of a "virtual machine" to develop an interactive system that could support multiple users and applications, thereby beginning the virtualization era [111]. Fast forward to the early 2000s and a different problem was brewing. Data centers were filled with expensive servers running at very low utilization levels because the software stack was unable to effectively utilize processor resources. Again, the solution was a form of virtualization that established a stranglehold in enterprise data centers. VMware, then a startup out of Stanford, enabled enterprises to dramatically increase server utilization by allowing multiple applications (including Operating Systems) to be packed into a single server [105]. The server utilization and cost savings from virtualization resulted in cloud computing as we know it today.

Table 1.2: Timeline of the evolution of software abstraction layers in response to the unprecedented growth in hyperscale web service trends and today's hardware reality: There has been a shift from heavy-weight abstractions (e.g., virtualization) to light-weight abstraction layers (e.g., containers).

1964	IBM introduces the concept of a "virtual machine" [111]
1971	The original idea behind "containers" is employed on Unix systems [1]
1980	Kernel bypass is invented for High Performance Computing [29]
1998	VMWare begins as a startup out of Stanford to create a virtualization layer that runs multiple applications on a single server, marking the start of an era where virtualization establishes a stranglehold in every enterprise data center [105]
2007	Linux becomes the primary OS used in many hyperscale data centers, eliminating the need for a separate virtualization layer that supports multiple Operating Systems on the same server [30]
2010	Intel releases the Data Plane Development Kit for kernel bypass [3]
2012	Containers become mainstream, isolating an application from the underlying hardware while still providing bare-metal performance [27]
2014	Kernel bypass becomes more common in data centers [154]
2018	Software Defined Data Centers begin to emerge [63]

In the late 2000s, a quiet technology revolution got under way at companies like Google and Facebook. Faced with the unprecedented challenge of serving billions of users in real time, these companies realized the need to build tailored systems stacks that aggregated (rather than carved) thousands of small, cheap servers and replaced larger, expensive monoliths. What these smaller, cheaper servers lacked in computing power they made up for in number, and sophisticated software (e.g., efficient request schedulers [475]) glued it all together to create a hyperscale computing infrastructure. The data center's shape changed. Linux became the Operating System (OS) of choice, making moot one of virtualization's core value propositions: the ability to simultaneously run different "guest" Operating Systems on the same physical server [30].

2007 - Present (2021): The era of light-weight abstractions. More recently, there is a shift towards developing "slimmer" abstraction layers in response to the growth in hyperscale trends and today's hardware reality. For example, virtual machines have been replaced by containers as a key abstraction primitive in many data centers [27]. Although containers are not new (the concept initially came about in the 1970s [1]), they are taking off now as they are a lighter-weight abstraction primitive. Unlike virtual machines that virtualize the hardware and contain an OS with the application stack, containers virtualize only the OS and contain only the application. As a result, containers have very small footprints and can be launched in mere seconds, enabling them to efficiently support granular microservices [27].

Another example of a “slimmer” abstraction and isolation software trend is the advent of OS kernel bypass mechanisms [3]. Over the past decade, I/O devices in data centers have sped up while CPU performance scaling has declined. To compensate, software researchers proposed eliminating the work the CPU performs upon receiving an I/O, i.e., eliminating the processing of OS handlers altogether [154]. Modern kernel-bypass techniques such as Software Data Planes have become mainstream today, with several hyperscale enterprises reimagining their software stacks around them [357]. More generally, hyperscale enterprises appear to be moving towards building Software Defined Data Centers (SDDC), where each data center resource component (e.g., network, storage, and CPU) is virtualized and delivered as a software service via suitable Applications Programming Interfaces (APIs) [63].

1.1.4.3 Hardware layer: The shift towards hardware specialization

Table 1.3 references a selected timeline of events that influenced the hardware layer in response to the unprecedented growth in hyperscale trends and today’s hardware reality.

2005 - Present (2021): The era of multicore architectures. With the end of Dennard scaling around 2005 [203], power consumption became a primary constraint for computer hardware development, resulting in new power-aware hardware design trends. Since the hardware industry continued to provide increasing transistor densities [373], architects were able to leverage this opportunity to turn from single-core processors to multi-core processor designs [208, 233, 395] that made use of the available extra transistors with a constrained power budget [138, 256]. Multicore architectures still dominate the design of today’s commodity server-class processors [446].

Table 1.3: Timeline of the hardware layer's evolution in response to the unprecedented growth in hyperscale web service trends and today's hardware reality: There has been a shift towards building specialized hardware for various "killer" web applications.

<2005	Exponential growth in single core processor performance that lasted for almost half of a century [373]
2005	Computer architects recognize the beginning of the new era of power-aware computer hardware design [294]
2006	Multi-core architectures become mainstream [208, 233, 395]
2016	Moore's Law has been steadily grinding to a halt [480]
2017	Google announces its Tensor Processing Unit [279]
2018	John Hennessy and David Patterson win the 2017 ACM A.M. Turing Award and speak about Domain Specific Architectures in the wake of the Moore's Law decline at ISCA 2018 [62] Microsoft announces its Neural Processing Unit for Deep Neural Network operations performed by web services [86]
2019	Intel delays its 10 nm process multiple times [52]
2020	IBM announces a compression accelerator for web applications [116]
2021	Google announces its in-house development of a System on Chip [43]

2015 - Present (2021): The era of hardware specialization. About a decade after the end of Dennard scaling, Moore's Law is steadily grinding to a halt [480][4]. As a result, architects have been specializing hardware to meet emerging web services' performance and power needs [413, 279, 181, 116]. An early example of hardware specialization is the Graphics Processing Unit (GPU) that was developed to execute highly-parallel applications. More recently, instead of solely relying on traditional hardware vendors, hyperscale enterprises are developing highly specialized hardware in-house [413, 279, 181, 116] to improve the efficiency of their important web services (e.g. Google's Tensor Processing Unit [279])[5].

[4] For example, Intel delayed its 10 nm process multiple times [52]

[5] Google recently announced the in-house development of a System on Chip that will integrate numerous specialized hardware components for individual web service functionalities [43]. The fact that Google, traditionally a software company, recruited Intel veteran Uri Frank as Google's Vice President of Engineering [43], appears to imply that the future holds the promise of highly-specialized enterprise-specific hardware designs.

1.2 Research Challenges and Goals

Current software and hardware systems were conceived at a time when we had scarce compute and memory resources, limited data and users, and easy hardware performance scaling due to Moore's Law. These assumptions are not true today. Today, the world is undergoing a technological revolution where emerging web services require data centers that scale to hundreds of thousands of servers, i.e., hyperscale, to efficiently process requests from billions of users. This technological revolution of hyperscale computing is emerging at a time when hardware is facing a steady decline in performance scaling [480].

To enable this new era of hyperscale computing, there is a clear need for systems researchers who design efficient computing systems that can both support today's key web services as well as enable the web services of tomorrow. However, to design efficient computing systems in light of modern hyperscale web service trends and today's hardware reality, systems researchers can no longer afford to build each layer of the systems stack separately (as shown in Tables 1.1, 1.2, and 1.3). Instead, *the first research goal is that systems researchers must rethink the synergy between the software and hardware worlds from the ground up.* Specifically, to improve hyperscale efficiency, computer architects must now be aware of web service software requirements, and software developers can no longer treat hardware as a black box that magically becomes faster every year.

The main challenge in rethinking the synergy between the software and hardware worlds is a large and complex software and hardware design space that makes it intractable to manually identify optimal designs. As one example, work related to this dissertation discovered that the software threading design space has complex implications induced by the decline of hardware performance scaling, making it impractical for a software developer to manually identify the best threading design [449]. As a second example, the hardware customization design space has complex implications on web service software depending on whether the hardware customization is an on-chip CPU optimization or an off-chip or remote hardware accelerator [444].

Manually navigating this vast and complex design space to make efficient design decisions is often intractable at hyperscale since (1) design implications vary across secondary conditions such as web service load variations, (2) trial-and-error methods or experience-based intuition cannot systematically capture design space implications, (3) web service code evolves quickly, (4) synthetic experiments do not necessarily capture complex production behavior, and (5) the effects of tuning a single design configuration are often too small to be manually captured with sufficient statistical significance. Hence, even though systems researchers have been working to improve web service efficiency for the past twenty years, they cannot enable futuristic web services unless they achieve *the second research goal of automatically navigating, i.e., self-navigating, the complex software and hardware hyperscale design space*.

Given the widespread need for web services, to achieve both these research goals, it is of paramount importance to devise mechanisms that can automatically enhance the synergy between the complex software and hardware worlds. In other words, rather than following the traditional approach of building each layer of the systems stack separately, modern hyperscale web service trends and today's hardware reality have created a need to automatically (1) bring new hardware insights when designing software stack layers and (2) draw on fundamental software design principles to systematically architect the hardware layer.

My work pursues the vision of bridging the software and hardware worlds, demonstrating the importance of that bridge in enabling the hyperscale web services of tomorrow via efficient self-navigating solutions that span the systems stack. Specifically, this dissertation's vision is to (1) redesign web service software based on new overheads induced by the decline of hardware performance scaling and (2) rearchitect data center commodity and custom hardware to support new software requirements that are a consequence of the unprecedented growth in hyperscale web service trends.

To achieve this research vision in a way that self-navigates the complex software and hardware design space, this dissertation spans two broad thrusts: (1) a software and (2) a hardware thrust to study both the complex software and hardware design space. In the software thrust, I ask the question: how do we design hyperscale web service software based on the overheads induced by today's hardware reality? In the hardware thrust, I ask the question: how do we architect data center commodity and custom hardware to support the unprecedented growth in hyperscale software trends? In light of emerging hyperscale trends and today's hardware reality, it is critical to systematically answer both questions to enable the hyperscale web services of tomorrow.

To answer both questions, I employ a three-fold research approach. First, I systematically lay out a taxonomy of various design axes in a particular software or hardware design space (e.g., microservices' software threading designs), analyzing their efficiency implications in a structured and comprehensive manner. Second, I use insights from my characterization to design practical, scalable solutions that self-navigate a complex software or hardware design space to improve hyperscale efficiency. I also mitigate key overheads identified in my analyses. Third, I build these systems and when possible, deploy them in real hyperscale systems.

Overall, the work related to this dissertation addresses the gap between hyperscale efficiency and growth expectations and today's hardware reality. The remainder of this section provides an overview of the specific challenges addressed by this dissertation, highlighting this dissertation's goals and contributions along the way.

1.2.1 Enabling the Study of Modern Web Services

Challenge: Lack of open-source benchmarks to study modern web services

Modern web services are increasingly built using microservice architectures, wherein a complex web service is composed of numerous distributed microservices such as HTTP connection termination, key-value serving [223], query rewriting [144], click tracking, access-control management, and protocol routing [69]. Whereas monoliths face greater than 100 ms Service Level Objectives (SLOs) (e.g., $\sim$300 ms for web search [471]), microservices must often achieve sub-ms SLOs (e.g., $\sim$100 μs for protocol routing [501]) as many microservices must be invoked serially to serve a user's query. Hence, sub-ms–scale OS/network overheads (e.g., a context switch cost of 5-20 μs [470]) are often insignificant for monoliths. However, the microservice regime differs fundamentally: OS/network overheads (e.g., context switches, network protocol delays, inefficient thread wakeups, and lock contention) that are often minor with monolithic request service times of 100s of milliseconds, can dominate microservice latency distributions. For example, even a single 20 μs spurious context switch implies a 20% latency penalty for a request to a 100 μs-response latency protocol routing microservice [501]. Hence, it is critical to revisit prior conclusions on sub-ms–scale OS/network overheads for this new microservice regime [145].

At the time I started my dissertation work, there existed no representative, open-source benchmarks to study the microservice regime. Widely-used academic data center benchmark suites, such as CloudSuite [221] or Google PerfKit [78], were unsuitable for characterizing sub-ms–scale overheads in microservices as they use monolithic rather than microservice architectures and largely have request service times that are greater than 100 ms. Hence, there was a real need for open-source benchmarks that enable the study of microservices.

Goal: Open-source a benchmark suite of representative modern web services

To study microservices, as a part of this dissertation's software contributions, I introduce the first open-source benchmark suite[6] of end-to-end modern web services composed of microservices, called *μSuite* [448]. *μSuite* includes four end-to-end web services that incorporate open-source software: a content-based high dimensional search for image similarity—`HDSearch`, a replication-based protocol router for scaling fault-tolerant key-value stores—`Router`, a service for performing set algebra on posting lists for document retrieval—`Set Algebra`, and a user-based item recommender system for predicting user

[6] Available at https://github.com/wenischlab/MicroSuite

ratings—Recommend. Since its publication [448], *μSuite* has been used by researchers in academia and industry (e.g., MIT, UIUC, UT Austin, Georgia Tech, Cornell, ARM, and Intel).

In this dissertation, I use *μSuite* to study the OS/network performance overheads incurred by microservices. My main finding is that the threading interactions with the OS and network layers introduce microsecond-scale overheads that significantly affect microservices, but are insignificant to their monolithic counterparts. I also observe that inefficient OS scheduler decisions can degrade microservice latency by up to 87%. Hence, intelligent thread scheduling and better threading models can greatly improve microservice performance.

1.2.2 Redesigning Software Based on Underlying Data Center Hardware Constraints

Challenge: Software threading models are unaware of overheads caused by hardware constraints

My study of OS/network performance overheads using *μSuite* showed that microservices can benefit from better threading designs. These threading-induced overheads that microservices face are due to today's hardware reality, where network devices have sped up while CPU performance scaling has nearly stopped [196]. Today, a CPU thread's accesses to the underlying OS/network stacks cause threading-induced overheads that arise from sources such as thread contention on locks, thread wakeup delays, and context switching of threads. Hence, analyzing various software threading designs' implications and rethinking software threading models for modern microservices has become a deeply important problem.

Goal: Rethink threading models to overcome overheads faced by modern web services

To study threading-induced software overheads that arise due to hardware constraints, there is a need to systematically analyze the sub-ms–scale OS and network overheads that arise from threading and concurrency design decisions. As a part of this dissertation's software contributions, I use *μSuite* to systematically introduce and comprehensively characterize a *taxonomy of threading models* [449]. My taxonomy is composed of software threading dimensions that are commonly used to build a microservice, such as synchronous or asynchronous RPCs, in-line or dispatched RPC handlers, and interrupt- or poll-based network reception. I also vary thread pool sizes dedicated to the various functionalities, i.e., network polling, RPC handling, and response execution. These threading design axes yield a rich space of microservice software threading architectures that interact with the underlying OS and hardware in starkly varied ways. Hence, my threading taxonomy and analysis enables expert and novice developers alike to guide their microservice threading designs.

This dissertation makes the important observation that no single threading model is best across all hyperscale load conditions, paving the way for an automatic load adaptation system that tunes threading models to improve microservice efficiency. Specifically, my threading model characterization demonstrates that the relationship between optimal threading model and service load is complex—one could not expect a developer to pick the best threading model a priori. For example, at low load, models that poll for network traffic perform best, as they avoid thread wakeup delays. Conversely, at high load, models that separate network polling from RPC execution enable higher service capacity and blocking outperforms polling for incoming network traffic as it avoids wasting CPU on fruitless poll loops. Hence, exploiting these inherent threading model trade-offs during system runtime can significantly improve microservice latency.

To exploit threading trade-offs at runtime, I present a system, *μTune* [449][7], that features a framework that builds upon open-source RPC platforms [45] to abstract threading model design from service code. *μTune*'s second feature is an intelligent run-time system that determines load via event-based monitoring and automatically adapts to time-varying service load by self-navigating the threading design space, i.e., tuning threading models and scaling thread pool sizes. Both features enable *μTune* to dynamically curtail microservice latency by 1.9× over static peak load-sustaining threading models (that an expert developer might

[7] Available at https://github.com/wenischlab/MicroTune

have picked) and state-of-the-art adaptation techniques [242, 316, 117][8].

In follow-on work, my co-authors and I mitigate OS/network-induced microsecond-scale stalls identified in my threading characterization [449]. We present *Duplexity*, a heterogeneous server architecture that schedules latency-insensitive jobs when a microservice faces microsecond-scale stalls, improving data center performance and energy efficiency.

1.2.3 Architecting Commodity Hardware for New Web Service Software Paradigms

At global user population scale, important web services that are composed of numerous microservices can grow to account for an enormous installed base of physical hardware. For example, across Facebook's global server fleet, seven key microservices in four service domains run at hyperscale, occupying a large portion of the compute-optimized installed base [446]. In light of this new microservice software paradigm, it is important to answer the question: do commodity server platforms serve microservices well? Are there common bottlenecks across microservices that we might address when designing future server architectures?

Challenge: Commodity hardware does not efficiently support modern web service software paradigms

To identify whether commodity hardware efficiently supports microservices, I undertake comprehensive system-level and architectural characterizations of important microservices on Facebook production systems serving live traffic. I find that web service functionality disaggregation across microservices has resulted in enormous diversity in system and CPU architectural requirements, with new CPU bottlenecks (e.g., high I/O processing latency and high instruction cache misses). The bottlenecks identified in this work made hardware vendors reconsider the benchmarks they used for decades to evaluate new servers.

As examples, I find that caching microservices [171] require intensive I/O and microsecond-

[8] My conversations with researchers at several hyperscale enterprises revealed that μ*Tune* could find an immediate application in their data centers; I was invited to intern at several of these companies to integrate μ*Tune* into their hyperscale web services.

scale response latency and frequent OS context switches comprise 18% of CPU time. In contrast, a `Feed` [506] microservice computes for seconds per request with minimal OS interaction. Facebook's `Web` [388] microservice exhibits massive instruction footprints, leading to astonishing instruction cache misses and branch mispredictions, while other microservices exhibit much smaller instruction footprints. Some microservices depend heavily on floating-point performance while others have no floating-point instructions. This great diversity in hardware bottlenecks across microservices makes it challenging for a one-size-fits-all commodity processor to efficiently support diverse microservices.

Goal 1: Extract greater performance from existing commodity data center hardware

The diversity in hardware bottlenecks across microservices might suggest a strategy to specialize CPU architectures to suit each microservice's distinct needs. Indeed, this dissertation has identified new hardware bottlenecks that have since influenced the design of commercial server-class processors [446, 445]. However, hyperscale enterprises have strong economic incentives to limit hardware platforms' diversity to (1) maintain fungibility of hardware resources, (2) preserve procurement advantages that arise from economies of scale, and (3) limit the overhead of qualifying/testing myriad hardware platforms. As such, there is an immediate need for strategies that extract greater performance from existing commodity server architectures to efficiently support diverse microservices on commodity hardware.

As a part of this dissertation's hardware contributions, I introduce an automated approach and tool to improve hyperscale microservice performance on cheap commodity server architectures (often called "SKUs," short for "Stock Keeping Units"). This approach called *SoftSKU*, which is presented in Chapter IV, is a design-time strategy that tunes coarse-grain (e.g., boot time) OS and hardware configuration knobs available on commodity processors to help a processor platform or SKU better support its assigned microservice. OS and CPUs provide several specialization knobs; I focus on seven: (1) core frequency, (2) uncore frequency, (3) active core count, (4) code vs. data prioritization in the last-level cache ways, (5) hardware prefetcher configuration, (6) use of transparent huge pages, and (7) use of statically-allocated huge pages. I also propose new CPU knobs (e.g., Branch Target Buffer ways) that can be made configurable to create finer-grained soft SKUs.

Manually identifying a microservice-specific *SoftSKU* is impractical since the design space is large, code evolves quickly, synthetic load tests do not often capture production behavior, and the effects of tuning a single knob are often small. Hence, I build an automated design tool—μSKU—that self-navigates the hardware configuration design space to optimize a hardware SKU for each microservice. μSKU automatically varies configurable server knobs, by searching within a predefined design space via A/B testing, where it compares the performance of two identical servers that differ only in their knob configuration. μSKU collects copious fine-grain performance measurements while conducting automated A/B tests on production systems serving live traffic to search for statistically significant performance gains. I evaluate μSKU on hyperscale production microservices and demonstrate that the soft SKUs designed by μSKU outperform stock and production server configurations by up to 7.2% and 4.5% respectively, with no additional hardware requirement.

SoftSKU demonstrates that before resorting to hardware customization, there is still significant performance that can be extracted from commodity processors by tuning their OS and hardware knobs. In this manner, soft SKUs significantly improve the performance efficiency of real-world Facebook production microservices that serve billions of users. Moreover, by better utilizing cheap commodity hardware, soft SKUs save millions of dollars and also meaningfully reduce the global carbon footprint [5]. Since the publication of this work [446], several hyperscale enterprises have dedicated teams of engineers to explore additional configurable hardware/OS soft-SKU knobs (e.g., SIMD width).

Goal 2: Redesign commodity hardware to overcome new web service overheads

In my characterization of system-level and architectural bottlenecks faced by Facebook microservices, I observe that several microservices frequently make I/O requests and await I/O responses. This behavior is because a microservice typically communicates with numerous I/O devices and queues. For example, a microservice may receive network requests from tens to hundreds of other microservices via the Network Interface Controller alone. Since several microservices expect microsecond-scale service times, the microsecond-scale I/O notification latency [449], which used to be insignificant for monolithic services, can now dominate the microservice regime. Moreover, the I/O notification overhead also quickly adds up across microservice chains to dominate the end-to-end web service latency [448].

Since I/O notification overheads dominate in microservices, it is critical to understand why existing I/O notification paradigms (which were primarily built for monoliths), fall short for microservices. This understanding paves the way for redesigning I/O event notification paradigms for the microservice regime. To this end, I comprehensively characterize state-of-the-art I/O notification paradigms that real-world microservices use [18, 73, 89, 97] (e.g., OS interrupts, spin-polling, and MWAIT variants) to analyze how well they meet microservice requirements. The main takeaway from my characterization is that existing notification paradigms do not scale well and execute expensive I/O stacks. Hence, there is a critical need to redesign I/O notification for the microservice regime.

As a part of this dissertation's hardware contributions, I present μNotify, the first I/O notification paradigm to achieve scalable, near-constant time notification. μNotify reimagines a commodity CPU core's software and hardware design dedicated for monitoring, prioritizing, and receiving I/O. Such a design must (1) bypass expensive I/O stacks, (2) scale across tens to hundreds of I/O queues, and (3) prioritize I/O work items. To achieve these design goals, μNotify's key insight is to make better use of cache coherence signals generated by existing commodity processors. Specifically, μNotify observes writes to I/O queues by tracking hardware-generated cache line invalidation coherence signals generated by an I/O device writing to a I/O queue. Recording hardware-generated coherence invalidation signals takes near-constant time, serving as low-overhead notification. Since the invalidation is hardware-generated, μNotify bypasses the OS and scales across numerous I/O queues, achieving 15.63x better throughput and 14.2x better latency than the state-of-the-art I/O notification mechanisms.

In follow-on work [298, 299, 297], my co-authors and I mitigate the architectural bottlenecks in the frontend of the processor pipeline (e.g., instruction cache misses) that I found to be significant in my characterization of Facebook's microservices [446]. We use profile-guided optimization techniques to inform frontend operations (e.g., I-cache and BTB prefetching and replacement decisions) to achieve near-ideal frontend performance.

1.2.4 Architecting Custom Hardware for New Web Service Software Paradigms

Challenge: Lack of a systematic understanding of hardware acceleration opportunities at hyperscale

My prior work [446] revealed that modern microservices are so diverse that they could benefit from running on custom hardware. In fact, to improve hardware efficiency, several architects today work on developing numerous specialized hardware accelerators for important microservice domains (e.g. Machine Learning tasks). Designing such custom hardware accelerators for each microservice operation might improve performance or energy. However, designing custom hardware for each microservice operation is prohibitively expensive at hyperscale since data center operators lose procurement advantages that arise from economies of scale and must also develop and test on myriad custom hardware platforms. Hence, an important question arises: *which microservice software operations consume the most CPU cycles and are worth accelerating in the hardware?*

To build specialized accelerators for these key microservice operations, it is important to first systematically identify which type of accelerator meets microservice requirements and is worth designing and deploying. Deploying specialized hardware is risky at hyperscale, as the hardware might under-perform due to performance bounds from the microservice's software interaction with the hardware, resulting in high monetary losses. For example, when hyperscale data center operators tried to adopt a few new accelerators, they observed that these accelerators reduced performance due to overlooked microservice software interaction overheads, inducing high monetary losses [445]. To make well-informed hardware decisions, it is crucial to systematically answer the following question early in the design phase of a new accelerator to determine whether the new accelerator is worth designing: *how much can the accelerator realistically improve its targeted microservice overhead?*

Goal: Analytically model hardware acceleration opportunities at hyperscale

To answer the first question posed above (i.e., which microservice software operations consume the most CPU cycles and are worth accelerating in the hardware?), I undertake a comprehensive characterization of how microservices spend their CPU cycles (as a part of this dissertation's hardware contributions). I study seven important hyperscale Facebook microservices in four diverse service domains that run across hundreds of thousands of servers, occupying a large portion of Facebook's global server fleet. My characterization reveals that microservices spend only a small fraction of CPU cycles executing their main application functionality (e.g., performing a Machine Learning operation); the remaining cycles are spent in common *orchestration overheads*, i.e., operations that are not critical to the main microservice functionality (e.g., I/O notification, logging, and compression). Accelerating such common building blocks can greatly improve data center performance. Already, a few hardware vendors have used this study's insights to influence hardware customization for orchestration overheads [445].

My characterization drove a hardware vendor to consider more representative benchmarks (in place of traditional ones they used for decades) when evaluating hardware designs [445]. This characterization work has resulted in an industry-academia joint collaborative effort to design and open-source data center benchmarks that represent the hyperscale behaviors identified in my characterization. Additionally, my characterization tool has been integrated into Facebook's fleet-wide performance monitoring infrastructure; it currently assimilates statistics from hundreds of thousands of servers from around the world to help developers visualize the performance impact of their code changes at hyperscale [445].

To answer the second question posed above (i.e., how much can the accelerator realistically improve its targeted microservice overhead?), I develop *Accelerometer*[9], an analytical model for hardware acceleration [444]. *Accelerometer* estimates realistic gains from hardware acceleration by self-navigating the various performance bounds that arise from a microservice's software interactions with the hardware. *Accelerometer* identifies performance bounds and design bottlenecks early in the hardware design cycle, and provides insight into which hardware acceleration strategies may alleviate these bottlenecks.

Accelerometer models both synchronous and asynchronous microservice software interactions for three hardware acceleration strategies—on-chip, off-chip, and remote. It assumes an abstract system with three components (1) *host*: a general-purpose CPU, (2) *accelerator*: custom hardware to accelerate a kernel, and (3) *interface*: the communication layer between the host and the accelerator (e.g., a PCIe link). *Accelerometer* models the microservice throughput speedup and the per-request latency reduction for the three acceleration strategies. Modeling both speedup and latency reduction ensures that acceleration enables a higher throughput without violating latency Service Level Objectives.

I validate *Accelerometer*'s utility via three retrospective case studies conducted on production systems, by comparing model-estimated speedup with real microservice speedup. In all three studies, *Accelerometer* estimates the real microservice speedup with an error that is less than or equal to 3.7%. I also use *Accelerometer* to project speedup for the acceleration recommendations derived from key common overheads identified by my characterization of Facebook's microservices.

As microservices evolve, Accelerometer's generality makes it even more suitable in determining new hardware requirements early in the design phase. Since I validated *Accelerometer* in production and made it open-source [4][10], I am happy to report that it has been adopted by multiple hyperscale enterprises (e.g., with developing their encryption and compression accelerators) to make well-informed hardware decisions [445].

[9] *Accelerometer* has been recognized for its long-term impact potential with an IEEE Micro Top Picks distinction (one of 12 total computer architecture papers to receive this recognition in 2020) [445].

[10] Available at https://doi.org/10.5281/zenodo.3612796

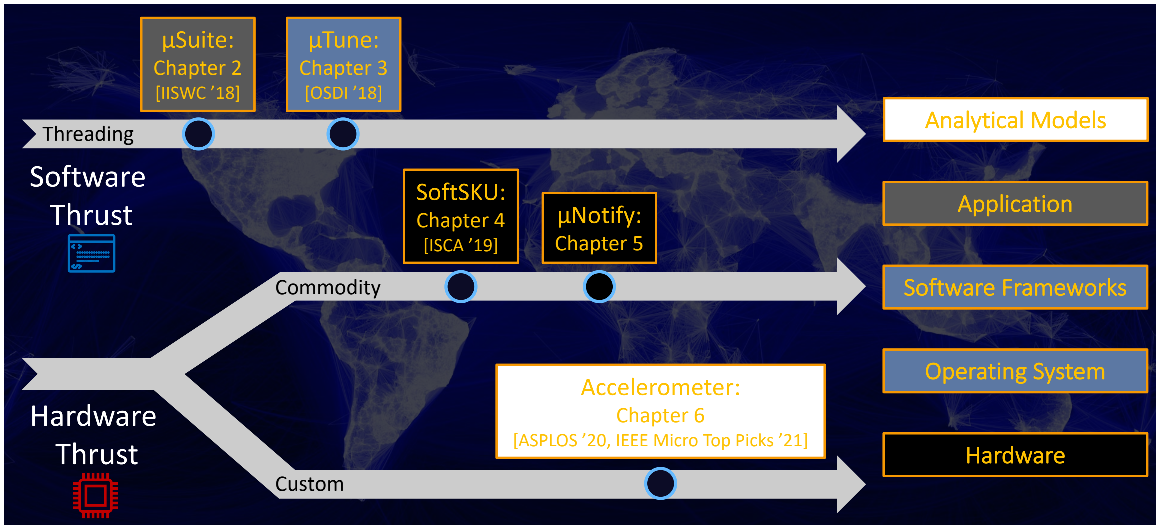


Figure 1.1: A timeline of the work presented in this dissertation, organized horizontally according to the years when portions of the individual dissertation chapters were published. The timeline is divided into this dissertation's software and hardware thrusts to show a bird's eye view of the software and hardware design space studied in this dissertation. Each work of research is annotated with the dissertation chapter in which it is covered and the publication venue where portions of that chapter were published. The background color of the box representing each work of research is color coordinated with the levels of the systems stack (shown on the right) that the technologies presented in that chapter cover.

1.3 Dissertation Contributions

My dissertation is motivated by the scale of the problems I am solving and the opportunity for real-world technical and societal impact. This dissertation makes a number of novel and impactful software and hardware contributions that bridge the software and hardware worlds to enable the hyperscale web services of tomorrow. Rather than following the traditional approach of building each layer of the systems stack separately, this dissertation uniquely brings new hardware insights when designing software stack layers and draws on fundamental software design principles to systematically architect the hardware layer.

Fig. 1.1 shows a graphical depiction of a timeline of the research work presented in this dissertation. The timeline categorizes this dissertation's software and hardware thrusts, depicting an overview of the software and hardware design spaces studied in this dissertation. Fig. 1.1 also annotates each research work to show the levels of the systems stack that the technologies presented in that work cover.

As shown in Fig. 1.1, in the software thrust, this dissertation's software contributions in terms of a representative, open-source benchmark suite of modern web services built of microservices facilitate future research on the modern microservice paradigm. Using this benchmark suite, this dissertation builds on decades of software threading model research, identifying gaps in existing software threading designs that arise due to the decline in hardware performance scaling. This dissertation presents new software threading model insights that enables fundamentally redesigning software threading models for the emerging hyperscale microservice paradigm.

In the hardware thrust, this dissertation's hardware contributions systematically architect data center hardware in a way that is aware of fundamental software design principles to support the unprecedented growth in hyperscale software trends. By comprehensively characterizing the commodity and custom hardware design space in light of emerging hyperscale software trends, this dissertation facilitates a holistic approach to future hardware design. This characterization has influenced the design of commercial hardware architectures, enabled industry-academia joint benchmarking efforts, and improved the software development process. My characterization's insights enable techniques that help maintain the performance improvement rate for commodity processors, triggering a significant shift in the hardware industry, saving millions of dollars, and meaningfully reducing the global carbon footprint. Furthermore, driven by this characterization, this dissertation presents a rigorous, analytical alternative to ad hoc hardware customization approaches that enables hyperscale enterprises to make well-informed hardware decisions.

Overall, through the software and hardware contributions summarized below, this dissertation realizes efficient web services from analytical models on paper to system deployment at hyperscale.

- **Demonstration of the benefits of cross-stack design to enable hyperscale web services.** This dissertation's primary, unique contribution is bridging the software and hardware worlds and demonstrating the importance of that bridge in realizing efficient hyperscale web services via solutions that span the systems stack. Specifically, this dissertation spans two broad thrusts: (1) a software and (2) a hardware thrust to answer two important questions. First, how do we redesign hyperscale web service software based on the overheads induced by today's hardware reality? Second, how do we rearchitect data center hardware to support the unprecedented growth in hyperscale software trends? By systematically answering both questions, this dissertation uniquely demonstrates the importance of cross-stack design in enabling the hyperscale web services of tomorrow.

- **Presentation of the first open-source benchmark suite of end-to-end modern web services that facilitates future academic and industry research [448].** This dissertation is the first to present an open-source benchmark suite of web services built with the microservice paradigm—*μSuite*. By demonstrating how this benchmark suite can be used to study new overheads that manifest in the microservice regime, this dissertation facilitates future research, with this benchmark suite being used by researchers in academia and industry to analyze microservices.

- **Identification of new insights in the age-old research area of software threading models that led to redesigning threading models for hyperscale web services [449].** This dissertation revisits the study of threading models in the context of today's hyperscale web services by systematically laying out a taxonomy of threading models and analyzing them to make important observations about new threading implications that manifest at hyperscale. Driven by this systematic threading analysis, this dissertation demonstrates how threading models must be redesigned for modern web services by presenting an automated approach and associated tool, *μTune*, that makes intelligent threading decisions during system runtime.

- **Characterization of shortcomings in commodity hardware running hyperscale**

web services that influenced the design of commercial CPU architectures [446]. This dissertation comprehensively characterizes system-level and architectural bottlenecks faced by real-world, production web services running on commodity hardware deployed at hyperscale. This characterization has influenced the design of commercial server-class commodity processors.

- **Demonstration of how existing commodity hardware can be used more efficiently to enable hyperscale web services that resulted in real-world data centers prioritizing this approach's adoption over the modern-day trend of customizing hardware [446].** In today's trend of expensive hardware customization, this dissertation presents an alternative approach and automated tool, *SoftSKU*, that demonstrates how cheap commodity data center hardware can still efficiently support new web service software paradigms in this post-Moore era. This dissertation demonstrates how the *SoftSKU* approach significantly improves the performance efficiency of real-world, production web services that serve billions of users, saving millions of dollars and meaningfully reducing the global carbon footprint [5].

- **Demonstration of how existing hardware mechanisms can intelligently be used to overcome new web service software overheads [450].** This dissertation demonstrates how commodity hardware can be redesigned with minimal modifications for modern web services by intelligently extracting greater benefits from existing hardware mechanisms. Specifically, by presenting *μNotify*, this dissertation demonstrates how existing cache coherence mechanisms can be intelligently used better to mitigate new I/O notification overheads that are faced by modern hyperscale web services.

- **Presentation of a systematic understanding of hardware customization opportunities at hyperscale that enabled industry-academia joint benchmarking efforts, influenced commercial hardware design, and improved software development [444, 445].** In today's era of developing custom hardware for "killer" applications, this dissertation takes a step back and systematically answers the Amdahl's Law question of which hyperscale web service software operations are worth accelerating in the hardware. This study's insights have received recognition in academia and industry with (1) hardware vendors developing hardware customizations based on this study's insights, (2) an industry-academia joint collaborative effort to develop benchmarks that represent the hyperscale behaviors identified in this study (and replace traditional benchmarks used for decades), and (3) the deployment of the characterization approach and tool in a hyperscale company's fleet-wide performance monitoring production infrastructure to improve the software development process.

- **Presentation of a rigorous, analytical alternative to ad hoc hardware customization approaches that enabled real-world hyperscale data centers to make well-informed hardware investments [444, 445].** This dissertation presents an analytical model, *Accelerometer*, to estimate realistic gains from hardware acceleration early in the hardware design cycle. *Accelerometer*'s generality has resulted in multiple hyperscale companies and hardware vendors adopting *Accelerometer* to make well-informed hardware decisions.

1.4 Dissertation Outline

This dissertation is designed to be educational, both to convey the results of the research that I undertook as well as some of the lessons I learned about doing research. The latter are captured in footnotes so as to not detract from the technical content. The rest of this dissertation is organized as follows.

First, Chapters II and III detail this dissertation's software contributions. Chapter II introduces the first-ever benchmark suite of end-to-end modern web services composed of microservices, *µSuite*. After introducing *µSuite*, the second half of Chapter II uses *µSuite* to present a characterization of new OS and network overheads incurred by modern web services. Chapter III uses the insights identified in Chapter II's characterization to systematically lay out a taxonomy of software threading models, analyzing the trade-offs between the different threading models in light of the overheads induced by today's underlying hardware constraints. This chapter makes the important observation that no single software threading model is best across all loads, paving the way for the second half of Chapter III which presents *μTune*, an automatic load adaptation system that tunes threading models and thread pool sizes to improve microservice performance efficiency.

Next, Chapters IV, V, and VI detail this dissertation's hardware contributions. Chapter IV presents a comprehensive characterization of system-level and architectural bottlenecks faced by Facebook's production microservices. Using this characterization's insights, this chapter introduces the first approach and associated automated tool, *SoftSKU*, to improve hyperscale microservice performance on existing, cheap commodity hardware. The second half of Chapter IV details an evaluation of how *SoftSKU* improves the performance efficiency of Facebook production microservices that serve billions of users, saving millions of dollars and meaningfully reducing the global carbon footprint. Chapter V makes better use of commodity CPU architecture mechanisms to mitigate new I/O notification overheads identified in Chapter IV's analysis. After analyzing the limits of existing I/O notification paradigms, this chapter introduces and evaluates the first I/O notification paradigm that achieves scalable, near-constant time I/O notification, μNotify. While Chapters IV and V focus on making better use of existing commodity hardware, Chapter VI focuses on developing and deploying new custom hardware in a well-informed manner at hyperscale. Chapter VI presents a systematic characterization of hardware acceleration opportunities at hyperscale. Based on this characterization, the second half of Chapter VI introduces *Accelerometer*, an analytical model that estimates realistic gains from hardware acceleration early in the hardware design phase, to make well-informed hardware customization decisions at hyperscale.

Finally, Chapter VII presents ongoing and future research directions and subsequently concludes this dissertation.

CHAPTER II

A Benchmark Suite for Microservices

As discussed in Chapter I, web services such as web search, advertising, and online retail form a major fraction of data center applications [362]. Meeting soft real-time deadlines in the form of SLOs determines end-user experience [135, 169, 307, 64] and is of paramount importance. Whereas web services once had largely monolithic software architectures [146], modern web services are composed of numerous, distributed microservices [448, 286, 207, 378]. These microservices are composed via standardized RPC interfaces, such as Google's Stubby and gRPC [45] or Facebook/Apache's Thrift [37].

While monolithic applications face tail (99^{th}+%) latency SLOs of the order of hundreds of milliseconds (e.g., $\sim$300 ms for web search [471, 447]), microservices must often achieve single-digit millisecond tail latencies implying sub-ms medians (e.g., $\sim$100 μs for protocol routing [501]) as many microservices must be invoked serially to serve a user's query. For example, a Facebook news feed service [248] query may flow through a serial pipeline of many microservices invoked via RPCs, such as 1) Sigma [38]: a spam filter; 2) McRouter [385]: a protocol router; 3) Feed [446]: a news feed stories extractor; 4) Tao [171]: a distributed social graph data store; and 5) MyRocks [75]: a user database. Serial microservice interactions place tight single-digit millisecond latency constraints on individual microservices. We expect continued growth in web service data sets and applications with composition of ever more microservices with increasingly complex interactions. Hence, the pressure for better microservice latency continually mounts.

Prior academic studies focused on monolithic services [221], which typically have tail latency SLOs of the order of hundreds of milliseconds. [359]. Hence, sub-ms-scale OS/network overheads (e.g., a context switch cost of 5-20 μs [470]) are often insignificant for monolithic services. However, the sub-ms-scale regime differs fundamentally: OS/network overheads that are often minor with latency SLOs of the order of hundreds of milliseconds, such as spurious context switches, network and RPC protocol delays, inefficient thread wakeups, or lock contention, can come to dominate microservice latency distributions. For example, even a single 20 μs spurious context switch implies a 20% latency penalty for a request to a 100 μs-response latency protocol routing microservice [501]. Hence, prior conclusions must be revisited for the microservice regime [145].

Modern web services are composed of a complex web of microservices that interact via RPCs [207] (Fig. 2.1). Many prior works have studied leaf servers [471, 340, 341, 198, 472, 409], as they are typically most numerous, making them cost-critical. However, we[1] find that *mid-tier* servers, which must manage both incoming and outgoing RPCs to many clients and leaves, perhaps face even greater tail latency optimization challenges, but have not been similarly scrutinized. The mid-tier microserver is a particularly interesting object of study since (1) it acts as both an RPC client and an RPC server, (2) it must manage fan-out of a single incoming query to many leaf microservers, and (3) its computation typically takes tens of microseconds, about as long as OS, networking, and RPC overheads.

While it may be possible to study mid-tier microservice overheads in a purely synthetic context, greater insight can be drawn in the context of complete end-to-end web services. Widely-used academic data center benchmark suites, such as CloudSuite [221] or Google PerfKit [78], are unsuitable for characterizing microservices as they (1) include primarily leaf services, (2) use monolithic rather than microservice architectures, and (3) largely have request service times of the order of hundreds of milliseconds.

[1] Some of the work in this chapter was performed in collaboration with my Ph.D. advisor, Thomas. F. Wenisch [448]. Therefore, I use the "we" pronoun in this chapter to acknowledge Wenisch's involvement in this work.

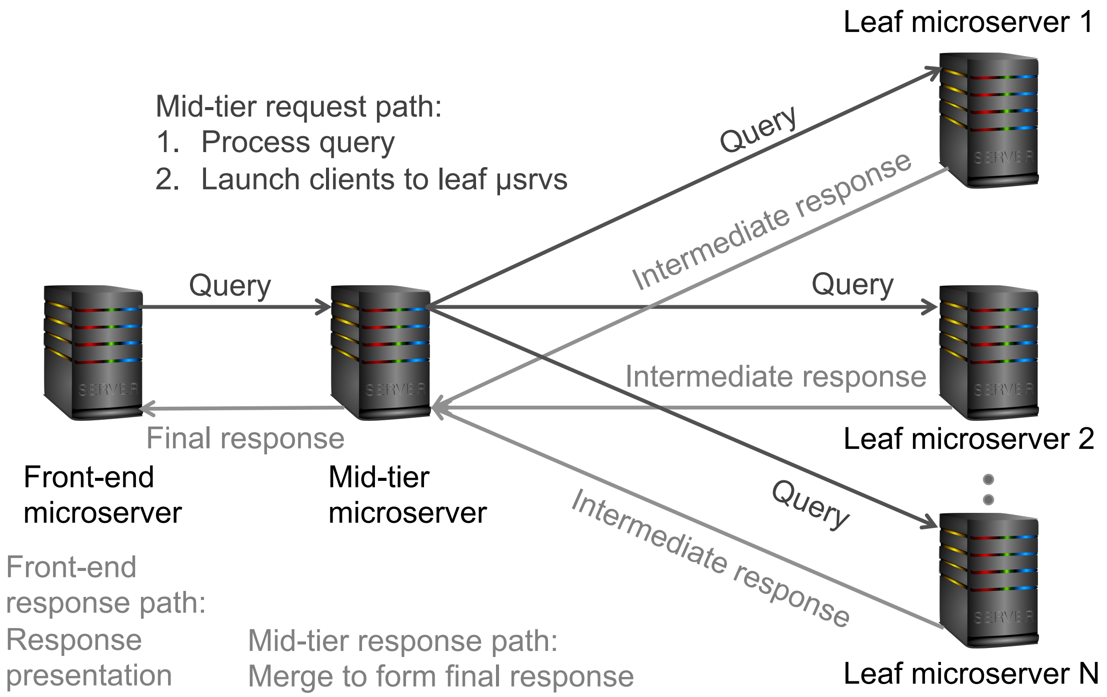


Figure 2.1: A typical web application fan-out: Modern web services are composed of a complex web of microservices that interact via RPCs.

No existing open-source benchmark suite represents the typical three-tier microservice structure employed by modern web services. As a part of this dissertation's software contributions, we introduce a benchmark suite—*μSuite*[2]—of end-to-end web services composed of three microservice tiers that exhibit traits crucial for our study (sub-ms service time, high peak request rate, scalable across cores, mid-tier with fan-out to leaves). We use *μSuite* to study the OS/network performance overheads incurred by mid-tier microservices.

μSuite includes four end-to-end web services that incorporate open-source software: a content-based high dimensional search for image similarity—`HDSearch`, a replication-based protocol router for scaling fault-tolerant key-value stores—`Router`, a service for performing set algebra on posting lists for document retrieval—`Set Algebra`, and a user-based item

[2] At the time, the decision to start my dissertation with a large benchmark development effort was a challenge in and of itself. I knew that I couldn't even start asking the interesting research questions until at least after a year of development work. A year when everyone around me published research papers of their own. Sometimes, these emotional challenges can be greater than the research challenge itself. Ultimately, this "mere development work" served as a foundation for my own research and facilitated other researchers to study microservices. *Moral of the story:* Instead of measuring success in terms of research papers, it is more meaningful and rewarding to measure success in terms of contributions to the scientific community.

recommender system for predicting user ratings—`Recommend`. Each service's constituent microservices' goal is to perform their individual functionality in at most a few single-digit milliseconds for large data sets.

Upon using *µSuite* to study the OS and network overheads faced by microservices, we find that the relationship between optimal OS/network parameters and service load is complex. Specifically, we find that non-optimal OS scheduler decisions can degrade microservice tail latency by up to $\sim 87\%$.

2.1 Prior Work

Existing works on benchmarking latency-critical web services suffer from several drawbacks that make them unsuitable to study microservices (summarized in Table 2.1).

Closed-source. Many works [355, 499, 340, 341, 362, 501, 140] use services internal to companies such as Google or Facebook and hence do not promote further academic study.

Too few latency-critical benchmarks. Some academic studies analyze only one latency-critical benchmark [472, 261], thereby limiting the generality of their conclusions.

Not representative. Some works [184, 504] treat sequential and parallel applications (e.g., SPEC CPU2006 [253] and PARSEC [162]) as web services. However, these applications are not representative of latency-critical web services as they intrinsically vary in terms of continuous activity vs. bursty request-responses, architectural traits, etc.

Monolithic architectures. CloudSuite [221], PerfKit [78], and TailBench [293] are perhaps closest to *µSuite*[3]. CloudSuite focuses on microarchitectural traits that impact throughput for both latency-critical and throughput-oriented services. CloudSuite largely incurs tail latencies of the order of hundreds of milliseconds and is less susceptible to sub-ms–scale OS/network overheads faced by microservices. Moreover, CloudSuite load

[3] Published in 2018, *µSuite* is the first benchmark suite of web services composed of microservices [448]. The DeathStarBench microservice benchmark suite [230] was published after *µSuite*, in 2019. The fundamental difference between the two benchmark suites is that *µSuite* employs a three-tiered microservice architecture, while DeathStarBench builds web services as a Directed Acyclic Graph of microservices. Both types of microservice architectures are used to build real-world web applications used in the industry [230, 445].

Table 2.1: Summary of a comparison of μSuite with prior works: Unlike prior works, μSuite is open-source, has web services composed of microservices, and enables the study of mid-tier microservices.

Prior work	**Open-source**	**μservice arch.**	**Mid-tier study**
SPEC [253]	✓	✗	✗
PARSEC [162]	✓	✗	✗
CloudSuite [221]	✓	✗	✗
TailBench [293]	✓	✗	✗
PerfKit [78]	✓	✗	✗
Ayers et al. [140]	✗	✓	✓
μSuite	✓	✓	✓

testers (YCSB [185] and Faban [36]) model only a *closed-loop system* [501], which is methodologically inappropriate for tail latency measurements [501] due to the coordinated omission problem [463]. BigDataBench [482] also lacks a rigorous latency measurement methodology, even though it uses representative data sets. *μSuite*'s load testers account for these problems and record robust and unbiased latencies. CloudSuite, PerfKit, and TailBench employ monolithic architectures instead of microservice architectures, making them unsuitable to study overheads faced by microservices.

Target only leaves. Several studies target only leaf servers [362, 471, 340, 341, 198, 472, 409, 261] as they are typically most numerous. Hence, conclusions from these works do not readily extend to mid-tier microservers.

Machine-learning based. Recent benchmark suites such as Sirius [247] and Tonic [246] mainly scrutinize ML-based services and incur higher latencies than microservices that *μSuite* targets.

2.2 μSuite: Benchmarks Description

Although there are many open-source microservices, such as Memcached [223], Redis [88], and McRouter [69], that can serve as individual components of a service with a typical three-tier front-end; mid-tier; leaf architecture, there are no representative open-source three-tiered web services composed of microservices. Hence, we develop four web services in *μSuite*, each composed of three microservices. To include web services that dominate today's data centers in *μSuite*, we consider a set of information retrieval (IR)-based internet services based on their popularity [126].

All *μSuite* web services/benchmarks are built using a state-of-the-art open-source RPC platform—gRPC [45].

2.2.1 HDSearch

HDSearch performs content-based image similarity search. Like Google's "find similar images" [98], this service searches an image repository for matches with content similar to a user's query image. This technique entails Nearest Neighbor (NN) matching in a high-dimensional abstract feature space to identify images that have similar content to the query image.

Related work. High dimensional search is an intrinsic part of many user-facing web services, hence its accuracy and performance have been extensively studied. Many prior works [155, 156, 166, 175, 338, 361, 422] improve high dimensional search via tree-based indexing. Since data sets are growing rapidly in both size and dimensionality, tree-based indexing techniques that are efficient for modest dimensionality data sets no longer apply. Instead, hash-based indexing techniques that exploit data locality are now more common [348, 460, 461, 132, 151, 430, 464, 438, 209]. Another indexing algorithm class clusters adjacent data [324, 232, 222, 396, 358, 283, 375, 204]. These primarily theoretical works explore high dimensional search's algorithmic foundations; their contributions are orthogonal to the software structure of a service such as HDSearch.

Service description. HDSearch indexes a corpus of 500K images taken from Google's Open Images data set [76]. Each image in the corpus is represented by a feature vector, an n-dimensional numerical representation of image content. Today, feature vectors summarizing each image are typically obtained from a deep learning technique. We use the Inception V3 neural network [458] implemented in TensorFlow [115] to represent each data set image in the form of a 2048-dimensional feature vector. The data set size is $\sim$ 10 GB.

One can find images similar to a query image by searching the corpus for response images whose feature vectors are near the query image's feature vector [200, 314]. Proximity is measured using distance metrics such as Euclidean or Hamming distance. The goal of `HDSearch`'s constituent microservices is to perform this image search functionality in at most a few single-digit milliseconds for a large image repository. We describe `HDSearch`'s constituent microservices below.

Front-end microservice. `HDSearch`'s front-end presentation microservice is not studied in this work; we describe its components only to provide brief context (Fig. 2.2).

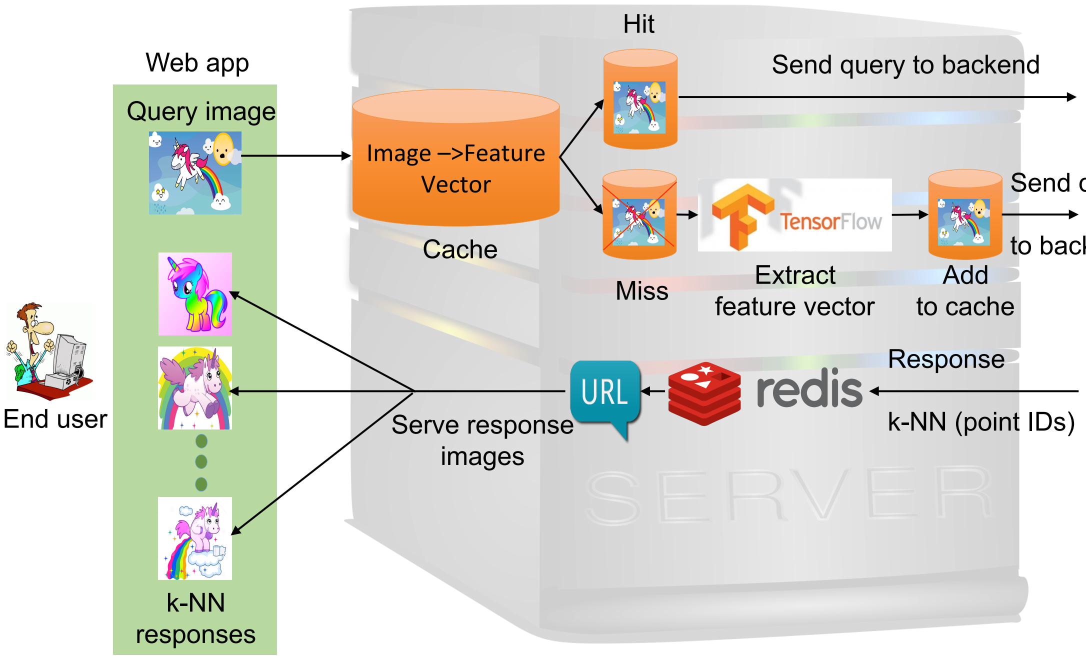


Figure 2.2: HDSearch: Front-end presentation microservice.

Web application. The web application is merely a useful interface that allows the end-user to upload query images to the front-end microserver and view received query responses.

Feature extraction. The query image is initially transformed into a discriminative intermediate feature vector representation. We employ Google's Inception V3 neural network [458], implemented in TensorFlow [115], to extract a feature vector for the query

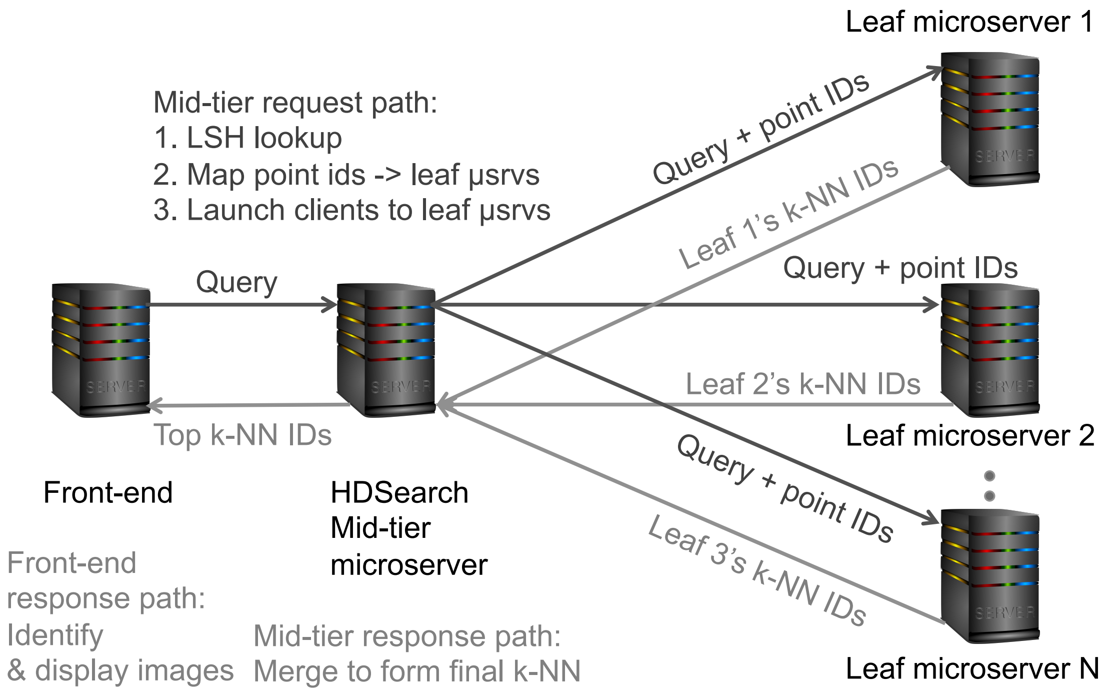


Figure 2.3: HDSearch: Back-end request and response pipelines.

image. This feature vector is sent to the mid-tier microservice to retrieve the IDs of the "k" Nearest Neighbors, i.e., k-NN images. This query's execution in the mid-tier is the object of study for `HDSearch` in this dissertation.

Feature vector caching. To minimize feature vector extraction time, a mapping from images to feature vectors is cached in a `Redis` [88] instance, avoiding repeated feature computations.

Response image look-up. Once the query returns, a second `Redis` [88] instance is consulted to map image IDs to URLs. The presentation microservice then constructs a response web page and returns it to the web application.

Mid-tier microservice. Solving the k-NN problem efficiently is hard due to the *curse of dimensionality* [266], and the problem has been studied extensively [234, 348, 460, 461, 132, 151, 430, 464]. To prune the search space, modern k-NN algorithms use indexing structures, such as Locality-Sensitive Hash (LSH) tables, kd-trees, and k-means clusters to exponentially reduce the search space relative to brute-force linear search.

`HDSearch`'s mid-tier microservice uses LSH, an indexing algorithm that optimally reduces the search space within precise error bounds [197, 234, 348, 460, 461, 132, 151, 430, 464, 438, 209, 133]. We extend the LSH algorithm from the most widely-used open-source k-NN library—the Fast Library for Approximate Nearest Neighbors (FLANN) [376]—into HDSearch's mid-tier. During an offline index construction step, we construct multiple LSH tables for our image corpus. Each LSH table entry contains points that are likely to be near one another in the feature space. Most LSH algorithms use multiple hash tables, and access multiple entries in each hash table, to optimize the performance vs. error trade-off [348].

We extend FLANN's [376] LSH indexes such that the mid-tier microservice does not store feature vectors directly. Rather the LSH tables reference {leaf server, point ID list} tuples, which indirectly refer to feature vectors stored in the leaves.

During query execution, the mid-tier performs look-ups in its in-memory LSH tables to gather potential NN candidates, as shown in Fig. 2.3. It formulates an RPC request to each leaf microserver with a list of point IDs that may be near the query feature vector. Each leaf calculates distances and returns a distance-sorted list. The mid-tier then merges these responses and returns the k-NN across all shards. We quantify HDSearch's accuracy in terms of the cosine similarity between the feature vector it reports as the NN for each query and ground truth established by a brute-force linear search of the entire data set. Various LSH parameters can be tuned based on accuracy and latency requirements. We tune these LSH parameters to target a sub-ms end-to-end median response time with a minimum accuracy score of 93% across all queries.

Leaf microservice. Distance computations are embarrassingly parallel and can be accelerated with SIMD, multi-threading, and distributed computing techniques [200]. We employ all of these. We distribute distance computations over many leaves until the computation time and network communication are roughly balanced. Hence, the mid-tier microservice latency and its ability to fan out RPCs quickly is extremely critical: mid-tier microservice and network overheads limit leaf scalability.

Leaf microservers compare query feature vectors against point lists sent by the mid-

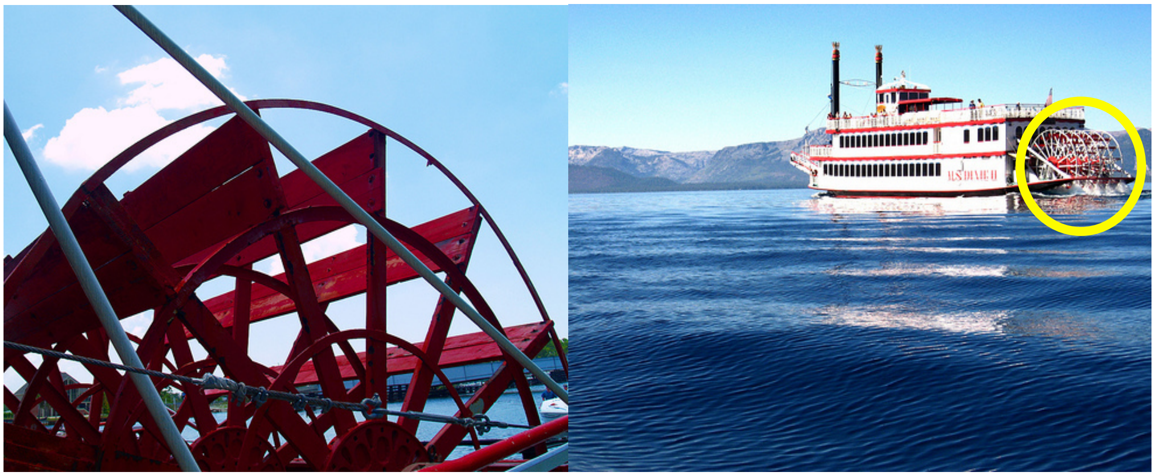


Figure 2.4: An example of HDSearch's request (left) and 1-NN response (right): Response's highlighted circular segment illustrates why the images match.

tier. We use the Euclidean distance metric, which has been shown to achieve a high accuracy [234]. A sample request and response image[4] are shown in Figure 2.4[5].

2.2.2 Router

Memcached-like key-value stores are widely used by web services as they are highly performance efficient and scalable [70]. However, memcached [223] has many drawbacks: (1) its servers are a single point of failure [81] causing frequent fallback to an underlying database access, (2) it is not scalable beyond 200K Queries Per Second (QPS) [70], and (3) it faces network saturation due to network congestion-based timeouts [81]. Memcached must be made as available and performance efficient as the database it fronts. These goals can be achieved by distributing load across many memcached servers via efficient routing. Routing-based redundancy can avoid the failure issue.

[4] These images belong to the Google's `Open Images` data set [76].

[5] Fun fact: After taking a few months to develop HDSearch, when I ran it for the first time, I fed it the query image shown in Fig. 2.4. At the time, I did not realize that the query image was a portion of a paddle wheel. I was also completely expecting a nonsensical response image. (Whoever heard of something working in the first try?!) Both these factors made me utterly convinced that the response image was in no way similar to the request image. I also showed it to my labmate, Vaibhav Gogte, who burst out laughing because he did not identify the similarity either. When I told my Ph.D. advisor, Tom Wenisch, the sad story and showed him the images, he looked extremely confused. It took him a hilarious few minutes to realize that I had not spotted the similarity. This episode resulted in one of Tom's favorite stories—he still likes to tell people about how an ML-based application turned out to be smarter than two of his Ph.D. students.

Related work. McRouter [69] is one such memcached protocol router that helps scale memcached deployments at Facebook. Through efficient routing, McRouter [69] can handle up to 5 billion QPS. It offers features such as connection pooling, prefix routing, replicated pools, production traffic shadowing, and online reconfiguration. We introduce a *μSuite* service called `Router` that includes a simplified subset of McRouter's features, while still drawing insights from McRouter.

Service description. `Router`'s features include (1) routing key-value store queries to memcached deployments, (2) abstracting the routing and redundancy logic from clients, allowing clients to use standard memcached protocols, (3) requiring minimal client modification (i.e., it is a drop-in proxy between the client and memcached hosts), and (4) providing replication-based protocol routing for fault-tolerant memcached deployments.

`Router`'s primary functionality is to route client requests to suitable memcached servers. It supports typical memcached [223] client requests. In this study, we evaluate only `get` and `set` requests. We describe `Router`'s functionality as a series of stages. In the first stage, `Router` parses the clients' requests and forwards them to the route computation code, which uses a proven well-distributed hashing algorithm, SpookyHash [19], to distribute keys from clients' `get` or `set` requests uniformly across destination memcached servers. SpookyHash [19] is a non-cryptographic hash function that is used to produce well-distributed 128-bit hash values for byte arrays of any length. `Router` uses SpookyHash [19] as it (1) enables quick hashing (1 byte/cycle for short keys and 3 bytes/cycle for long keys), (2) can work for any key data type, and (3) incurs a low collision rate. Based on the SpookyHash [19] outcome, `Router` invokes its final stage where it calls internal client code to suitably forward the clients' requests to specific destination memcached servers. The internal client code opens only one TCP connection to a given destination per `Router` thread. All requests sent to that memcached server will share the same connection.

`Router` also provides fault-tolerance for memcached. For large-scale memcached deployments, the frequently-accessed data are read by numerous clients. Too many concurrent

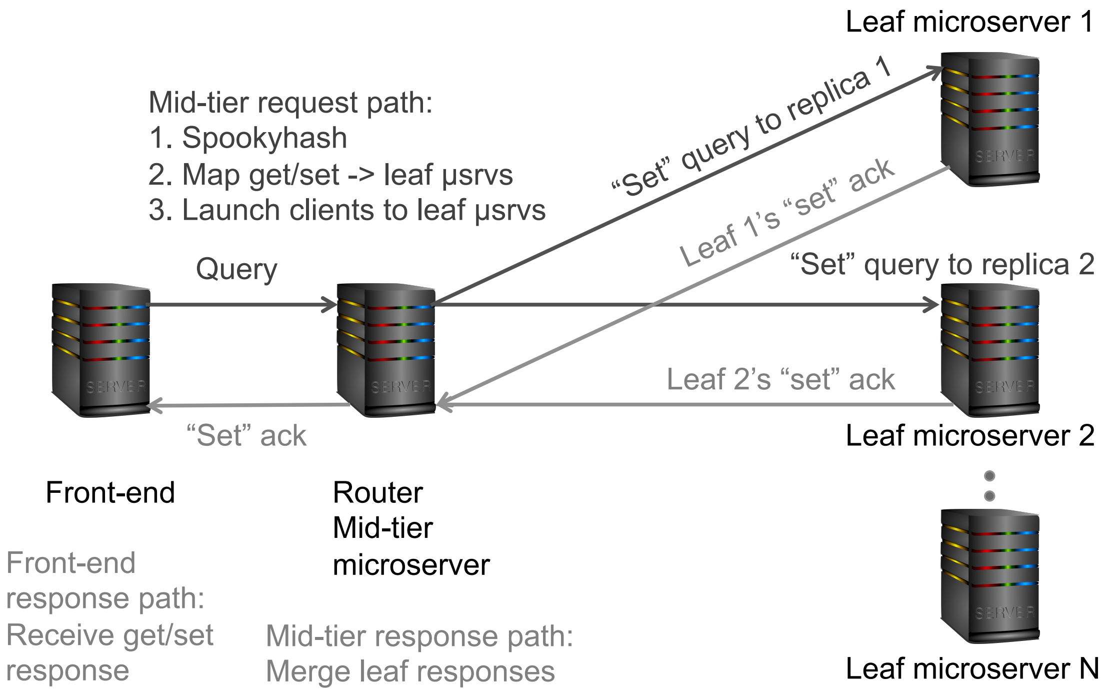


Figure 2.5: Router: Back-end request and response pipelines.

client connections may overwhelm a memcached server. Furthermore, ensuring high availability of critical data even when servers go down is challenging. `Router` uses replicated key-value store data pools, detailed below, to solve both these problems.

Front-end microservice. Our front-end microservice provides a client library that transports memcached `get/set` requests over a gRPC [45] interface. We do not study the front-end in this work. We emulate a large pool of `Router` clients using a synthetic load generator that picks key or key-value pair queries from an open-source "Twitter" data set [221]. The load generator's `get` and `set` request distributions mimic YCSB's Workload A [185] with 50/50 `gets` and `sets`.

Mid-tier microservice. The mid-tier uses SpookyHash [19] to distribute keys uniformly across leaves and then routes `get` or `set` requests as shown in Fig. 2.5. `Router` uses replication both to spread load and to provide fault tolerance. `Router`'s mid-tier forwards `sets` to a fixed number of leaves (i.e., a replication pool; three replicas in our experiments), allowing the same data to reside on several leaves. The mid-tier randomly picks a leaf replica to service `get` requests, balancing load across leaves.

Leaf microservice. The leaf microserver uses `gRPC` [45] to build a communication wrapper around a memcached [223] server process. The leaf microservice is written such that it can handle multiple concurrent requests from several mid-tier microservices. The leaf uses `gRPC` APIs to receive the mid-tier's `get` and `set` queries. It then rewrites received queries to suitably query its local memcached server. The memcached server's responses are then sent to the mid-tier via the `gRPC` [45] response.

2.2.3 Set Algebra

Fast processing of set intersections is a critical operation in many database and information retrieval query processing tasks. For example, in the database realm, set intersections are used for data mining, text analytics, and evaluation of conjunctive predicates. They are also the key operations in enterprise and web search.

Related work. Many open-source web search platforms, such as Lucene [359] and CloudSuite's Web Search [221], perform set intersections for document retrieval. However, these monolithic web searches face response latencies of the order of hundreds of milliseconds as they perform many other tasks (querying a database, scoring page ranks, custom filtering, etc.) apart from set intersections. Hence, these searches are unsuitable for characterizing microservice OS/network overheads. While `Set Algebra` draws algorithmic insights from these works [359, 221], its microservices perform only set intersections to achieve single-digit millisecond tail latencies.

Service description. `Set Algebra` performs document retrieval for web search by performing set intersections on posting lists. The posting list of each term is a sorted list of document identifiers, stored as a skip list [410]. A skip is a pointer i→j between two non-consecutive documents i and j in the posting list. The number of documents skipped between i and j is defined as the skip size. For a term t, the posting list $L(t)$ is a tuple (S_t, C_t) where $S_t = s_1, s_2, ..., s_k$ is a sequence of skips and C_t contains the remaining documents (between skips). These remaining documents can be stored using different compression schemes [507] where decompression can be handled by a separate microservice. Skips are typically used to speed up list intersections.

`Set Algebra` searches through a corpus of 4.3 million WikiText documents (approximately 10 GB in size) randomly drawn from Wikipedia [110] and sharded uniformly across leaves, to return documents containing all search terms to the client. The leaf microservices index posting lists for each term in their sharded document corpus. We reduce leaf computations by excluding extremely common terms, called *stop words*, that have little value in helping select documents matching a user's need from the leaves' inverted index. `Set Algebra` determines a stop list by sorting terms by their collection frequency (the total number of times each term appears in the document collection), treating the most frequent terms as a stop list. Members of the stop list are discarded during indexing.

Front-end microservice. We synthetically emulate multiple clients via a load generator that picks search queries from a query set. Each search query typically spans fewer than ten words [14]. We synthetically generate a query set of 10K queries, based on Wikipedia's word occurrence probabilities [110].

Mid-tier microservice. The mid-tier forwards client queries of search words or terms to the leaves, which return intersected posting lists to the mid-tier, as portrayed in Fig. 2.6. The mid-tier then merges intersected posting lists received from all leaves via set union operations and sends the outcome to the client.

Leaf microservice. The leaf microservice performs the set intersection operations. Leaves hold ordered posting lists as an inverted index where documents are identified via a document ID, and for each term t, the inverted index is a sorted list of all document IDs containing t. Using this representation, the leaves intersect two sets $L1$ and $L2$ using a linear merge by scanning both lists in parallel, requiring an $O(|L1|+|L2|)$ time complexity ("merge" step in merge sort). The resulting intersected posting list is then passed to the mid-tier.

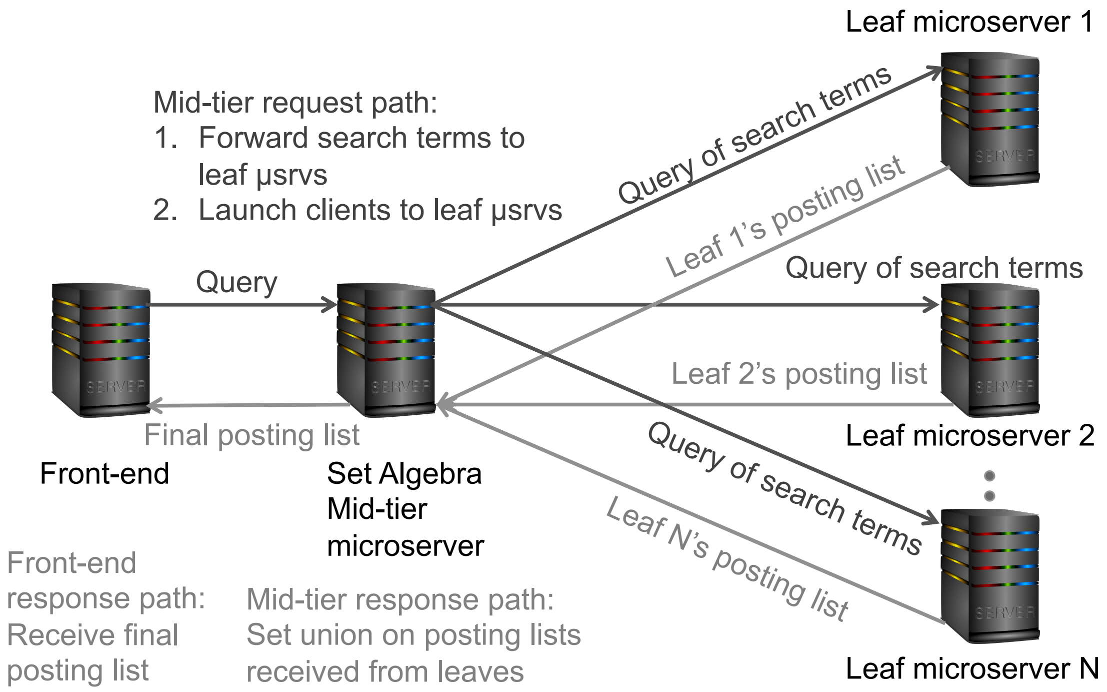


Figure 2.6: Set Algebra: Back-end request and response pipelines.

2.2.4 Recommend.

Recommendation systems help real-world services generate revenue, notably in the fields of e-commerce and behavior prediction [26]. Many web companies use smart recommender engines that study prior user behavior to provide preference-based data, such as relevant job postings, movies of interest, suggested videos, friends users may know, and items to buy.

Related work. Open-source recommendation engines, such as PredictionIO [82], Raccoon [85], HapiGER [47], EasyRec [6], Mahout [68], and Seldon [95], use various recommendation algorithms. However, these recommendation engines lack the distributed microservice structure (i.e, front-end, mid-tier, and leaf) that we study. We build `Recommend` using the state-of-the-art fast, flexible open-source ML library—`mlpack` [189] such that `Recommend` is composed of distributed microservices.

Service description. `Recommend` is a recommendation service that uses numerous users' overall preference to predict user ratings for specific items. For each {user, item} query pair, `Recommend` performs user-based collaborative filtering to predict the user's preference for that item, based on how similar users ranked the item. Collaborative filtering is typically performed on {user, item, rating} tuple data sets. Our collaborative filtering technique has three stages of (1) sparse matrix composition, (2) matrix factorization, and (3) rating approximations for missing entries in the sparse matrix (e.g., a movie that a user has not rated) via a neighborhood algorithm.

Sparse matrix composition. `Recommend`'s data set is 10K {user, item, rating} tuples from the `MovieLens` [245] movie recommendation data set. We represent the data set as a sparsely populated user-item rating matrix $V \in R^{m \times n}$—the *utility matrix*—where m is the number of users and n is the number of items (i.e., movies) in the data set. Hence, V_{ij} (if known), represents the rating of movie j by user i. Each user typically rates a small subset of movies. Many techniques address the *cold start problem* of recommending to a fresh user with no prior ratings. For simplicity, `Recommend` only focuses on users for whom the system has at least one rating.

Matrix factorization. Collaborative filtering often uses matrix factorization. For instance, a matrix factorization model won the Netflix Challenge in 2009 [310]. Matrix factorization's goal is to reduce the sparse user-item rating utility matrix V's dimensionality and to aid similarity identification. We decompose the sparse low-rank matrix V into two "user" and "item" matrices W and H. These decomposed matrices approximate missing values in the utility matrix V.

We employ Non-negative Matrix Factorization (NMF) to decompose V. NMF performs $V \approx WH$ to create two non-negative matrix factors W and H of V. NMF approximately factorizes V into an $m \times r$ matrix W and $r \times n$ matrix H.

$$V = \begin{bmatrix} v_{1,1} & v_{1,2} & \dots & v_{1,n} \\ v_{2,1} & v_{2,2} & \dots & v_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{m,1} & v_{m,2} & \dots & v_{m,n} \end{bmatrix} = WH$$

where,

$$W_{m\times r} = \begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,r} \\ w_{2,1} & w_{2,2} & \dots & w_{2,r} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \dots & w_{m,r} \end{bmatrix}, H_{r\times n} = \begin{bmatrix} h_{1,1} & h_{1,2} & \dots & h_{1,n} \\ h_{2,1} & h_{2,2} & \dots & h_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ h_{r,1} & h_{r,2} & \dots & h_{r,n} \end{bmatrix}$$

Dimension r is V's rank, and it represents the number of similarity concepts NMF identifies [201]. For example, one similarity concept may be that some movies belong to the “comedy” category, while another may be that most users that liked the movie “Harry Potter 1” also liked “Harry Potter 2”. W captures the correlation strength between a row of V and a similarity concept—it expresses how users relate to similarity concepts such as preferring “comedy” movies. H captures the correlation strength of a column of V to a similarity concept—it identifies the extent to which a movie falls in the “comedy” category. The NMF representation, hence, results in a compressed form of the user-item rating utility matrix V.

Neighborhood algorithm. The NMF decomposed matrix is used to approximate missing movie ratings in the user-item rating utility matrix V. We also remember the initial movies that the users rated. We use a neighborhood algorithm, allknn [189], which relies on similarity measures such as cosine, Pearson, and Euclidean, to generate ratings for movies in a user's neighborhood. This algorithm can also be further extended to recommend items

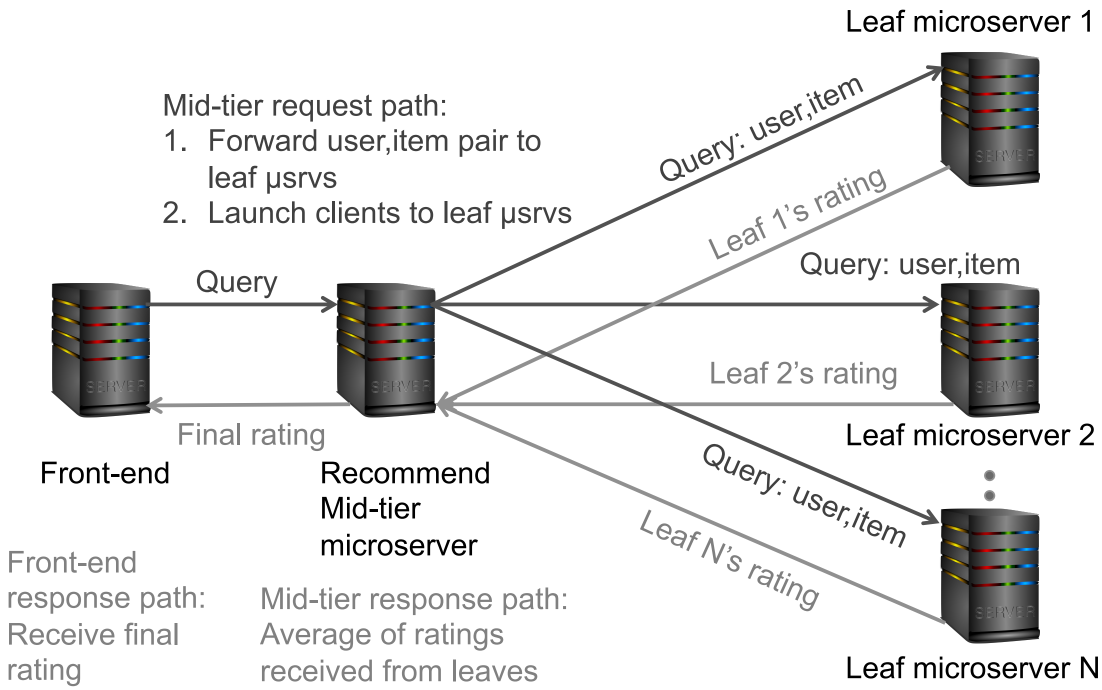

Leaf microserver 1
Mid-tier request path:
1. Forward user,item pair to leaf µsrvs
2. Launch clients to leaf µsrvs
Query: user,item
Leaf 1's rating
Query
Query: user,item
Leaf 2's rating
Final rating
Leaf microserver 2
Front-end
Recommend Mid-tier microserver
Query: user,item
Leaf N's rating
Front-end response path: Receive final rating
Mid-tier response path: Average of ratings received from leaves
Leaf microserver N

Figure 2.7: Recommend: Back-end request and response pipelines.

which were not rated by the user.

Front-end microservice. We emulate multiple `Recommend` clients via a load generator that picks 1K {user, item} query pairs from the `MovieLens` [245] movie recommendation data set. The {user, item} query pairs are different from the user→item rating mappings present in the data set (utility matrix). In other words, the load generator always picks queries from the "empty" cells of the utility matrix V so that we do not test on the same data that `Recommend` trained on.

Mid-tier microservice. `Recommend` uses the mid-tier microservice primarily as a forwarding service, as shown in Fig. 2.7. The mid-tier microserver receives {user, item} query pairs from the client and forwards them to the leaves. Item ratings returned by the leaves are then averaged and sent back to the client.

Leaf microservice. Leaves perform collaborative filtering by first performing sparse matrix composition and matrix factorization offline. During run-time, they perform collaborative filtering on their corresponding matrix V's shard using the allknn neighborhood approach [189], to predict movie ratings. Rating predictions are then sent to the mid-tier.

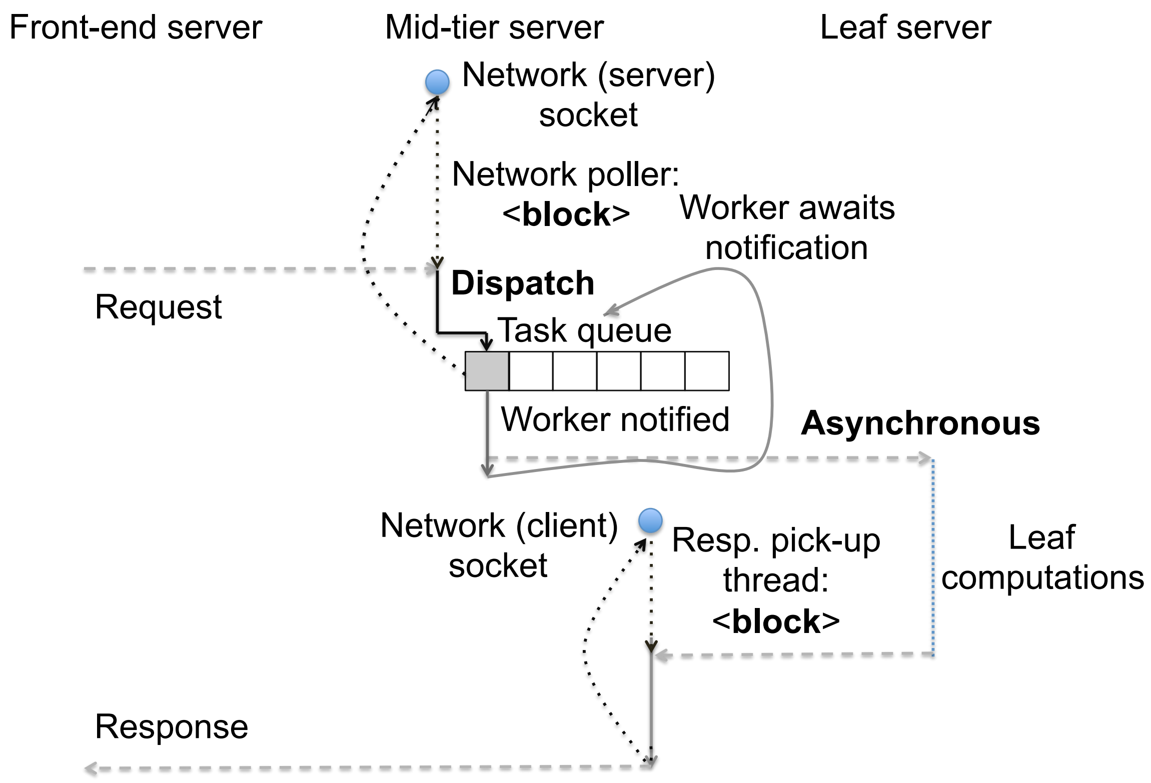


Figure 2.8: μSuite's mid-tier microservice design: μSuite's mid-tier microservices *block* on the front-end network socket, *dispatch* processing of front-end requests, and *asynchronously* communicate with leaf microservices.

2.3 μSuite: Framework Design

We present the software designs used to build μ*Suite*'s mid-tier microservers.

Thread-pool architecture. μ*Suite* has a thread-pool architecture that concurrently executes requests by judiciously "parking" and "unparking" threads to avoid thread creation and deletion overheads. μ*Suite* uses thread-pool architectures (vs. architectures such as thread-per-connection), as thread-pool architectures scale better for microservices [332].

We describe the following μ*Suite* framework designs with the aid of a simple figure (Fig. 2.8) of a three-tier service with a single client, mid-tier, and leaf.

Blocking on the front-end network socket. μ*Suite*'s blocking design is composed of *network poller* threads awaiting new work from the front-end via blocking system calls, yielding the CPU if no work is available. Threads block on I/O interfaces (e.g., `read()` or `epoll()` system calls) awaiting work. Blocking designs conserve precious CPU resources by avoiding wasting CPU time in fruitless poll loops, unlike poll-based designs. Hence, services such as Redis BLPOP [18] employ a block-based design.

Asynchronous communication with leaf microservers. There is no fixed association between an execution thread and a particular RPC—all RPC state is explicit. Asynchronous services are event-based—an event, such as the completion of an outgoing leaf request, arrives on any leaf response reception thread and is matched to a particular parent RPC through a shared data structure. Hence, mid-tier microservers can proceed to process successive requests after sending requests to leaf microservers. We build *μSuite* asynchronously to leverage the greater performance efficiency that asynchronous designs offer compared to synchronous ones [486]. For this reason, several cloud applications such as Apache [13], Azure blob storage [71], Redis replication [89], Server-Side Mashup, CORBA Model, and Aerospike [9] have an asynchronous architecture.

Dispatch-based processing of front-end requests. *μSuite*'s dispatch-based design separates responsibilities between network threads, which accept new requests from the underlying RPC interface, and worker threads, which execute RPC handlers. Network threads dispatch the RPC to a worker thread pool by using producer-consumer task-queues and signalling on condition variables. Workers pull requests from task queues and then process them by forking for fan-out and issuing asynchronous leaf requests. A worker then goes back to blocking on the condition variable to await new work. To aid non-blocking calls to both leaves and front-end microservers, we add another thread pool that exclusively handles leaf server responses. These response threads count down and merge leaf responses. We do not explicitly dispatch responses, as all but the last response thread do negligible work (stashing a response packet and decrementing a counter). Several cloud applications such as IBM's WebSphere for z/OS [65, 254], Oracle's EDT image search [50], Mule ESB [24], Malwarebytes [46], Celery for RabbitMQ and Redis [23], Resque [91] and RQ [92] Redis queues, and NetCDF [227] are dispatch-based.

Table 2.2: Mid-tier microservice processor specification.

Processor features	Specification
Microarchitecture	Intel Gold 6148 CPU “Skylake”
Clock frequency	2.40 GHz
Cores / HW threads	40 / 80
DRAM	64 GB
Network	10 Gbit/s
Linux kernel version	4.13.0

2.4 Methodology

We now describe the experimental setup that we use to characterize *μSuite*’s OS and network overheads.

We characterize each service in terms of its constituent mid-tier and leaf microservices. We use service-specific synthetic load generators that mimic many end-users to issue queries to the mid-tier. These load generators are operated in a closed-loop mode to establish each service’s peak sustainable throughput. We measure tail latencies by operating the load generators in an open-loop mode, selecting inter-arrival times from a Poisson distribution [172]. Load generators are run on separate hardware, and we validated that neither the load generator nor the network bandwidth is a performance bottleneck in our experiments. We average measurements over five trials.

We run experiments on a distributed system of a load generator, a mid-tier microservice, and (1) a four-way sharded leaf microservice for `HDSearch`, `Set Algebra`, and `Recommend` and (2) a 16-way sharded leaf microservice with three replicas for `Router`. Our setup’s hardware configuration is shown in Table 2.2. The leaves run within Linux `tasksets` limiting them to 18 logical cores for `HDSearch`, `Set Algebra`, and `Recommend` and 4 logical cores for `Router`. Each microservice runs on dedicated hardware. The mid-tier is not CPU bound; peak-load performance is limited by leaf CPU.

On this setup, we run load generators in open loop mode to characterize OS and network overheads for various loads. We use the eBPF [66] `syscount` tool to first characterize system

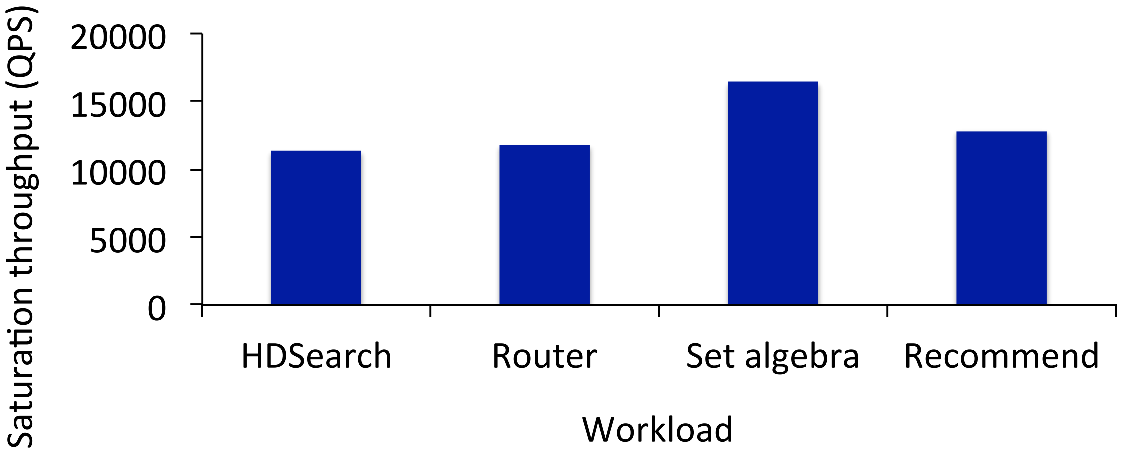


Figure 2.9: Saturation throughput (QPS): μSuite is similar to real-world web services.

call invocations for the mid-tier. We then study request latency breakdowns incurred within the OS (e.g., interrupt handler latency for network-based hard interrupts and scheduler-based soft interrupts, time to switch a thread from "active" to "running" state) using eBPF's [66] `hardirqs`, `softirqs`, and `runqlat` tools. We report network delays in terms of the number of TCP re-transmissions measured using eBPF [66]'s `tcpretrans` tool. Additionally, we use Linux's `perf` utility to profile context switch overheads faced by the mid-tier. We use Intel's HITM (as in hit-Modified) PEBS coherence event, to detect true sharing of cache lines; an increase in HITMs indicates a corresponding increase in lock contention [347].

2.5 Results

2.5.1 Saturation Throughput

Production services typically saturate at tens of thousands of QPS [44]. *μSuite* is representative of production services, achieving a similar saturation throughput range for each of its benchmark services. Using our load generator in closed-loop mode, we measure the saturation throughput for all benchmarks. We find that `HDSearch` saturates at $\sim$ 11.5K QPS, `Router` at $\sim$ 12K QPS, `Set Algebra` at $\sim$ 16.5K QPS, and `Recommend` at $\sim$ 13K QPS, as shown in Fig. 2.9.

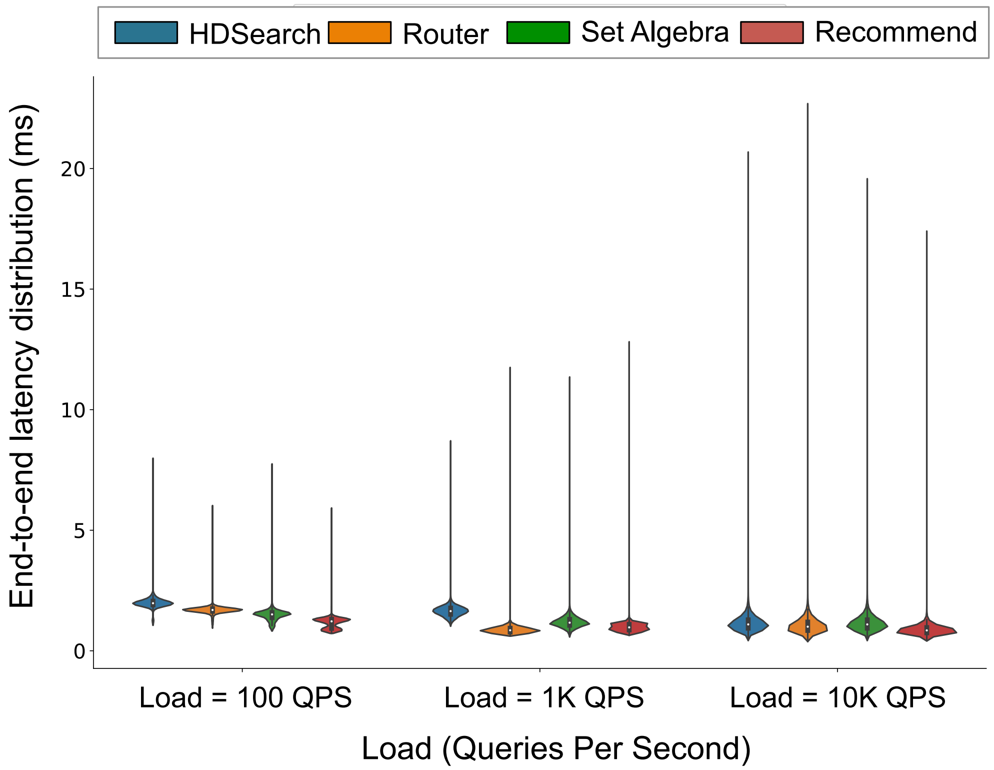


Figure 2.10: End-to-end response latency across different loads for each benchmark: Median latency is higher at low load.

2.5.2 End-to-end Response Latency

Several web services face (1) drastic diurnal load changes [248], (2) load spikes due to "flash crowds" (e.g., traffic after a major news event), or (3) explosive customer growth that surpasses capacity planning (e.g., the Pokemon Go [79] launch). Supporting wide-ranging loads aids rapid web service scale-up. Furthermore, we cannot meaningfully measure latency at saturation, as the offered load is unsustainable and queuing grows without any bounds. Hence, we characterize *μSuite*'s end-to-end (across mid-tier and leaves) response latency vs. load trade-off (Fig. 2.10) across wide-ranging loads up to saturation—100 QPS, 1K QPS, and 10K QPS.

Each end-to-end response latency distribution is portrayed as a violin plot with the bars in the violin centers representing the median latency and the thin black lines representing the higher-order tail latency. While the tail latency increases with an increase in load, we

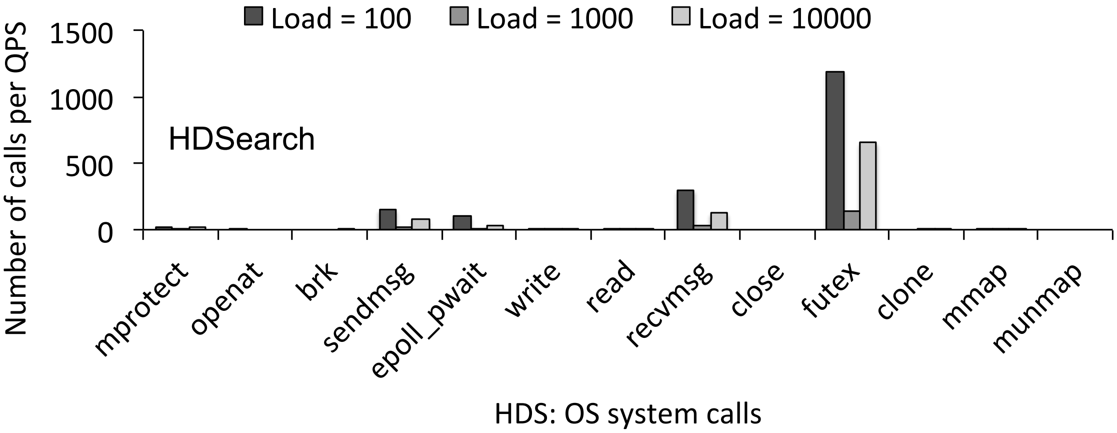


Figure 2.11: HDSearch's counts of OS system call invocations per QPS: The *futex* system call is predominantly invoked.

find that the median latency at 100 QPS is up to 1.45× higher than the median latency at 1,000 QPS since there is better temporal locality at a higher load, and OS and networking performance tend to improve due to batching effects in the networking stack. We explain this behavior further when we subsequently characterize *µSuite*'s OS and network overheads. Additionally, we note that the worst-case end-to-end web service tail latency (across all constituent microservices) is never more than 22 ms for any service; the constituent microservices face a worst-case tail latency of at most a few single-digit milliseconds.

2.5.3 OS and Network Overheads

For each service, we show a breakdown of (1) number of invocations of heavily-invoked system calls, (2) latency distributions across OS and network stacks, (3) network delays due to TCP re-transmissions, and (4) OS-induced effects such as context switches and thread contention. As before, we characterize the OS and network overheads at three distinct loads of 100 QPS, 1,000 QPS and 10,000 QPS.

System call invocations. We first analyze various system call invocation distributions per QPS load level for *µSuite* in Figs. 2.11, 2.12, 2.13, and 2.14. We find that *futex* (*fast userspace mutex*) system calls are invoked most frequently by all services. Our services involve (1) network threads locking the front-end query reception network sockets, (2)

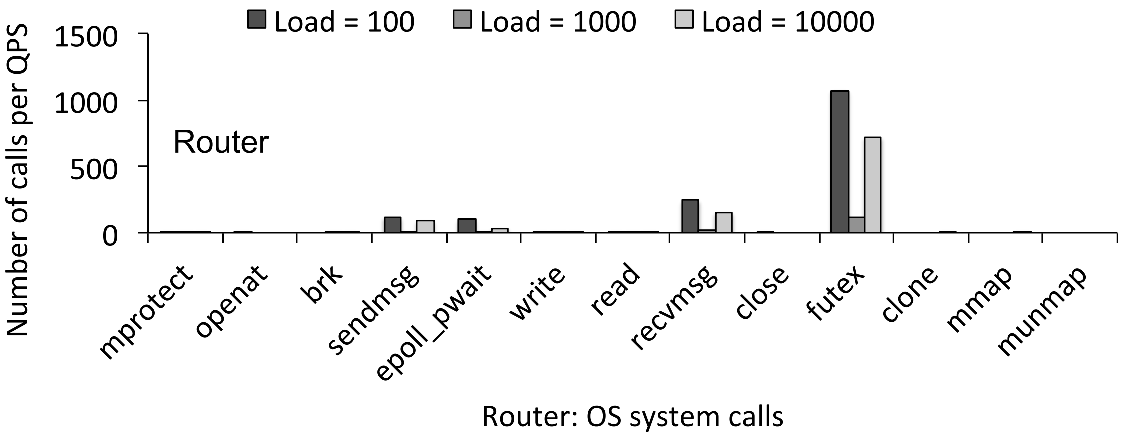


Figure 2.12: Router's counts of OS system call invocations per QPS: The *futex* system call is predominantly invoked.

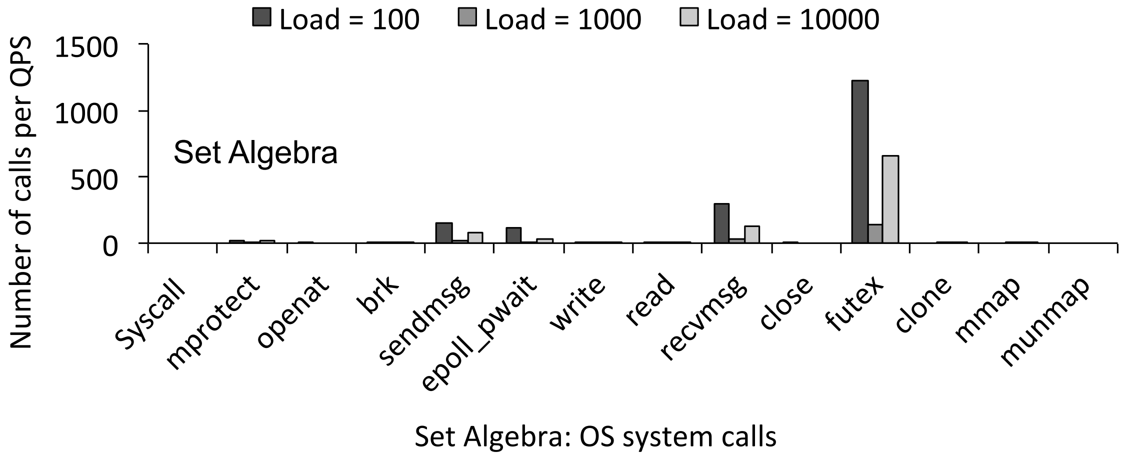


Figure 2.13: Set Algebra's counts of OS system call invocations per QPS: The *futex* system call is predominantly invoked.

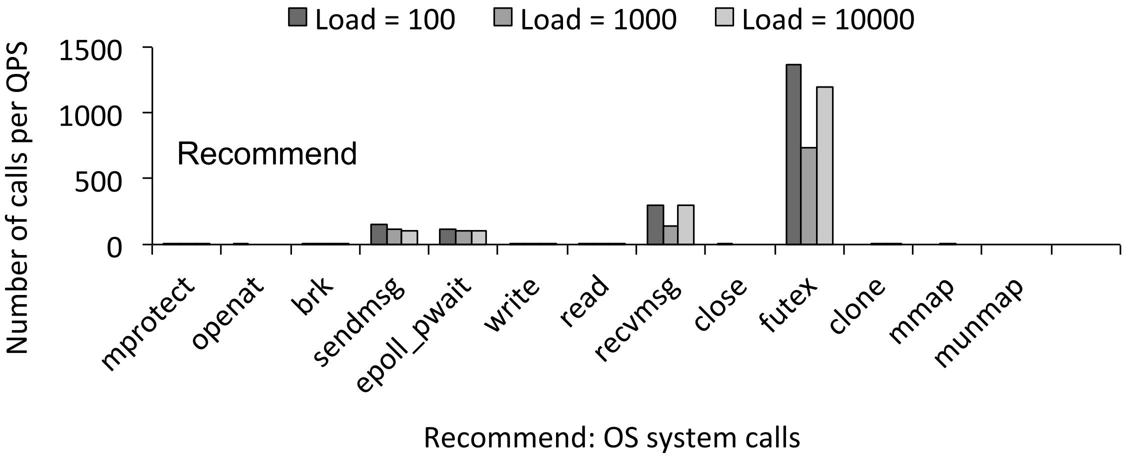


Figure 2.14: Recommend's counts of OS system call invocations per QPS: The *futex* system call is predominantly invoked.

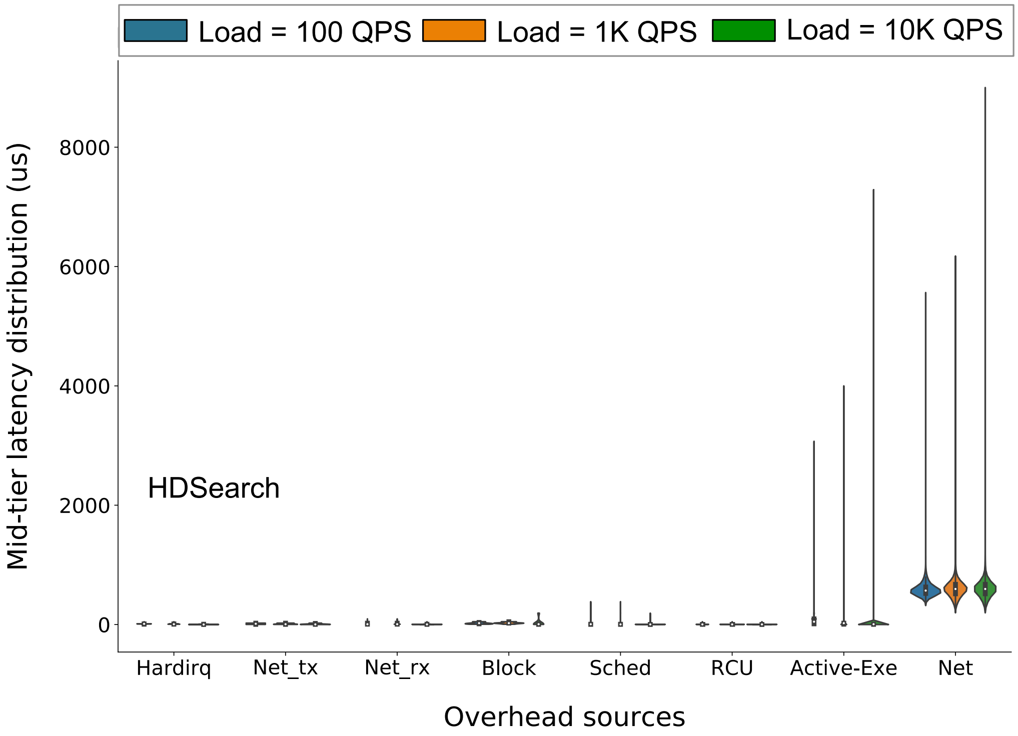


Figure 2.15: HDSearch's breakdown of OS overheads: Time to switch a thread from the *active* to the *running* state is high.

response threads locking the leaf response reception network socket, and (3) worker threads blocking on producer-consumer task queues via condition variables, while awaiting new work. These high-level locking abstractions result in several *futex* system call invocations. Furthermore, we find, much like the end-to-end median latency distribution, `futex` invocations per QPS are higher at low load. At low load, several threads invoke `futex()`, but, only one thread successfully acquires the synchronization object the `futex()` protects (e.g., network socket lock). The remaining threads wake up and try to acquire the network socket lock via further `futex()` calls. Hence, for a small QPS count (low load), a relatively large number of `futex()` calls are invoked by various thread pools.

We also see several *sendmsg*, *recvmsg*, and *epoll_pwait* invocations as these system calls are regularly invoked by blocking network/worker/response threads to send/receive RPCs on the incoming/outgoing network sockets.

Overheads due to OS operations. We next scrutinize latency distributions of fine-

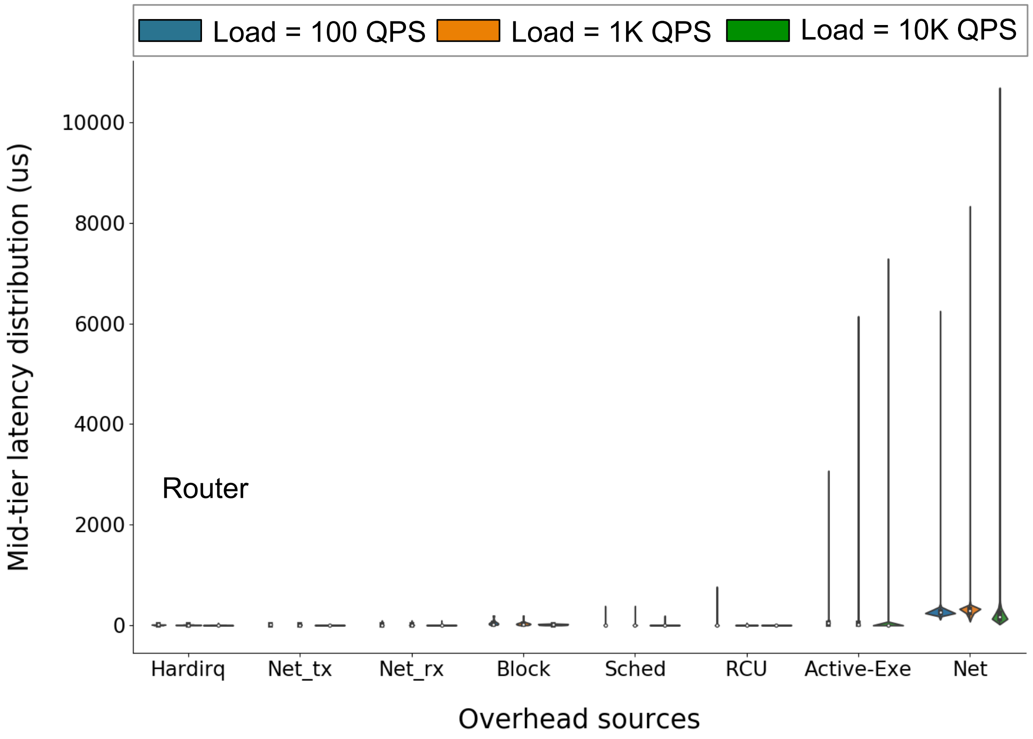


Figure 2.16: Router's breakdown of OS overheads: Time to switch a thread from the *active* to the *running* state is high.

grained OS operations performed while serving mid-tier requests. We study these latencies across various loads for each service, as shown in Figs. 2.15, 2.16, 2.17, and 2.18. In each graph, the X-axis represents various sources of OS overhead, and the Y-axis shows the latency distribution across all mid-tier requests served in a 30s time frame, represented as a violin plot. The various OS overheads are: (1) *Hardirq*—interrupt handler latency while receiving hard network interrupts, (2) *Net_tx*—soft interrupt handler latency while sending network messages, (3) *Net_rx*—soft interrupt handler latency while receiving network messages, (4) *Block*—soft interrupt handler latency while a thread enters the "blocked" state, (5) *Sched*—soft interrupt handler latency while triggering scheduling actions, (6) *RCU*—soft interrupt handler latency for read-copy-updates, (7) *Active-Exe*—time from when a thread enters the *active* or *runnable* state to when it starts running on a CPU, and (8) *Net*—total mid-tier latency.

We find that μ*Suite*'s mid-tier tail latencies arise mainly from the OS scheduler. *Active-*

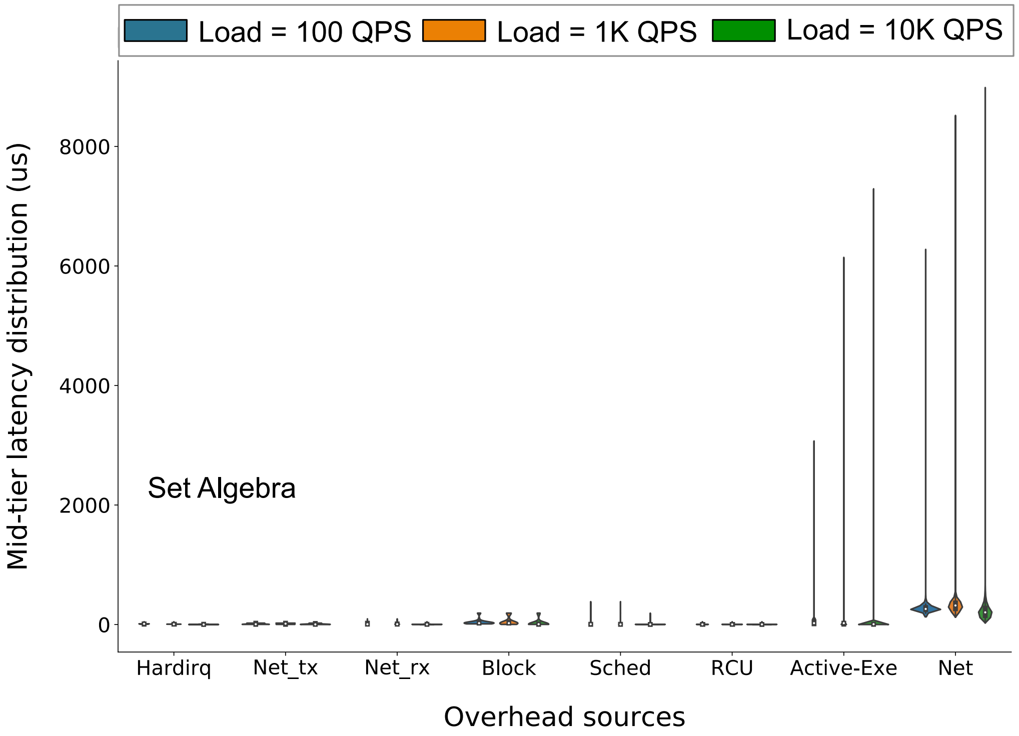


Figure 2.17: Set Algebra's breakdown of OS overheads: Time to switch a thread from the *active* to the *running* state is high.

Exe contributes to mid-tier tails by up to ∼ 50% for HDSearch, ∼ 75% for Router, ∼ 87% for Set Algebra, and ∼ 64% for Recommend. These latencies are a part of thread wakeup delays and can arise from (1) network thread wakeups via interrupts on query arrivals, (2) worker wakeups upon RPC dispatch, or (3) response thread wakeups upon leaf response arrivals. We also see some *Sched* overheads.

Context switch and thread contention overheads. We next analyze *µSuite*'s context switch (CS) and thread contention (HITM) overheads across all three load levels in Fig. 2.19. We find that the CS and HITM overheads are similar for all *µSuite* services ("HDS"—HDSearch, "SA"—Set Algebra, and "Rec"—Recommend)—both overheads increase as load increases. HITM counts are more than CS counts as various threads are woken up when a *futex* returns, and they all contend with each other while trying to acquire a network socket lock. Additionally, we see only a single-digit number of TCP re-transmissions for all services, and hence do not report it.

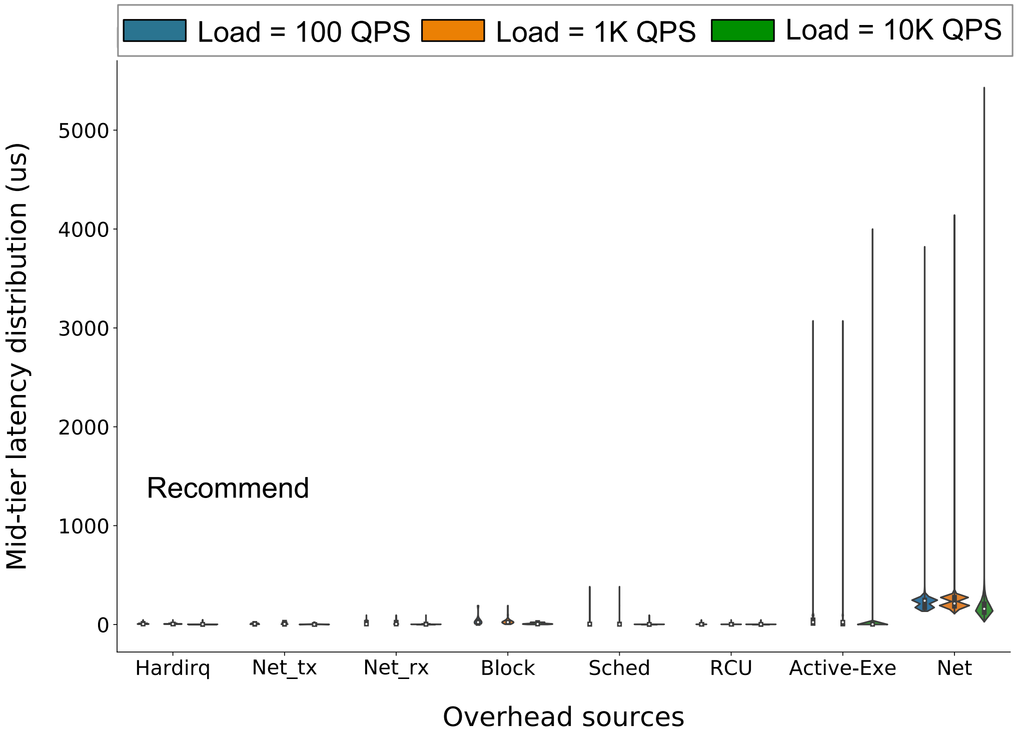


Figure 2.18: Recommend's breakdown of OS overheads: Time to switch a thread from the *active* to the *running* state is high.

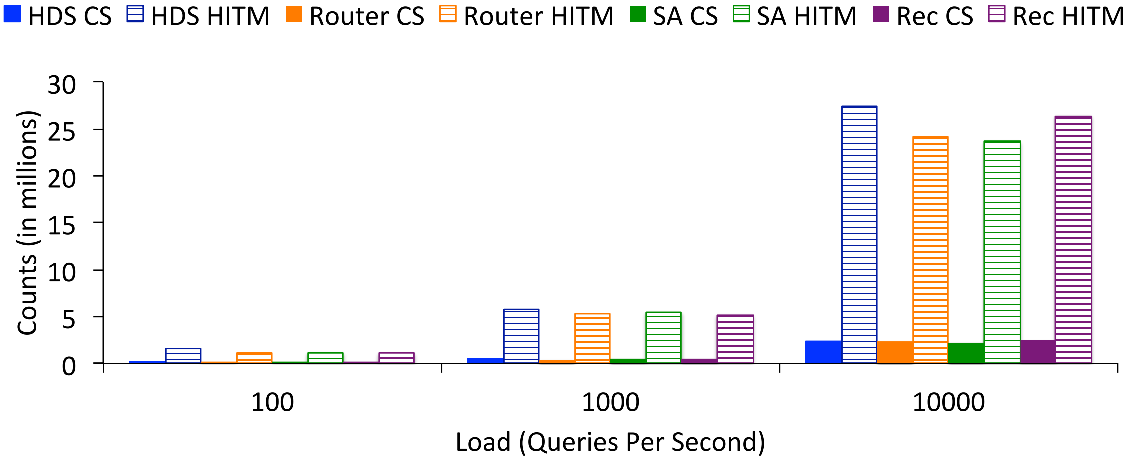


Figure 2.19: Context switches (CS) and thread contention (HITM) incurred (in millions) for each benchmark across diverse loads: Thread contention is a significant overhead.

2.6 Long-Term Impact Potential

We discuss how *μSuite* has facilitated additional research and can do so in the future.

Further study of microservices. When this work was published [448], there existed no representative open-source benchmarks to study microservices. *μSuite* is the first open-source benchmark suite of end-to-end web services composed of microservices [448][6]. *μSuite* can aid future microservice research in both academia and industry. Already, *μSuite* has been used by researchers in academia and industry (e.g., MIT, UIUC, UT Austin, Georgia Tech, Cornell, ARM, and Intel) to study microservice behaviors.

Latency degradation caused by blocking designs. *μSuite* blocks on the front-end request reception network socket and leaf response reception sockets. We choose this design since polling can be prohibitively expensive as it wastes CPU time in fruitless poll loops, degrading a data center's energy efficiency. However, our results show that blocking incurs OS-induced thread wakeup latencies that significantly increase microservice tail latency. The insights that *μSuite* provides about these new threading-induced OS and network overheads in the microservice context leads directly to our contributions in the next chapter. We use *μSuite* to present an automated approach and associated tool that dynamically switches between block- and poll-based threading designs at system runtime.

Thread wakeups due to dispatch-based designs. Thread wakeup latencies that dominate microservice latency tails arise from both (1) network threads picking up requests from the front-end socket and (2) workers waking up to receive dispatched requests. In-line designs that avoid explicit state hand-off and thread-hops to pass work from network to worker threads may avoid expensive thread wakeups. However, in-line models are only efficient at low loads and for short requests where dispatch overheads undermine service times. Since leaf nodes computationally dominate in most distributed services, most mid-tiers can benefit from dispatching. Moreover, if a single in-line thread cannot sustain the service load, multiple in-line threads contending for work will typically outweigh the dispatch design's hand-off costs, which can be carefully tuned. Additionally, in-line models are prone to high queuing, as each thread processes whichever request it receives. In contrast, dispatched

[6] *μSuite* is available at https://github.com/wenischlab/MicroSuite

models can explicitly prioritize requests. *µSuite* will enable researchers to explore (1) policies that trade-off in-line vs. dispatched designs (as presented in Chapter III) and (2) dispatch paradigms to identify optimal microservice dispatch designs.

Thread pool sizing. *µSuite*'s design supports large thread pools that sustain peak loads by "parking" or "unparking" on conditional variables, as needed. However, these large pools can contend on (1) the front-end socket while receiving requests, (2) the producer-consumer task queue while picking up dispatched requests, or (3) the leaf response reception socket. Hence, a user-level thread scheduler that dynamically selects suitable thread pool sizes can reduce thread contention and improve scalability. We use these insights to study the implications of thread pool sizing in Chapter III.

2.7 Chapter Summary

We summarize our contributions as follows:

- ***µ*Suite:** We introduced the first benchmark suite of end-to-end web services composed of microservices built using a state-of-the-art open-source RPC platform. *µSuite* is open source and publicly available [448].

- **A comprehensive characterization of OS and network overheads incurred by microservices.** We used *µSuite* to comprehensively characterize the OS and network overheads incurred by mid-tier microservices. Our characterization revealed that the OS scheduler can significantly influence microservice tail latency. Hence, we identified that threading interactions with the underlying OS and network stacks can significantly influence microservice performance.

- **Facilitating future microservice research.** By demonstrating how *µSuite* can be used to study new overheads that manifest in the microservice regime, this work facilitates future research, with *µSuite* being used by researchers in academia and industry to analyze microservices.

Modern web services have evolved from monolithic systems to instead comprise numerous, distributed microservices interacting via RPCs. Microservices face sub-ms to single-digit millisecond RPC latency goals—much tighter than their monolithic ancestors that must meet latency targets that are of the order of hundreds of milliseconds. Sub-ms–scale OS/network overheads that were once insignificant for such monoliths can now come to dominate in the sub-ms–scale microservice regime. It is therefore vital to characterize the influence of OS and network effects on microservices.

Unfortunately, widely-used academic data center benchmark suites are unsuitable to aid the characterization of OS and network overheads faced by microservices. These benchmark suites use monolithic rather than microservice architectures and largely have request service times that are of the order of hundreds of milliseconds. In this chapter, we investigated how OS and network overheads impact microservice median and tail latency by developing a complete benchmark suite of end-to-end web services composed of microservices, called *μSuite*, that we used to facilitate our study. *μSuite* includes four web services composed of microservices: image similarity search, protocol routing for key-value stores, set algebra on posting lists for document search, and recommender systems.

Our characterization of OS and network overheads incurred by microservices revealed that the relationship between optimal OS and network parameters and service load is complex. Our primary finding is that non-optimal OS scheduler decisions can degrade microservice tail latency by up to $\sim 87\%$. In other words, threading interactions with the underlying OS and network stacks can significantly affect microservice performance. Additionally, by demonstrating how *μSuite* can be used to study new overheads that manifest in the microservice regime, this work enables researchers to further analyze microservices (e.g., performance analyses, power analyses, and micro-architectural overhead analyses).

CHAPTER III

Auto-Tuned Software Threading for Microservices

Threading and concurrency design have been shown to critically affect web service response latency [486, 242]. However, prior works [221] focus on monolithic services, which typically have tail latency SLOs of the order of hundreds of milliseconds [359]. In Chapter II, we demonstrated that threading interactions with the OS and network stacks can have a greater impact on the sub-ms–scale microservice regime.

The new sub-ms–scale overheads that arise from threading interactions with the OS and network stacks are due to today's hardware reality, where network devices have sped up while CPU performance scaling has nearly stopped [196]. As discussed in Chapter II, these sub-ms–scale threading-induced OS and network overheads (e.g., a context switch cost of 5-20 μs [470, 325]) are often insignificant for monolithic services. However, sub-ms–scale microservices differ intrinsically: spurious context switches, network/RPC protocol delays, inept thread wakeups, or lock contention can dominate microservice latency distributions [106]. For example, even a single 20 μs spurious context switch implies a 20% latency penalty for a request to a 100 μs SLO protocol routing microservice [501]. Hence, prior conclusions on threading interactions with the OS and network stacks must be revisited for the microservice regime [145].

In this chapter, as a part of this dissertation's software contributions, we[1] study how

[1] Some of the work in this chapter was performed in collaboration with my Ph.D. advisor, Thomas. F. Wenisch [449]. Therefore, I use the "we" pronoun in this chapter to acknowledge Wenisch's involvement in this work.

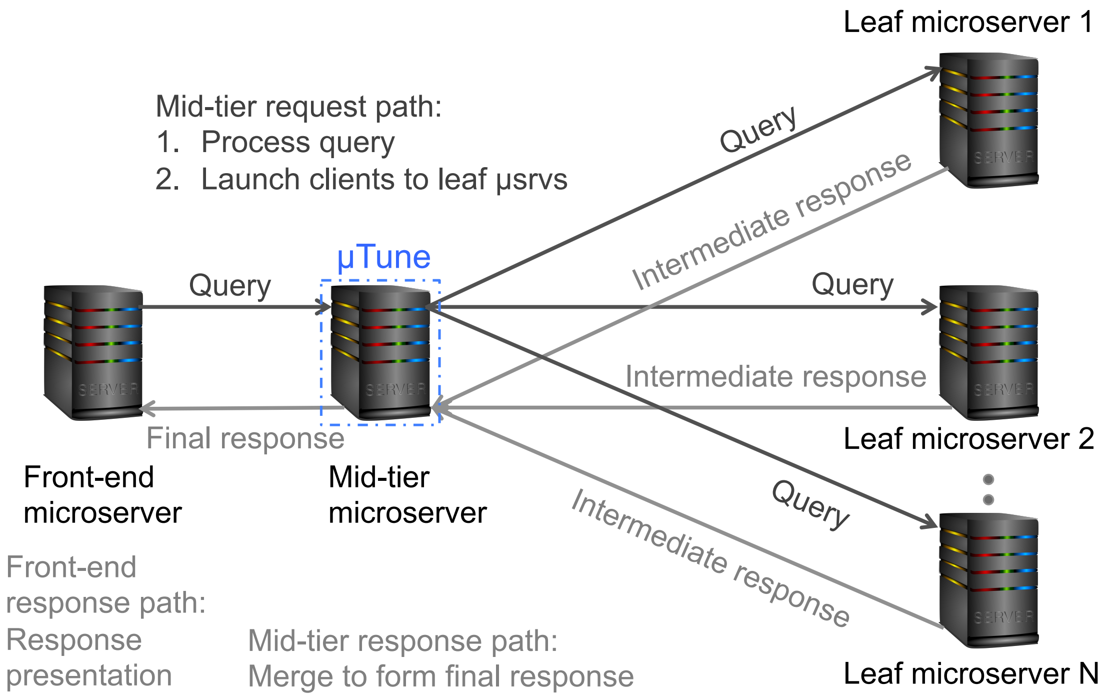


Figure 3.1: A typical web application fan-out.

software threading design affects microservice tail latency and leverage these design effects to dynamically reduce microservice tail latency. We develop a system called *μTune*[2], which features a framework that builds upon open-source RPC platforms [45] to enable microservices to abstract threading model design from microservice code. We analyze a *taxonomy of threading models* enabled by *μTune*. We examine synchronous or asynchronous RPCs, in-line or dispatched RPC handlers, and interrupt- or poll-based network reception. We also vary thread pool sizes dedicated to various purposes (network polling, RPC handling, and response execution). These design axes yield a rich space of microservice architectures that interact with the underlying OS and hardware in starkly varied ways. We find that these threading models often have surprising OS and hardware performance effects including cache locality and pollution, scheduling overheads, and lock contention.

We study *μTune* in the context of four end-to-end web services adopted from *μSuite* [448] (detailed in Chapter II). Each web service is composed of sub-ms microservices that operate

[2] Available at https://github.com/wenischlab/MicroTune

on large data sets. We focus our study on *mid-tier microservers*: widely-used [146] microservices that accept service-specific RPC queries, fan them out to *leaf microservers* that perform relevant computations on their respective data shards, and then return results to be integrated by the mid-tier microserver, as illustrated in Fig. 3.1. As discussed in Chapter II, the mid-tier microserver is a particularly interesting object of study since (1) it acts as both an RPC client and an RPC server, (2) it must manage fan-out of a single incoming query to many leaf microservers, and (3) its computation typically takes tens of microseconds, about as long as OS, networking, and RPC overheads.

We investigate threading models for mid-tier microservices. Our results show that the best threading model depends critically on the offered load. For example, at low loads, models that poll for network traffic perform best, as they avoid expensive OS thread wakeups. Conversely, at high loads, models that separate network polling from RPC execution enable higher service capacity and blocking outperforms polling for incoming network traffic as it avoids wasting precious CPU on fruitless poll loops.

We find that the relationship between optimal threading model and service load is complex—one could not expect a developer to pick the best threading model a priori. Hence, we build an intelligent system that uses offline profiling to automatically adapt to time-varying service load.

μTune's second feature is an adaptation system that determines load via event-based load monitoring and tunes both the threading model (polling vs. blocking network reception; inline vs. dispatched RPC execution) and thread pool sizes in response to load changes. *μTune* reduces tail latency by up to 1.9× over static peak load-sustaining threading models and state-of-the-art adaptation techniques, incurring less than 5% mean latency and instruction overhead. Hence, *μTune* can be used to dynamically reduce sub-ms–scale threading-induced OS/network overheads that dominate in modern microservices.

3.1 Motivation

We motivate the need for a threading taxonomy and adaptation systems that respond rapidly to wide-ranging loads.

As discussed in Chapter II, many prior works have studied leaf servers [471, 340, 341, 198, 472, 409], as they are typically most numerous, making them cost-critical. *Mid-tier* servers [216, 317], which manage both incoming and outgoing RPCs to many clients and leaves, perhaps face greater tail latency optimization challenges, but have not been similarly scrutinized. Their network fan-out multiplies underlying software stack interactions. Hence, performance and scalability depend critically on mid-tier threading model design.

Expert developers extensively tune critical web services via trial-and-error or experience-based intuition [263]. Few services can afford such effort; for the rest, we must appeal to software frameworks and automatic adaptation to improve performance. *μTune* empowers small teams to develop performance efficient mid-tier microservices that meet latency goals without enormous tuning efforts.

3.1.1 The Need for a Threading Model Taxonomy

We develop a structured understanding of rational threading design options for architecting microservices' OS/network interactions in the form of a *taxonomy of threading models*. We study these models' latency effects under diverse loads to offer guidance on when certain models perform best.

Prior works [486, 263, 262, 483, 219] broadly classify monolithic services as: thread-per-request *synchronous* or event-driven *asynchronous*. We note threading design space dimensions beyond these coarse-grain designs. We build on prior works' insights, such as varying parallelism to reduce tail latency [242], to consider a more diverse taxonomy and identify sub-ms performance concerns.

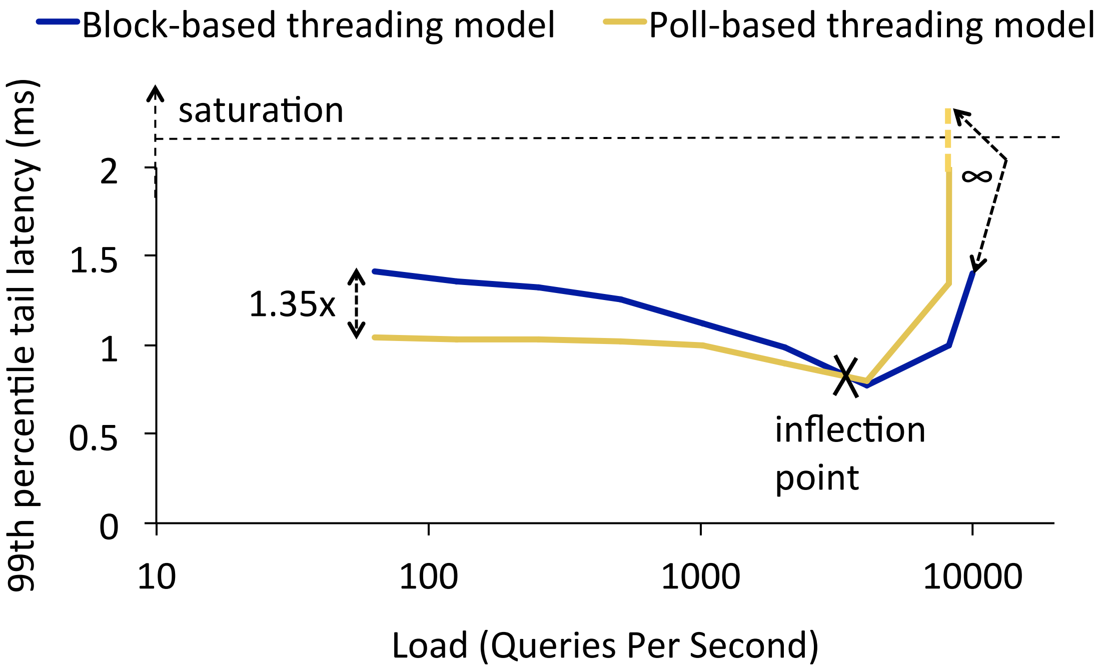


Figure 3.2: $99^{th}\%$ tail latency for an RPC handled by a block-based and poll-based model: Poll-based model reduces latency by 1.35x at low load, and saturates at high load.

3.1.2 The Need for Automatic Load Adaptation

Subtle changes in a microservice's OS interaction (e.g., how it accepts incoming RPCs) can cause large tail latency differences. For example, Fig. 3.2 depicts the $99^{th}\%$ tail latency for a sample RPC handled by an example mid-tier microservice as a function of load. We use a mid-tier microserver with 36 physical cores that dispatches requests received from the front-end to a group of worker threads, which then invoke synchronous calls to the leaf microservices. The yellow line is the tail latency when we dedicate a thread to poll for incoming network traffic in a CPU-unyielding spin loop. The blue line blocks on the OS socket interface awaiting work to the same RPC handler. We see a stark load-based performance inflection even for these simple designs. At low load, the poll-based model gains 1.35× latency as it avoids OS thread wakeups. Conversely, at high load, fruitless poll loops waste precious CPU that might handle RPCs. The poll-based model becomes saturated, with arrivals exceeding service capacity and unbounded latency growth. Blocking-based models conserve CPU and are more scalable.

We assert that such design trade-offs are not obvious: no single threading model is optimal at all loads, and even expert developers have difficulty making good choices. Moreover, most software adopts a threading model at design time and offers no provision to vary it at runtime.

3.1.3 A Microservice Framework

We present a novel microservice framework in *μTune* that abstracts threading design from the RPC handlers. The *μTune* system adapts to the service load by choosing optimal threading models and thread pool sizes dynamically to reduce tail latency.

μTune aims to allow a microservice to be built once and be scalable across wide-ranging loads. Many web services experience drastic diurnal load variations [248]. Others may face "flash crowds" that cause sudden load spikes (e.g., intense traffic after a major news event). New web services may encounter explosive customer growth that surpasses capacity planning (e.g., the meteoric launch of Pokemon Go [79]). Supporting load scalability over many orders of magnitude in a single framework facilitates rapid scale-up of a popular new web service.

3.2 A Taxonomy of Threading Models

A *threading model* is a software system design choice that governs how responsibility for key application functionality will be divided among threads and how the application will achieve request concurrency. Threading models critically impact the service's throughput, latency, scalability, and programmability. We characterize preemptive instead of co-operative (e.g., Node.js [466]) threading models.

3.2.1 Key Dimensions

We identify three key threading model dimensions and discuss their programmability and performance implications.

Synchronous vs. asynchronous communication. Prior works have identified synchronous vs. asynchronous communication as a key design choice in monolithic web services [486, 263, 262, 483, 219]. Synchronous models map a request to a single thread throughout its lifetime. Request state is implicitly tracked via the thread's Program Counter (PC) and stack—programmers simply maintain request state in automatic variables. Threads use blocking I/O to await responses from storage or leaf nodes. In contrast, asynchronous models are event-based—programmers explicitly define state machines for a request's progress [262]. Any ready thread may progress a request upon event reception; threads and requests are not associated.

Programmability: Synchronous models are typically easier to program, as they entail writing straight-forward code without worrying about elusive concurrency-related subtleties. Conversely, asynchronous models require explicit reasoning about request state, synchronization, and races. Ensuing code is often characterized as "spaghetti"—control flow is obscured by callbacks, continuations, futures, promises, and other sophisticated paradigms. Due to this vast programmability gap, we spent *three weeks* implementing synchronous models and *four months* implementing asynchronous models.

Performance: As synchronous models await leaf responses before processing new requests, they face request/response queuing delays, producing worse response latencies and throughput than asynchronous models [374, 483, 219]. Adding more synchronous threads can allay queuing, but can induce secondary bottlenecks, such as cache pollution, lock contention, and scheduling/thread wakeup delays.

Synchronous applications: Several real-world applications such as Azure SQL [16], Google Cloud SQL's Redmine [22, 323], and MongoDB replication [73] are built using the synchronous threading dimension.

Asynchronous applications: Several real-world applications such as Apache [13], Azure blob storage [71], Redis replication [89], Server-Side Mashup [335], CORBA Model, and Aerospike [9] are built using the asynchronous threading dimension.

In-line vs. dispatch-based RPC processing. In in-line models, a single thread manages the entire RPC lifetime, from the point where it is accepted from the RPC library until its response is returned. Dispatch-based models separate responsibilities between network threads, which accept new requests from the underlying RPC interface, and worker threads, which execute RPC handlers.

Programmability: In-line models are simple; thread pools block/poll on the RPC arrival queue and execute an RPC completely before receiving another. Dispatched models are more complex; RPCs are explicitly passed from network to worker threads via thread-safe queues.

Performance: In-line models avoid the explicit state hand-off and thread-hop to pass work from network to worker threads. Hence, they are efficient at low loads and for short requests, where dispatch overheads dominate service times. However, if a single thread cannot sustain the service load, multiple threads contending to accept work typically outweighs hand-off costs, which can be carefully honed. In-line models are prone to high queuing, as each thread processes whichever request it receives. In contrast, dispatched models can explicitly prioritize requests.

In-line applications: Several real-world applications such as Redis [88, 112] and MapReduce workers [199] process requests in-line.

Applications that dispatch: Several real-world applications such as IBM's WebSphere for z/OS [65, 254], Oracle's EDT image search [50], Mule ESB [24], Malwarebytes [46], Celery for RabbitMQ and Redis [23], Resque [91] and RQ [92] Redis queues, and NetCDF [227] dispatch requests to worker threads.

Block- vs. poll-based RPC reception. While the synchronous and in-line dimensions address outgoing RPCs, the block vs. poll dimension concerns incoming RPCs. In block-based models, threads await new work via blocking system calls, yielding the CPU if no work is available. Threads block on I/O interfaces (e.g., `read()` or `epoll()` system calls) awaiting work. In poll-based models, a thread spins in a loop, continuously looking for new work.

Performance: The poll vs. block trade-off is intrinsic: polling reduces latency, while blocking frees a waiting CPU to perform other work. Polling incurs lower latency as it avoids OS thread wakeups [339] to which blocking is prone. However, polling wastes CPU time in fruitless poll loops, especially at low loads. Yet, many latency-sensitive services opt to poll [89], perhaps solely to avoid unexpected hardware or OS actions, such as a slow transition to a low-power mode [148]. Many polling threads can contend to cause pathologically poor performance [277].

Applications that block: Real-world applications such as Redis BLPOP [18] use blocking system calls to await new work.

Applications that poll: Several real-world applications such as Intel's DPDK Poll Driver [83], Redis replication [89], Redis LPOP [67], DoS attacks and defenses [416, 442, 380], and GCP Health Checker [97] use poll-based models to look for new work.

These three threading dimensions lead to eight mid-tier threading models. Within these eight coarse-grain threading model choices, we can further vary individual thread pool sizes.

3.2.2 Synchronous Models

In synchronous models, we create maximally sized thread pools on start-up and then "park" extraneous threads on condition variables to rapidly supply threads as needed without `pthread_create()` call overheads. To simplify our figures, we omit parked threads from them.

The main thread that handles each RPC uses *fork-join* parallelism to fan concurrent requests out to many leaf microservices. The main thread wakes a parked thread to issue each outgoing RPC, blocking on its reply. As replies arrive, these threads decrement a

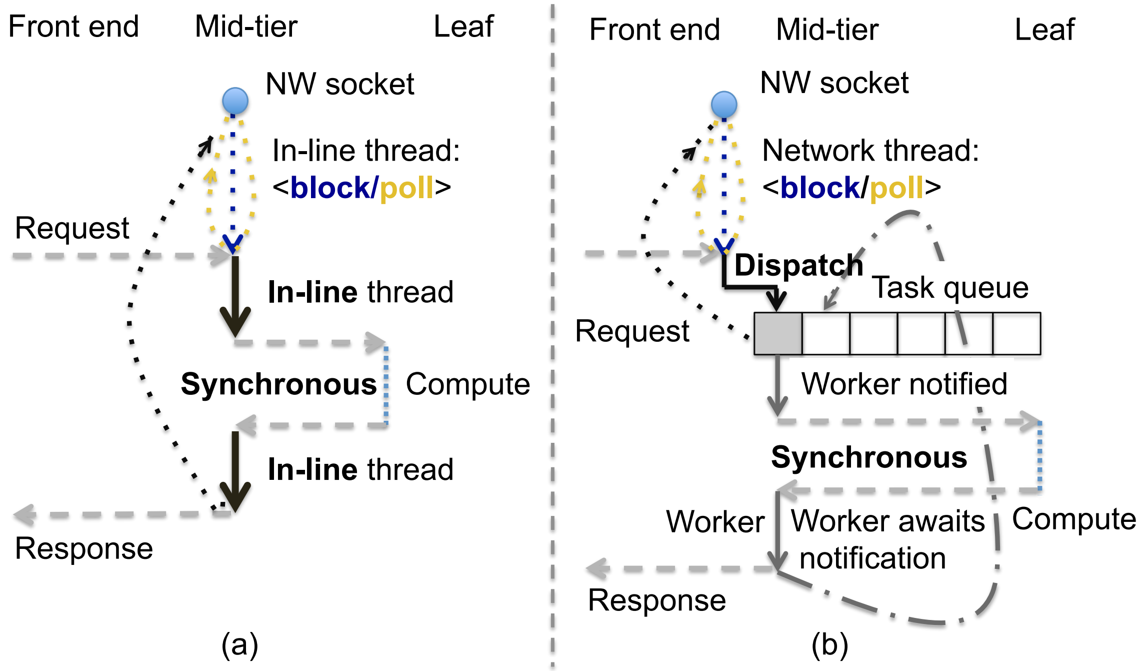


Figure 3.3: Execution of an RPC by (a) SIB/SIP (b) SDB/SDP.

shared atomic counter before parking on a condition variable to track the last reply. The last reply signals the main thread to execute the continuation that merges leaf results and responds to the client.

We next detail each synchronous model with respect to a single RPC execution. For simplicity, our figures show a three-tier service with a single client, mid-tier, and leaf.

Synchronous In-line Block (SIB). This model is the simplest, having only a single thread pool (Fig. 3.3(a)). *In-line* threads *block* on network sockets awaiting work, and then execute a received RPC to completion, signalling parked threads for outgoing RPCs as needed. The thread pool must grow with higher load.

Synchronous In-line Poll (SIP). SIP differs from SIB in that threads poll for new work using non-blocking APIs (Fig. 3.3(a)). SIP avoids blocked thread wakeups when work arrives, however, each in-line thread fully utilizes a CPU.

Synchronous Dispatch Block (SDB). SDB comprises two thread pools (Fig. 3.3(b)). The *network threads* block on socket APIs awaiting new work. However, rather than executing the RPC, they *dispatch* the RPC to a *worker* thread pool by using producer-consumer task-queues and signalling condition variables. Workers pull requests from task queues, and then process them much like the prior in-line threads (i.e., forking for fan-out and issuing synchronous leaf requests). A worker sends the RPC reply to the front-end, before blocking on the condition variable to await new work. Both network and worker pool sizes are variable. Concurrency is limited by the worker pool size. Typically, a single network thread is sufficient.

SDB restricts incoming socket interactions to the network threads, which improves locality; RPC and OS interface data structures do not migrate among threads.

Synchronous Dispatch Poll (SDP). Unlike SDB, in SDP, network threads *poll* on front-end sockets for new work (Fig. 3.3(b)).

3.2.3 Asynchronous Models

Asynchronous models differ from synchronous ones in that they do not tie an execution thread to a specific RPC—all RPC state is explicit. Such models are event-based—an event, such as a leaf request completion, arrives on any thread and is matched to its parent RPC using shared data structures. Hence, any thread may progress any RPC through its next execution stage. This approach requires drastically fewer thread switches during an RPC lifetime. For example, leaf request fan-outs require a simple for loop, instead of a complex *fork-and-wait*.

To aid non-blocking calls to both leaves and front-end servers, we add another thread pool that exclusively handles leaf server responses—the *response* thread pool.

Asynchronous In-line Block (AIB). AIB (Fig. 3.4(a)) uses in-line threads to handle incoming front-end requests and response threads to execute leaf responses. Both thread pools block on their respective sockets awaiting new work. Book-keeping on an RPC's progress is explicit. An in-line thread initializes a data structure for an RPC, records the number of leaf responses it expects, records a functor for the continuation to execute when the last response returns, and then fans leaf requests out in a simple for loop. Responses

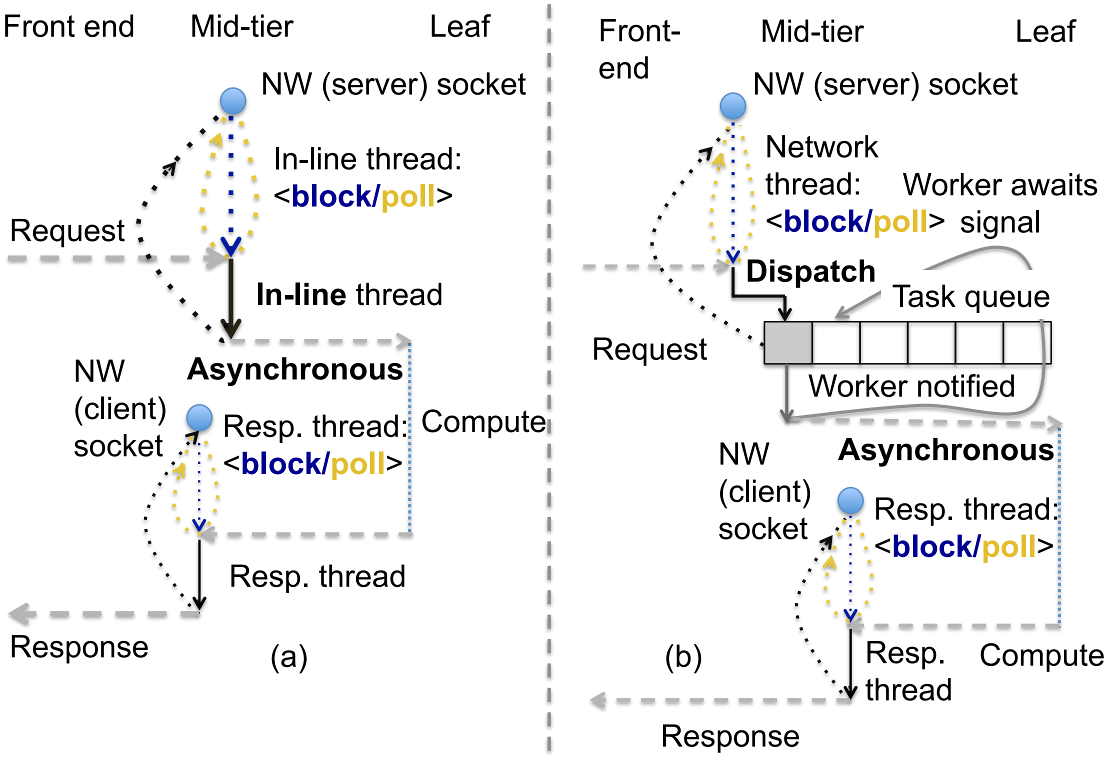


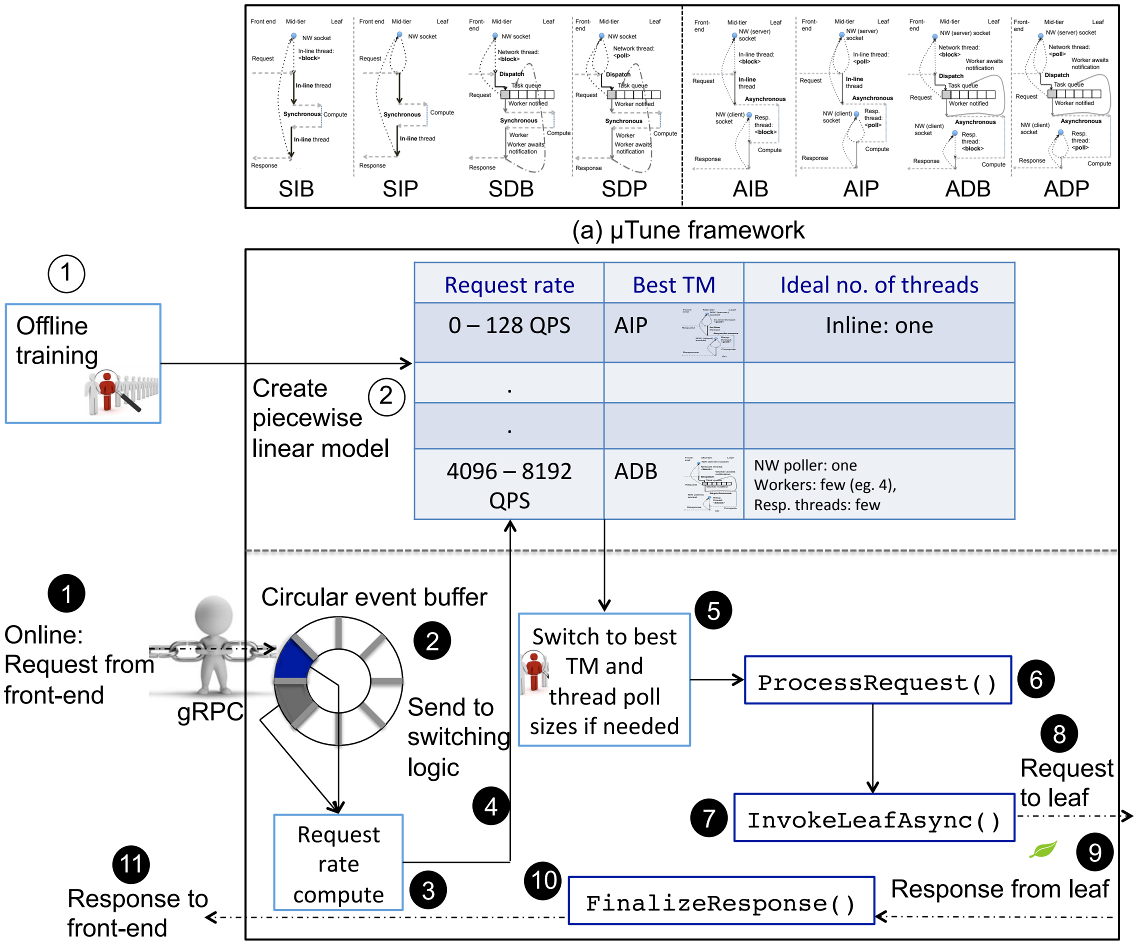
Figure 3.4: Execution of an RPC by (a) AIB/AIP (b) ADB/ADP.

arrive (potentially concurrently) on response threads, which record their results in the RPC data structure and count down until the last response arrives. The final response invokes the continuation to merge responses and complete the RPC.

Asynchronous In-line Poll (AIP). Unlike AIB, in AIP, in-line and response threads *poll* their respective sockets (Fig. 3.4(a)).

Asynchronous Dispatch Block (ADB). In ADB, dispatch enables network thread concentration, improving locality and socket contention (Fig. 3.4(b)). Like SDB, network and worker threads accept and execute RPCs, respectively. Response threads count-down and merge leaf responses. We do not explicitly dispatch responses, as all but the last response thread do negligible work (stashing a response packet and decrementing a counter). All three thread pools vary in size. Typically, one network thread is sufficient, while the other pools must scale with load.

Asynchronous Dispatch Poll (ADP). Unlike ADB, in ADP, network and response



(a) µTune framework

Request rate	Best TM	Ideal no. of threads
0 – 128 QPS	AIP	Inline: one
.		
.		
4096 – 8192 QPS	ADB	NW poller: one Workers: few (eg. 4), Resp. threads: few

(b) Async. µTune's automatic load adaptation system

Figure 3.5: μTune: System design.

threads *poll* for new work (Fig. 3.4(b)).

3.3 μTune: System Design

μ*Tune* has two features: (a) an implementation of all eight threading models, abstracting RPC (OS/network interactions) within the framework (Fig. 3.5(a)); and (b) an adaptation system that judiciously tunes threading models and scales thread pool sizes under changing load (Fig. 3.5(b)). μ*Tune*'s system design challenges include (1) offering a simple interface that abstracts threading design from service code, (2) quick load shift detection for efficient dynamic adaptation, (3) adept threading models switches, and (4) sizing thread pools without thread creation, deletion, or management overheads. We discuss how μ*Tune*'s design meets these challenges.

3.3.1 Framework

μTune abstracts the threading model boiler-plate code from service-specific RPC implementation details, wrapping the underlying RPC API. *μTune* enables characterizing the pros and cons of each model.

μTune offers a simple abstraction where service-specific code must implement RPC execution interfaces. For synchronous modes, the service must supply a `ProcessRequest()` method per RPC. `ProcessRequest()` is invoked by in-line or worker threads. This method prepares a concurrent outgoing leaf RPC batch and passes it to `InvokeLeaf()`, which fans it out to leaf nodes. `InvokeLeaf()` returns to `ProcessRequest()` after receiving all leaf replies. The `ProcessRequest()` continuation merges replies and forms a response to the client.

For asynchronous modes, *μTune*'s interface is slightly more complex. Again, the service must supply `ProcessRequest()`, but, it must explicitly represent RPC state in a shared data structure. `ProcessRequest()` may make one/more calls to `InvokeLeafAsync()`. These calls are passed an outgoing RPC batch, a tag identifying the parent RPC, and a `FinalizeResponse()` callback. The tags enable request-response matching. The last arriving response thread invokes `FinalizeReponse()`, which may access the RPC data structure and response protocol buffers from each leaf. A developer must ensure thread-safety. `FinalizeResponse()` may be invoked any time after `InvokeLeafAsync()`, and may be concurrent with `ProcessRequest()`. Reasoning about races is the key challenge of the asynchronous RPC implementation.

3.3.2 Automatic Load Adaptation

A key feature of *μTune* is its ability to automatically select among threading models in response to load, thereby relieving developers of the burden of selecting a threading model a priori.

Synchronous vs. asynchronous microservices have a major programmability gap. Although *μTune*'s framework hides some complexity, it is not possible to switch automatically and dynamically between synchronous and asynchronous modes, as their API and application code requirements necessarily differ. If an asynchronous implementation is available, it will outperform its synchronous counterpart. Hence, we build *μTune*'s adaption separately for synchronous and asynchronous models.

μTune picks the latency-optimal model among the four options (in-line vs. dispatch; block vs. poll) and tunes thread pool sizes dynamically with load to reduce the $99^{th}\%$ tail latency. It monitors service load and (a) picks a latency-optimal threading model and (b) scales thread pools by parking/unparking threads. Both adaptations use profiles generated during an offline training phase. We describe the training and adaptation steps shown in Fig. 3.5(b).

Training phase. (1) During offline characterization, we use a synthetic load generator to drive specific load levels for sustained intervals. During these intervals, we vary threading model and thread pool sizes and observe $99^{th}\%$ tail latencies. The load generator then ramps load incrementally, and we re-characterize at each load step. (2) *μTune* then builds a piece-wise linear model relating offered load to observed tail latency at each load level.

Run-time adaptation. (1) *μTune* uses event-based windowing to monitor loads offered to the mid-tier at runtime. (2) *μTune* records each request's arrival timestamp in a circular buffer. (3) It then estimates the inter-arrival rate using the circular buffer's size and the youngest and oldest recorded timestamps. The adaptation system's responsiveness can be tuned by adjusting the circular buffer's size. Careful buffer size tuning can ensure quick, efficient adaptation by avoiding oscillations triggered by outliers. Event-based monitoring can quickly detect precipitous load increases. (4) The inter-arrival rate estimate is then given as input to the switching logic that interpolates within the piece-wise linear model to estimate tail latency for each configuration under each model and thread pool size. (5) *μTune* then transitions to the predicted lowest latency threading model. *µTune* transitions by "parking" the current threading model and "unparking" the newly selected model using its framework abstraction and condition variable signaling, to (a) alternate between poll/block socket reception, (b) process requests in-line or via predefined task queues that dispatch requests to workers, or (c) park/unpark various thread pools' threads to handle new requests. Successive asynchronous requests invoke the (6) `ProcessRequest()`, (7) `InvokeLeafAsync()`, and (10) `FinalizeResponse()` pipeline as dictated by the new threading model. In-flight requests during transitions are handled by the earlier model.

3.4 Implementation

Framework. *µTune* builds upon Google's open-source `gRPC` library [45], which uses *protocol buffers* [84]—a language-independent interface definition language and wire format—to exchange RPCs. *µTune*'s mid-tier framework uses `gRPC`'s C++ APIs: (1) `Next()` and `AsyncNext()` with a zero second timeout are used to respectively block or poll for client requests, (2) `RPCName()` and `AsyncRPCName()` are called via `gRPC`'s `stub` object to send requests to leaf microservices. *µTune*'s asynchronous models explicitly track request state using finite state machines. Asynchronous models' response threads call `Next()` or `AsyncNext()` for block- or poll-based receive.

µTune uses `AsyncRPCName()` to handle asynchronous requests to leaf microservices. For asynchronous *µTune*, the leaf microservices must use `gRPC`'s `Next()` API variants to accept requests through explicitly managed completion queues; for synchronous, the leaf microservices can use underlying synchronous `gRPC` abstractions.

Using *µTune*'s framework to build a new microservice is simple, as only a few service specific functions must be defined. We took $\sim$2 days to build each service in Sec. 3.5.

Automatic load adaptation. We construct the piece-wise linear model of tail latency by averaging five 30s measurements of each threading model-thread pool pair at varying loads. *µTune*'s load detection relies on a thread-safe circular buffer built using scoped locks and condition variables. The circular buffer capacity is tuned to quickly detect load transients while avoiding oscillation. We find that a 5-entry circular buffer works best in all our experiments. *µTune*'s switching logic uses C++ atomics and condition variables to switch among threading models seamlessly. *µTune*'s adaptation code is 2371 LOC of C++.

3.5 Experimental Setup

We characterize threading models in the context of four information retrieval web services' mid-tier and leaf microservices adopted from *µSuite* [448]. Specifically, we study threading models in the context of a content-based high dimensional search for image similarity—`HDSearch`, a replication-based protocol router for scaling fault-tolerant key-value stores—`Router`, a service for performing set algebra on posting lists for document retrieval—`Set Algebra`, and a user-based item recommender system for predicting user ratings—`Recommend` (detailed in Chapter II).

We use a load generator that mimics many clients to send queries to each mid-tier microservice under controlled load scenarios. It operates in a closed-loop mode while measuring peak sustainable throughput. We measure end-to-end (across all microservices) $99^{th}\%$ latency by operating the load generator in open-loop mode with Poisson inter-arrivals [172]. The load generator runs on separate hardware and we validated that the load generator and network bandwidth are not performance bottlenecks[3].

Our distributed system has a load generator, a mid-tier microservice, and (1) a four-way sharded leaf microservice for `HDSearch`, `Set Algebra`, and `Recommend` and (2) a 16-way

[3] Fun (?) fact: It took nearly 1.5 months of debugging a large code base to write this single sentence in this chapter. For a while, network bandwidth *was* the performance bottleneck, causing large queue build-ups. Since our system is multi-tiered, I had to painstakingly debug each entry and exit point (building novel, approximate clock synchronization mechanisms along the way) to somehow "look" at invisible queuing effects. My Ph.D.'s lowest point was when I called the vertical blank space between two consecutive lines of code the "problem" to Tom Wenisch, causing him to give me the most incredulous look I've seen him wear. That's what elusive queuing effects do—they make you lose your mind in your quest for them. *Moral of the story*: With research, sometimes it takes months of effort to write little to no related content in your manuscript. But, that does not mean those efforts are meaningless; you grow as a researcher even if it seems like the world will never read about your painstaking efforts. Research growth is not measured by merely the papers written.

Table 3.1: Mid-tier microservice processor specification.

Processor features	Specification
Microarchitecture	Intel Xeon E5-2699 v3 “Haswell”
Clock frequency	2.30 GHz
Cores / HW threads	36 / 72
DRAM	500 GB
Network	10 Gbit/s
Linux kernel version	3.19.0

sharded leaf microservice with three replicas for `Router`. The hardware configuration of our measurement setup is in Table 3.1. The leaf microservers run within Linux `tasksets` limiting them to 20 logical cores for `HDSearch`, `Set Algebra`, and `Recommend` and 5 logical cores for `Router`. Each microservice runs on a dedicated machine. The mid-tier is not CPU bound; saturation throughput is limited by leaf server CPU.

To test the effectiveness of μ*Tune*’s load adaptation system and measure its responsiveness to load changes, we construct the following load generator scenarios. (1) *Load ramp*: We increase offered load in discrete 30s steps from 20 Queries Per Second (QPS) up to a microservice-specific near-saturation load. (2) *Flash crowd*: We increase load suddenly from 100 QPS to 8K/13K QPS. In addition to performance metrics measured by our load generator, we also report OS and microarchitectural statistics. We use Linux’s `perf` utility to profile the number of cache misses and context switches incurred by the mid-tier microservice. We use Intel’s HITM (hit-Modified) PEBS coherence event to detect true sharing of cache lines; an increase in HITM events indicates a corresponding increase in lock contention [347]. We measure thread wakeup delays (reported as latency histograms) using the BPF run queue (scheduler) latency tool [66].

3.6 Evaluation

We first characterize our threading models. We then compare μ*Tune* to state-of-the-art adaptation systems.

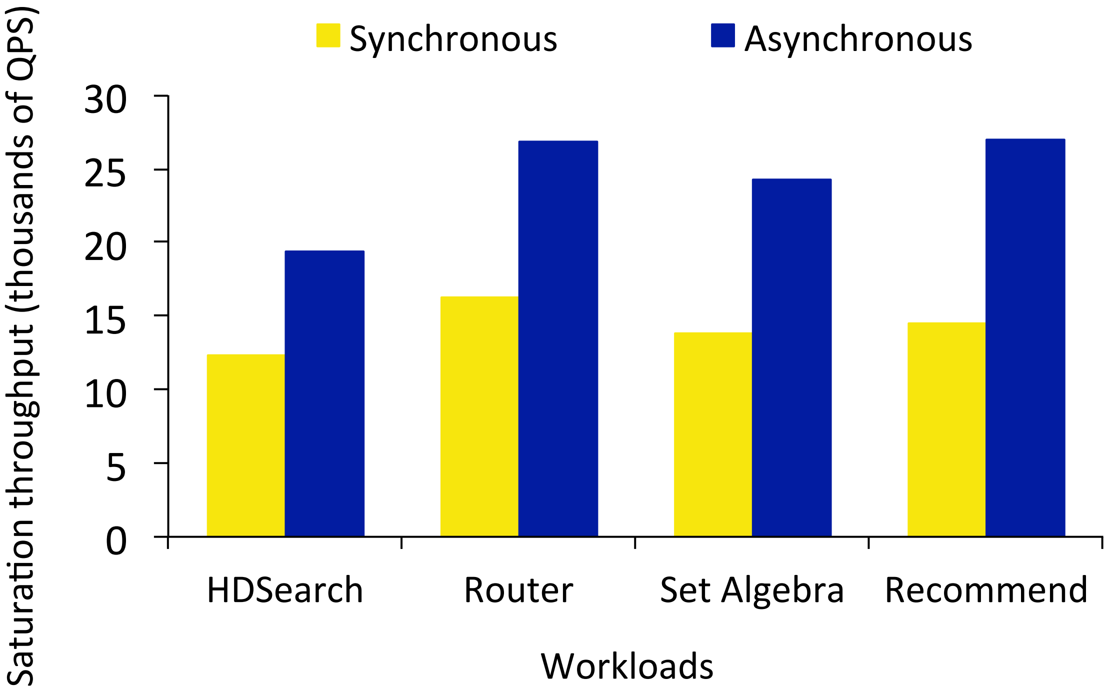


Figure 3.6: Synchronous vs. asynchronous model's saturation throughput: The asynchronous model performs better by 42% on average.

3.6.1 Threading Model Characterization

We explore microservice threading models by first comparing synchronous vs. asynchronous performance. We then separately explore trade-offs among the synchronous and asynchronous models to report how the latency-optimal threading model varies with load.

3.6.1.1 Synchronous vs. Asynchronous

The synchronous vs. asynchronous trade-off is one of programmability vs. performance. It would be unusual for a development team to construct both microservice designs; if the team invests in the asynchronous design, it will almost certainly be more performance efficient. Still, our performance study serves to quantify this gap.

Saturation throughput. We record saturation throughput for the "best" threading model at saturation (SDB/ADB). In Fig. 3.6, we see that the greater asynchronous efficiency improves saturation throughput for μ*Tune*'s asynchronous models, a 42% mean throughput improvement across all services. However, we spent 5× more effort to build, debug, and

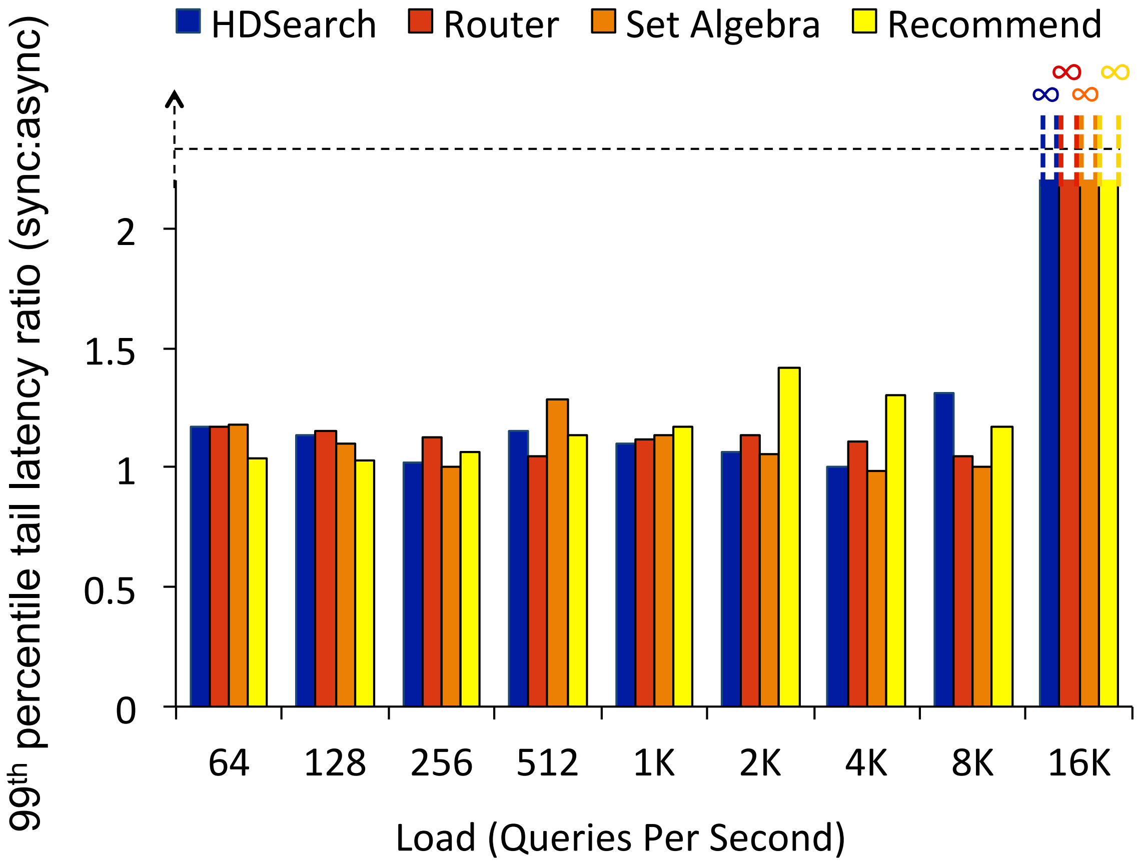
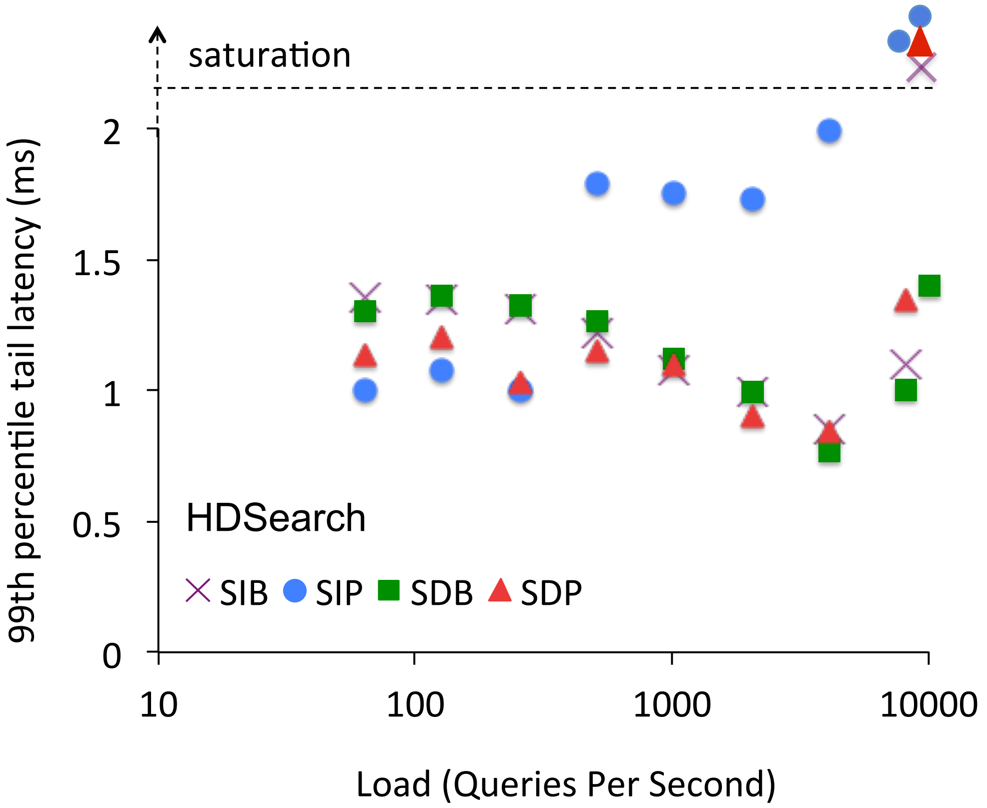


Figure 3.7: Ratio of the best synchronous model's latency to the best asynchronous model's latency: The best asynchronous model is faster by a mean 12% at the loads that are achievable by the best synchronous model, and infinitely faster at higher loads.

tune the asynchronous models.

Tail latency. Latency cannot meaningfully be measured at saturation, as the offered load is unsustainable and queuing delays grow unbounded. Hence, we compare tail latencies at load levels from 64 QPS up to synchronous saturation. In Fig. 3.7, we show the best sync-to-async ratio of $99^{th}\%$ tail latency across all threading models and thread pool sizes at each load level; we study inter-model latencies later. We find asynchronous models reduce tail latency up to $\sim 1.3\times$ (mean of $\sim 1.12\times$) over synchronous models (for loads that synchronous models can sustain; i.e., $\leq$ 8K). This substantial tail latency gap arises because asynchronous models prevent long queuing delays.



QPS	64	128	256	512	1K	2K	4K	8K	10K
SIB	1.4	1.3	1.3	**1**	**1**	**1**	1.1	1.1	∞
SIP	**1**	**1**	**1**	1.6	1.6	1.9	2.6	∞	∞
SDB	1.3	1.3	1.3	1.1	1.1	1.1	**1**	**1**	**1**
SDP	1.2	1.1	**1**	**1**	**1**	**1**	1.1	1.4	∞

Figure 3.8: Graph: Latency vs. load trade-off for `HDSearch`'s synchronous models. Table: Latencies at each load normalized to the best latency for that load: No threading model is always the best.

3.6.1.2 Synchronous models

We study the tail latency vs. load trade-off for services built with μ*Tune*'s synchronous models. We show a cross-product of the threading taxonomy across loads for `HDSearch` in Fig. 3.8. Each data point is the best 99^{th}% tail latency for that threading model and load based on an exhaustive thread pool size search. Points above the dashed line are in saturation, where tail latencies are very high and meaningless. The table reports the same data as the graph with each load latency normalized to the best latency for that load, which is highlighted in blue. We omit graphs for other applications as they match the `HDSearch` trends.

We make the following observations:

SDB enables highest load. The Synchronous Dispatch Block model, with a single network thread and a large worker pool of 50 threads is the only model that sustains peak loads ($\geq$ 10K QPS). SDB is best at high loads as (1) its worker pool has enough concurrency so that leaf microservers, rather than the mid-tier, pose the bottleneck; and (2) the single network thread is sufficient to accept and dispatch the offered load. SDB outperforms SDP at high load as polling consumes CPU in fruitless poll loops. For example, at 10,000 QPS, the mid-tier microserver receives one query every 100 microseconds. In SDP, poll loops are often shorter than 100 microseconds. Hence, some poll loops that do not retrieve any requests are wasted work and may delay critical work scheduling, such as RPC response processing. Under SDB, the CPU time wasted in empty poll loops can instead be used to progress an ongoing request.

SIP has lowest latency at low load. While SDB sustains peak loads, it is latency-suboptimal at low loads. SIP offers 1.4× better low-load tail latency by avoiding up to two OS thread wakeups relative to alternative models: (1) network thread wakeups via interrupts on query arrivals, and (2) worker wakeups for RPC dispatch. Work hand-off among threads may cause OS-induced scheduling tails.

SDP is best at intermediate loads. SIP ceases being the best model when the offered load grows too large for one in-line thread to sustain. Adding more in-line polling threads causes contention in the OS and RPC reception code paths. Additional in-line blocking threads are less disruptive, but SIB never outperforms SDP. By switching to a dispatched model, a single network thread can still accept the incoming RPCs, avoiding contention and locality losses of running the `gRPC` [45] and network receive stacks across many cores. The workers add sufficient concurrency to sustain RPC and response processing. We further note that SDP tail latencies at intermediate loads are better than at low load, since there is better

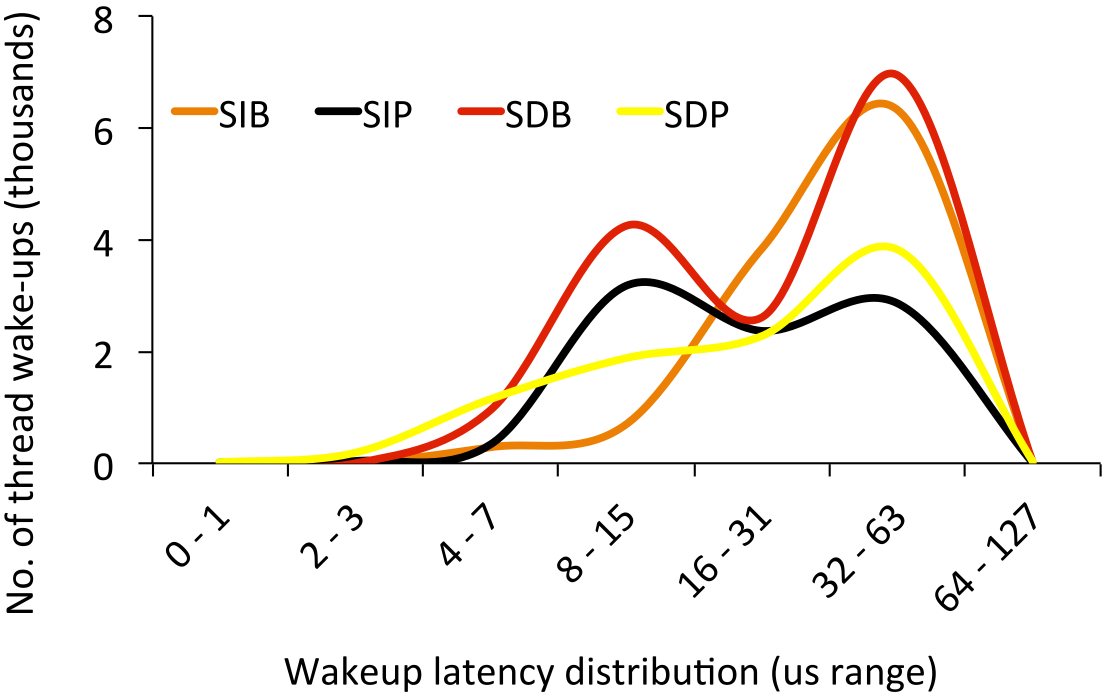


Figure 3.9: HDSearch synchronous thread wakeups at 64 QPS: Block incurs more wakeups.

temporal locality and OS and networking performance tend to improve due to batching effects in the networking stack.

OS and microarchitectural effects. We report OS thread wakeup latency distributions for HDSearch synchronous models at 64 QPS in Fig. 3.9. Although some OS thread wakeups are fast (∼5 μs), blocking models frequently incur 32-64 μs range wakeups. This data also depicts the advantage of in-line over dispatched models with respect to low-load worker wakeup costs.

Fig. 3.10 shows the relative frequency of true sharing misses (HITM), context switches, and cache misses for threading models at high load (10K QPS). These results show why SIP fails to scale as load increases. SIP needs multiple threads to sustain loads that are greater than or equal to 512 QPS. Multiple pollers contend pathologically on the network receive processing, incurring many sharing misses, context switches, and cache misses. SIB in-line threads contend less as they block, rather than poll. SDB and SDP exhibit similar contention. However, SDB outperforms SDP, since SDP incurs a mean $\sim 10\%$ higher wasted CPU

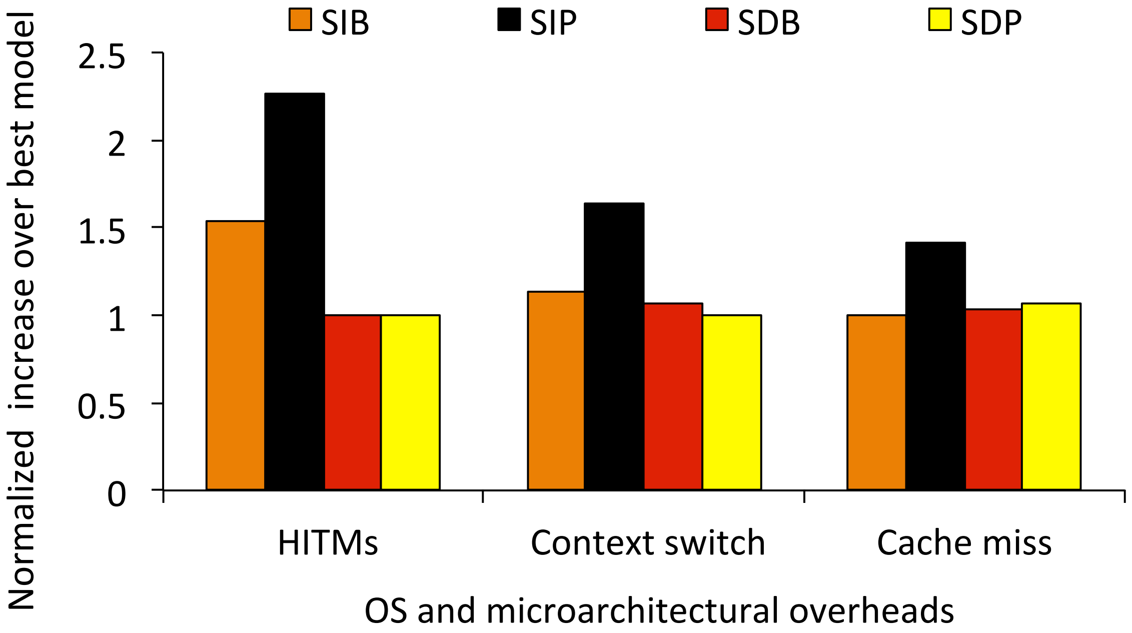
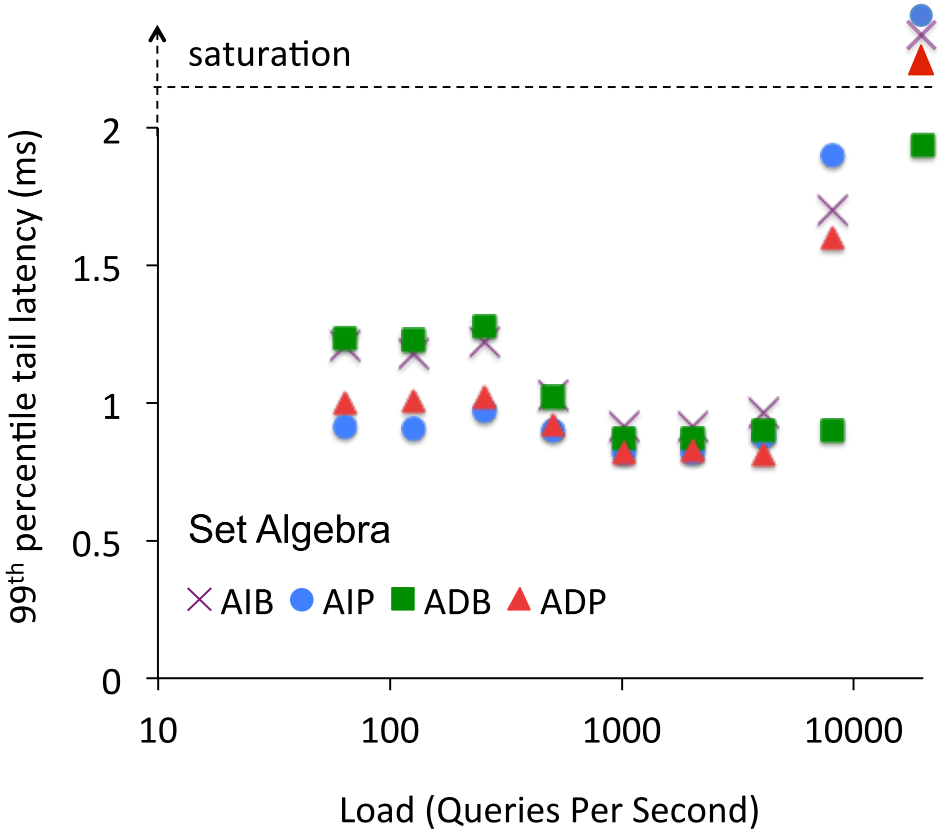


Figure 3.10: Relative frequency of synchronous contention, context switches, and cache misses at 10K QPS: SIP performs the worst.

utilization.

Additional Tests. (1) We measured *µTune* with null (empty) RPC handlers. Complete services incur higher tails than null RPCs as mid-tier and leaf computations add to tails. For null RPCs, SIP outperforms SDB by 1.57× at low loads. (2) We measured `HDSearch` on another hardware platform (Intel Xeon "Skylake" vs. "Haswell"). We notice similar trends as on our primary Haswell platform, with SIP outperforming SDB by 1.42× at low loads. (3) We note that the median latency follows a similar trend, however, with lower absolute values (e.g., `HDSearch`'s SIP outperforms SDB by 1.26× at low load). We omit figures for these tests as they match the reported `HDSearch` trends. Threading performance gaps will be wider for faster services (e.g., 200K QPS Memcached [70]) as slightest OS/network overheads become magnified [408].



QPS	64	128	256	512	1024	2048	4096	8192	20K
AIB	1.3	1.3	1.3	1.2	1.1	1.1	1.2	1.9	∞
AIP	**1**	**1**	**1**	**1**	**1**	**1**	1.1	2.1	∞
ADB	1.4	1.4	1.3	1.1	1.1	1.1	1.1	**1**	**1**
ADP	1.1	1.1	1.1	**1**	**1**	**1**	**1**	1.8	∞

Figure 3.11: Graph: Latency vs. load for `Set Algebra`'s asynchronous models. Table: The latency at each load level normalized to the best latency for that load: No threading model is always the best.

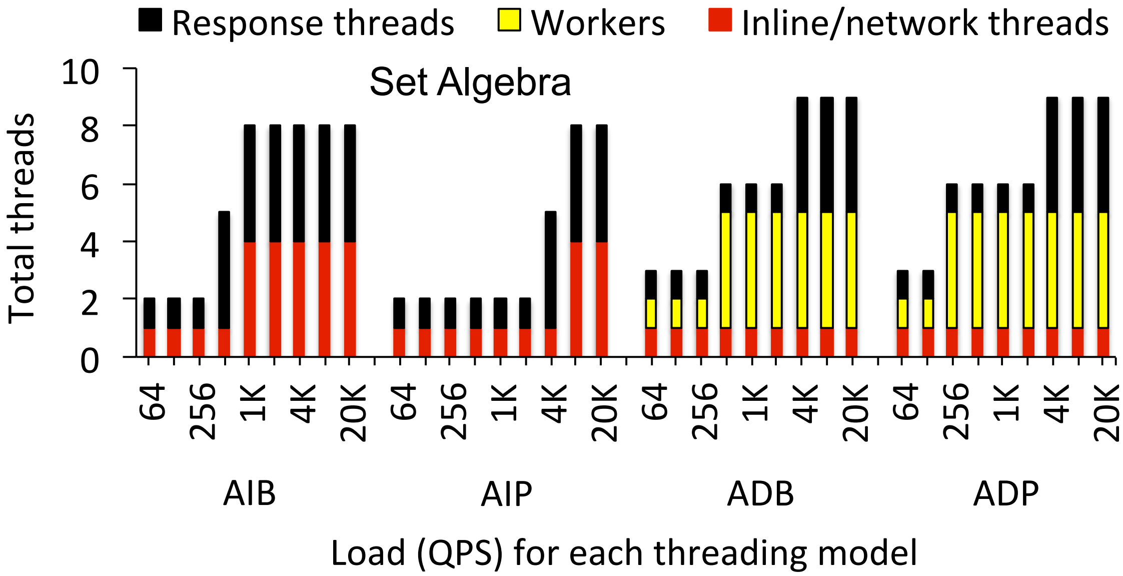


Figure 3.12: Asynchronous thread pools for best tail latency: Big thread pools content.

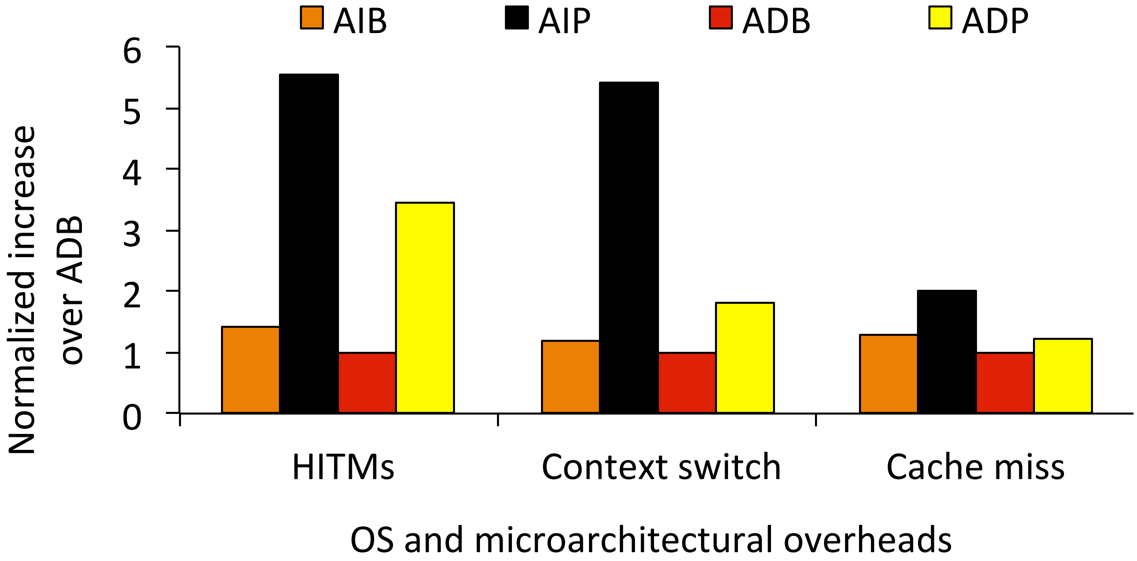


Figure 3.13: Asynchronous `Set Algebra`'s relative frequency of contention, context switches, and cache misses over the best asynchronous model at peak load: AIP performs worst.

3.6.1.3 Asynchronous models

We show results for `Set Algebra`'s asynchronous models in Fig. 3.11. As above, we omit figures for additional services as they match `Set Algebra` trends. Broadly, trends follow the synchronous models, but latencies are markedly lower. We note the following differences:

Smaller thread pool sizes. Significantly smaller (≤ 4 threads) thread pool sizes are sufficient at various loads, since asynchronous models capitalize on the available concurrency by quickly moving on to successive requests.

Fig. 3.12 shows `Set Algebra`'s asynchronous thread pool sizes that achieve the best tails for each load level. We find four threads enough to sustain high loads. Larger thread pools deteriorate latency by contending for network sockets or CPU resources. In contrast, SIB, SDB, and SDP need many threads (as many as 50) to exploit available concurrency.

AIP scales much better than SIP. AIP with just one in-line and response thread can tolerate much higher load (up to 4096 QPS) than SIP, since queuing delays engendered by both the front-end network socket and leaf node response sockets are avoided by the asynchronous design.

ADP scales worse than SDP. ADP with 4 worker and response threads copes worse than SDP at loads ≥ 8192 QPS even though it does not have a large thread pool contending for CPU (in contrast to SDP at high loads). This design fails to scale since response threads contend on the completion queue tied to leaf node response sockets.

OS and microarchitectural effects. Unlike SDP, ADP incurs more context switches, caches misses, and HITMs, due to response thread contention (Fig. 3.13).

3.6.2 Load Adaptation

We next compare *μTune*'s load adaptation against state-of-the-art baselines [242, 316, 117] for various load patterns.

3.6.2.1 Comparison to the state-of-the-art

We compare *µTune*'s run-time performance to state-of-the-art adaptation techniques [242, 316, 117]. We find that *µTune* offers better tail latency than these approaches.

Few-to-Many (FM) parallelism. FM [242] uses offline profiling to vary parallelism during a query's execution. The FM scheduler decides *when* to add parallelism for long-running queries and by *how much*, based on the dynamic load that is observed every 5 ms. In consultation with FM's authors, we decide to treat a microservice as an FM query, to create a fair performance analogy between *µTune* and FM. In our FM setup, we mimic FM's offline profiling by building an offline interval table that notes the software parallelism to add for varied loads in terms of thread pool sizes. We use the peak load-sustaining synchronous and asynchronous models (SDB and ADB). During run-time, we track the mid-tier's loads every 5 ms and suitably vary SDB/ADB's thread pool sizes. FM varies only pool sizes (vs. *µTune* also varying threading models), and we find that FM underperforms *µTune* (as we show later).

Integrating Polling and Interrupts (IPI). Langendoen et al. [316] propose a user-level communication system that adapts between poll- and interrupt-driven request reception. The system initially uses interrupts. It starts to poll when all threads are blocked. It reverts to interrupts when a blocked thread becomes active. We study this system for synchronous modes only; as its authors note [316], it does not readily apply for asynchronous modes.

To implement this technique, we keep (1) a global count of all threads and (2) a shared atomic count of *blocked* threads for the mid-tier. Before a thread becomes *blocked* (e.g., invokes a synchronous call), it increments the shared count and decrements it when it becomes active (i.e., synchronous call returns). After revising the shared count, a thread checks if the system's *active* thread count exceeds the machine's logical core count. If higher, the system blocks, otherwise, it shifts to polling. We will demonstrate that *µTune* outperforms this technique, as it considers additional model dimensions (such as inline/dispatch), as well as dynamically scales thread pools based on load.

Time window-Based Detection (TBD). Abdelzaher *et al.* [117] periodically observe request arrival times in fixed observation windows to track request rate. In our setup, we replace *µTune*'s event-based detector with this time-based detector. We pick 5 ms time-windows (like FM) to track low loads and react quickly to load spikes.

We evaluate the tail latency exhibited by *µTune* across all services and compare it to these state-of-the-art approaches [242, 316, 117] for both steady-state and transient loads. We examine *µTune*'s ability to pick a suitable threading model and size thread pools for time-varying load. We offer loads that differ from those used in training. We aim to study if *µTune* selects the best threading model, as compared to an offline exhaustive search.

3.6.2.2 Steady-state adaptation

Fig. 3.14 shows *µTune*'s ability in converging to the best threading model and thread pool size for steady-state loads. Our test steps up and down through the displayed load levels. We report the tail latency at each load averaged over five trials. The SIP1, SDP1-20, and SDB1-50 bars are optimal threading configurations for some loads. The nomenclature is the threading model followed by the pool sizes, in the form model-network-worker-response. The FM [242], Integrated Poll/Interrupt (IPI) [316], and Time-Based Detection (TBD) [117] bars are the tail latency of state-of-the-art systems. The red bars are *µTune*'s tail latency; bars are labelled with the configuration *µTune* chose.

In synchronous mode (Fig. 3.14 (top)), *µTune* first selects an SIP model with a single thread, until load grows to about 1K QPS, at which point it switches to SDP, and begins ramping up the worker thread pool size. At 8K QPS, it switches to SDB and continues growing the worker thread pool, until it reaches 50 threads, which is sufficient to meet the peak load the leaf microservice can sustain.

µTune reduces tail latency by up to 1.7× for `HDSearch`, 1.6× for `Router`, 1.4× for `Set Algebra`, and 1.5× for `Recommend` (at 20 QPS) over SDB—the static model that sustains peak loads. *µTune* reduces tail latency by a mean 1.3x over SDB across all loads and

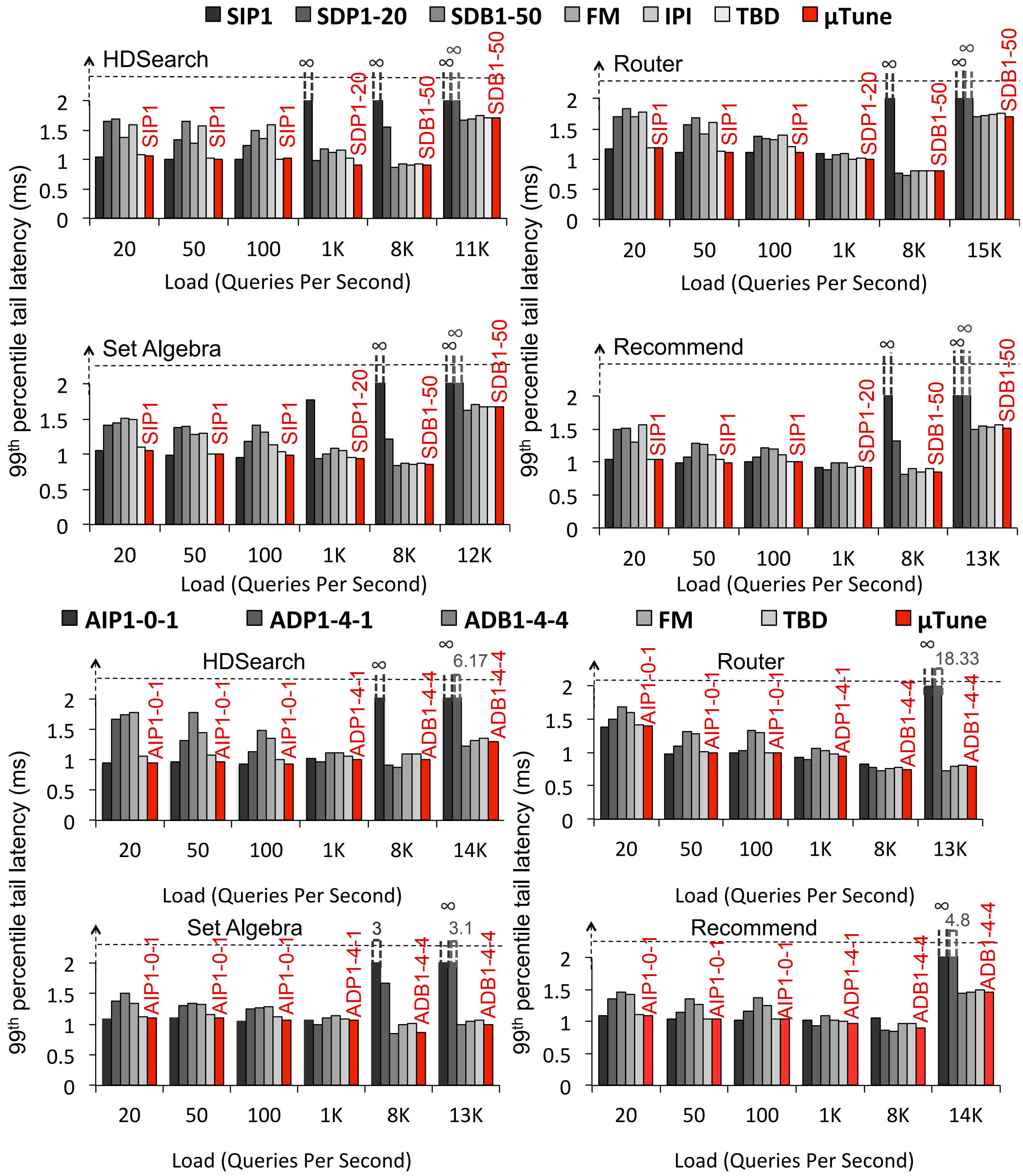


Figure 3.14: Synchronous (top) and asynchronous (bottom) steady-state adaptation.

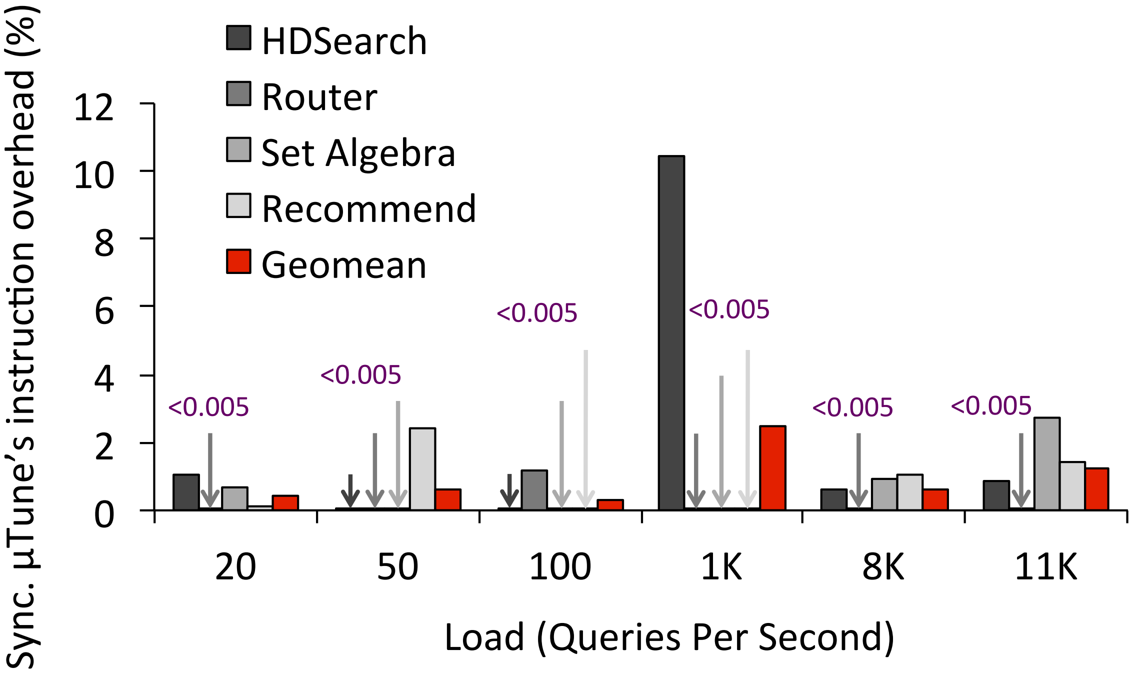


Figure 3.15: Synchronous *µTune*'s instruction overhead for steady-state loads: Less than 5% mean overhead incurred.

services. *µTune* also outperforms all state-of-the-art [242, 316, 117] techniques (except TBD) for at least one load level and never underperforms them. *µTune* outperforms FM by up to 1.3× for `HDSearch` and `Recommend`, and 1.4× for `Router` and `Set Algebra` under low loads, as FM only varies SDB's thread pool sizes and hence incurs high network poller and worker wakeups. *µTune* outperforms the IPI approach by up to 1.6× for `HDSearch`, 1.5× for `Router` and `Recommend`, and 1.4× for `Set Algebra` under low loads. At low load, IPI polls with many threads (to sustain peak load), succumbing to expensive contention. TBD does as well as *µTune* as the requests mishandled during the 5 ms monitor window fall in tails greater than the $99^{th}\%$ percentile that we monitor for 30s for each load level.

In asynchronous mode (Fig. 3.14 (bottom)), *µTune* again initially selects an in-line poll model with small-sized pools, transitioning to ADP and then ADB as load grows. Four worker and response threads suffice for all loads. We show that *µTune* outperforms static threading choices and state-of-the-art techniques by up to 1.9× for at least one load level.

Across all loads, *µTune* selects threading models and thread pool sizes that perform within 5% of the best model as determined by offline search. *μTune* incurs less than 5% mean instruction overhead over the load-specific "best" threading model, as depicted in Fig. 3.15. Hence, we find our piece-wise linear model sufficient to make good threading decisions. Note that *μTune* always prefers a single thread interacting with the front-end socket. This finding underscores the importance of maximizing locality and avoiding contention on the RPC receive path.

3.6.2.3 Load transients

Table 3.2 indicates *μTune*'s response to load transients, where the columns are a series of varied-duration load levels. The rows are the $99^{th}\%$ tail latency for the models between which *μTune* adapts in this scenario (SIP/AIP and SDB/ADB), state-of-the-art [242, 316, 117] techniques, and *μTune*. The key step in this scenario is the 8K/13K QPS load level, which lasts only 1s. We pick spikes of 8K QPS and 13K QPS for synchronous and asynchronous as these loads are SIP and AIP saturation levels, respectively.

We find that the in-line poll models accumulate a large backlog during the transient period as they saturate, and thus perform poorly even during successive low loads. FM and TBD incur high transient tail latencies as they allow requests during the 5 ms load detection window to be handled by sub-optimal threading choices. FM saturates at 8K QPS for `Recommend` since the small SDB thread pool size selected by FM at 100 QPS causes unbounded queuing during the load monitoring window. IPI works only for synchronous models and performs poorly at low loads, as its fixed-size thread pool leads to polling contention. We show that *μTune* detects the transient and transitions from SIP/AIP to SDB/ADB fast enough to avoid accumulating a backlog that affects tail latency. Once the flash crowd subsides, *μTune* transitions back to SIP/AIP, avoiding the latency penalty SDB/ADB suffer at low load.

		Synchronous				Asynchronous		
		100 QPS (0 - 30s)	8K QPS (30s - 31s)	100 QPS (31 - 61s)		100 QPS (0 - 30s)	13K QPS (30s - 31s)	100 QPS (31 - 61s)
HDSearch	SIP	0.99	>1s	>1s	AIP	0.95	>1s	>1s
	SDB	1.49	1.07	1.40	ADB	1.48	1.10	1.40
	FM	1.35	13.00	1.32	FM	1.28	4.73	1.33
	IPI	1.59	1.10	1.50	IPI	NA	NA	NA
	TBD	1.03	8.69	1.02	TBD	1.06	2.63	1.08
	μTune	1.01	1.09	0.99	*μTune*	0.98	1.13	0.96
Router	SIP	1.10	>1s	>1s	AIP	1.01	>1s	>1s
	SDB	1.31	0.83	1.36	ADB	1.35	1.13	1.31
	FM	1.33	9.40	1.40	FM	1.30	12.95	1.30
	IPI	1.4	1.10	1.38	IPI	NA	NA	NA
	TBD	1.13	4.51	1.11	TBD	1.03	6.24	1.01
	μTune	1.12	0.88	1.13	*μTune*	0.99	1.02	0.98
Set Algebra	SIP	0.95	>1s	>1s	AIP	1.04	>1s	>1s
	SDB	1.30	0.92	1.32	ADB	1.26	0.99	1.23
	FM	1.30	12.00	1.25	FM	1.28	4.14	1.27
	IPI	1.20	0.94	1.12	IPI	NA	NA	NA
	TBD	1.00	8.45	1.03	TBD	1.09	6.62	1.1
	μTune	0.97	0.92	1.03	*μTune*	1.06	1.1	1.06
Recommend	SIP	1.00	>1s	>1s	AIP	1.03	>1s	>1s
	SDB	1.26	0.96	1.22	ADB	1.37	1.30	1.32
	FM	1.23	>1s	>1s	FM	1.28	8.61	1.20
	IPI	1.13	1.02	1.13	IPI	NA	NA	NA
	TBD	1.02	4.96	1.03	TBD	1.06	6.00	1.07
	μTune	1.00	1.00	1.00	*μTune*	1.06	1.39	1.04

Table 3.2: $99^{th}\%$ tail latency (ms) for load transients.

3.7 Discussion

We briefly discuss open questions and *µTune* limitations.

Offline training. *µTune* uses offline training to build a piece-wise linear model. This phase might be removed by dynamically analyzing OS and hardware signals, such as context switches, thread wakeups, queue depths, cache misses, and lock contention, to switch threading models. Designing heuristics to switch optimally based on such run-time metrics remains an open question; our performance characterization can help guide their development.

Thread pool sizing. *µTune* tunes thread pool sizes using a piece-wise linear model. *µTune* differs from prior thread pool adaptation systems [242, 273, 302] in that it also tunes threading models. Some of these systems use more sophisticated tuning heuristics, but we did not observe opportunity for further improvement in our microservices.

CPU cost of polling. *µTune* polls at low loads to avoid thread wakeups. Polling can be costly as it wastes CPU time in fruitless poll loops. However, as most operators over-provision CPU to sustain high loads [431], when load is low, spare CPU time is typically available [248].

***µ*Tune's asynchronous framework.** Asynchronous RPC state must be maintained in thread-safe structures, which is challenging. More library/language support might simplify building asynchronous microservices with *µTune*. We leave such support to future work.

Comparison with optimized systems that use kernel-bypass, multi-queue NICs, etc. It may be interesting to study the implications of optimized systems [402, 289, 274, 154, 408, 329] that incorporate kernel-bypass, multi-queue NICs, etc., on threading models and *µTune*. Multi-queue NICs may improve polling scalability; multiple network pollers currently contend for the underlying gRPC [45] queues under *µTune*. OS-bypass may further increase the application threading model's importance; for example, it may magnify the trade-off between in-line and dispatch RPC execution, as OS-bypass eliminates latency and thread hops in the OS TCP/IP stack, shifting the break-even point to favor in-line execution for longer RPCs. However, we have limited our scope to study designs that can layer upon (unmodified) gRPC [45]; we defer studies that require extensive gRPC [45] changes (or an alternative reliable transport) to future work.

3.8 Related Work

We discuss several categories of related work.

Web server architectures. Web servers can have (a) thread-per-connection [390], (b) event-driven [394], (c) thread-per-request [263], or (d) thread-pool architectures [332]. Pai *et al.* [390] build thread-per-connection servers as multi-threaded processes. Knot [479] is a thread-per-connection non-blocking server. In contrast, *μTune* is a thread-per-request thread-pool architecture that scales better for microservices [332]. The Single Process Event-Driven (SPED) [390] architecture operates on asynchronous ready sockets. In contrast, *μTune* supports both synchronous and asynchronous I/O. The SYmmetric Multi-Process Event-Driven (SYMPED) [394] architecture runs many processes as SPED servers via context switches. The Staged Event-Driven Architecture (SEDA) [486] joins event-driven stages via queues. A stage's thread pool is driven by a resource controller. Apart from considering synchronous and asynchronous I/O as prior works [394, 486, 263, 262, 483, 219] did, *μTune* also studies a full microservice threading model taxonomy. gRPC-based systems such as Envoy [34] or Finagle [39] act as load balancers or use a single threading model.

Software techniques for tail latency: Prior works [486, 263] note that monolithic service software designs can significantly impact performance. However, micro-second–scale OS and network overheads that dominate in *μTune*'s regime do not manifest in these slower services. Some works improve web server software via software pattern re-use [424, 425], caching file systems [263], or varying parallelism [242], all of which are orthogonal to the questions we investigate. Kapoor *et al.* [289] also note that OS and network overheads impact short-running cloud services, however, their kernel bypass solution may not apply for all contexts (e.g., a shared cloud infrastructure).

Parallelization to reduce latency: Several prior works [164, 190, 280, 306, 321, 412, 457, 485, 332] reduce tails via parallelization. Others [220, 360, 143, 250] reduce medians by adaptively sharing resources. Prior works use prediction [273, 302], hardware parallelism [242], or data parallelism [225] to reduce monolithic services' tail latency. Lee *et al.* [321] use offline analysis (like *µTune*) to tune thread pools. We study microservice threading, and vary threading models altogether. However, we build on prior works' thread pool sizing insights.

Hardware mechanisms for tail latency: Several other prior works reduce leaf service tail latency via better co-location [327], voltage boosting [261, 292], or applying heterogeneity in multi-cores [243]. However, they do not study microservice tail latency effects engendered by software threading, OS, or network.

3.9 Long-Term Impact Potential

This work identifies new insights in the age-old research area of software threading models that resulted in redesigning threading models for hyperscale microservices [449]. To quote an anonymous expert reviewer for the USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2018, *"When I started reading the paper, I thought that it was yet another threading model paper, which it was; however, the systematic nature of the paper and new measurement comparisons applied to an important emerging service paradigm made it very interesting."* We discuss impact in terms of (1) long-term impact in industry and academia and (2) potential for follow-on research.

***µTune* can improve data center performance/watt.** Microservices are an important data center software trend that have profound implications on the future design of system features that currently induce sub-ms–scale overheads. Current sub-ms system overheads arise from threading designs that affect OS and I/O interactions, accesses to emerging storage-class memories, rack-scale memory disaggregation, 100+ Gbps network communication, accelerator/GPU micro-offloads, etc. We study a critical source of sub-ms system overhead—threading model design. Curtailing microservice tail latency by using μ*Tune* to enable efficient threading can result in significant data center performance/watt improvements. These benefits improve user experience, energy, and cost. To quote an anonymous expert OSDI 2018 reviewer, *"μTune abstracts all threading models under simple APIs, is practical, and shows tail latency benefits across the load spectrum", "The core idea of μTune is simple, but, very useful."* For this reason, my conversations with researchers at several hyperscale enterprises (e.g., Microsoft and Facebook) revealed that μTune could find an immediate application in their data centers.

Threading design space clarity. We show that threading trade-offs are not obvious: no single threading model achieves the best latency across all loads, and even expert developers have difficulty making good choices. Our threading taxonomy provides clarity to the microservice threading design space. Expert OSDI reviewers' comments include *"Great tutorial on RPC threading options", "The taxonomy brings clarity to different threading choices", "I feel like this is a very clear exploration of the different choices and when they improve and don't improve performance", "The work brings clarity to the threading design space"*. We enable expert and novice developers alike to use our taxonomy framework as well as latency trade-offs revealed from our taxonomy characterization to guide their microservice threading designs.

Applying μ*Tune* in a broad range of server tiers. One OSDI reviewer felt that μ*Tune* could be broadly used in all server tiers instead of solely the mid-tier. We conversed with researchers at several hyperscale enterprises about μ*Tune*'s applicability in other server tiers. We discovered that several hyperscale enterprises' leaf microservices have a sub-ms-scale latency, and threading design decisions such as in-line vs. dispatch or poll vs. block can significantly impact the leaf tier.

Conversations with these hyperscale enterprises' edge computing teams revealed that *µTune* can have even broader implications than merely the middle/leaf tiers. Systems researchers who work on edge data centers perpetually face threading challenges due to the highly distributed edge systems' structure. These researchers expressed interest in a threading model auto-tuner that is analogous to *µTune*, but applies to edge servers. *µTune* (or its variants) can therefore help reduce tail latency in several tiers of a hyperscale distributed system.

Future Research. Our threading taxonomy characterization and *µTune* can influence a wide spectrum of future research.

(1) *µTune*'s offline training phase can be removed by dynamically analyzing microarchitectural and OS signals, such as context switches, thread wakeups, queue depths, cache misses, and lock contention, to switch threading models and scale thread pools. Designing heuristics to switch optimally based on such run-time metrics remains an open question; our performance characterization can help guide their development[4].

(2) It will be interesting to study optimized systems that use kernel-bypass, multi-queue NICs, etc., on threading models and *µTune*. Multi-queue NICs might improve polling scalability. OS-bypass might further increase the application threading model's importance; for example, it might magnify the trade-off between in-line and dispatch RPC execution, as OS-bypass eliminates latency and thread hops in the OS TCP/IP stack, shifting the break-even point to favor in-line execution for longer RPCs.

(3) We show that thread wakeups can significantly deteriorate tail latency. As detailed in Chapter II, thread wakeup breakdowns indicate that OS scheduler decisions can be inefficient—thread wakeup delays primarily arise from an increase in the time from when a thread enters the *active* or *runnable* state to when it starts *running* on a CPU. Superior OS scheduling policies for microservices can diminish tail latency spikes that arise due to thread wakeup delays. Based on these insights, in Chapter V, we also design a more efficient event notification paradigm.

[4] Fun fact: I was invited to intern at Microsoft Research and use such run-time heuristics to integrate *µTune* into Microsoft Azure.

(4) μ*Tune* minimizes *tail latency* across wide-ranging load. μ*Tune* can be extended to optimize a myriad of data center-critical performance metrics. For example, μ*Tune*'s tuning mechanism can be altered to pick the Pareto optimal point in relation to tail latency, CPU cost/query, and energy efficiency.

(5) We find that the best threading model depends critically on the offered load. However, threading latency trade-offs can depend on other parameters that fluctuate at runtime such as co-runners, CPU power states, and garbage collection activities. μ*Tune* can be extended to include such variables.

3.10 Follow-On Research

In follow-on work[5], my co-authors and I masked the OS/network-induced microsecond-scale stalls identified in my threading analysis (detailed in Section 3.6). Specifically, due to these OS/network-induced microsecond-scale stalls, we are entering an era of "killer microseconds" in hyperscale web services. Killer microseconds refer to microsecond-scale "holes" in CPU schedules caused by stalls to access fast I/O devices (e.g., network devices) or brief idle times between requests in high throughput microservices. Whereas modern computing platforms can efficiently hide nanosecond-scale and millisecond-scale stalls through micro-architectural techniques and OS context switching respectively, they lack efficient support to hide the latency of microsecond-scale stalls. Simultaneous Multithreading (SMT) is an efficient way to improve core utilization and increase server performance density. Unfortunately, scaling SMT to provision enough hardware threads to hide frequent microsecond-scale stalls is prohibitive and SMT co-location can often drastically increase the tail latency of microservices.

We presented *Duplexity* [368, 369], a heterogeneous server architecture that employs aggressive multithreading to hide the latency of killer microseconds, without sacrificing

[5] This follow-on research was performed in collaboration with fellow graduate student Seyedamirhossein Mirhosseininiri (who is the lead author) and other contributors [368]. Concepts summarized in Section 3.10 are detailed in Mirhosseininiri's dissertation in Section 2 [370].

the Quality-of-Service (QoS) of latency-critical microservices. *Duplexity* provisions dyads (pairs) of two kinds of cores: director-cores[6], which each primarily executes a single latency-critical director-thread, and lender-cores, which multiplex latency-insensitive throughput threads. When the director-thread stalls, the director-core borrows filler-threads from the lender-core, filling microsecond-scale utilization holes of the microservice. We introduced critical mechanisms, including separate memory paths for the director-thread and filler-threads, to enable director-cores to borrow filler-threads while protecting director-threads' state from disruption. *Duplexity* facilitates fast director-thread restart when microsecond stalls resolve and minimizes the microservice's QoS violation. We demonstrated that *Duplexity* achieves 1.9× higher core utilization and 2.7× lower iso-throughput 99^{th}-percentile tail latency over an SMT-based server design, on average.

3.11 Chapter Summary

We summarize our contributions as follows:

- **A taxonomy of threading models.** We revisited the age-old research area of software threading models in the context of today's hyperscale microservices by systematically laying out a taxonomy of threading models for the microservice regime and analyzing them to identify new threading implications that manifest for hyperscale microservices.

- **A detailed performance study of threading model implications.** We systematically characterized our taxonomy of threading models to make the important and non-obvious observation that no single threading model is best across all hyperscale load conditions. We enable expert and novice developers alike to use our taxonomy framework as well as latency trade-offs revealed from our taxonomy characterization

[6] In the research paper that describes Duplexity [368], we call these cores "master-cores". In this dissertation, I have replaced the term "master-core" with the term "director-core" to promote the use of more inclusive language in technical writing.

to guide their microservice threading designs.

- ***μTune's*** **framework.** *μTune's* framework for developing microservices abstracts microservice threading design from application code and supports a wide variety of threading models. By abstracting complex threading details, *μTune* empowers small software developer teams to develop performance-efficient microservices that meet latency goals without spending enormous effort on optimizing complex threading details.

- ***μTune's*** **load adaptation system.** Driven by our observation that no single threading model is best across all hyperscale load conditions, we demonstrated how threading models must be redesigned for microservices by presenting *μTune's* run-time load adaptation system that intelligently tunes threading models and thread pool sizes under varying loads. We also presented a detailed performance study of web services' key tier (i.e., the mid-tier microserver) built with *μTune*.

Our study of OS/network performance overheads in Chapter II revealed that threading interactions with the underlying OS and network stacks can impact the tail latency of microservices more significantly than their monolithic counterparts. Threading-induced overheads that microservices face are due to today's hardware reality, where network and I/O devices have sped up while CPU performance scaling has nearly stopped [196]. Hence, we recognized the critical need to analyze threading effects for the microservice regime.

We investigated how threading design critically impacts microservice tail latency by developing a *taxonomy of threading models*—a structured understanding of the implications of how microservices manage concurrency and interact with RPC interfaces under wide-ranging loads. We used our taxonomy of threading models to systematically characterize threading-induced performance behaviors under wide-ranging load conditions.

We made the important observation that no single threading model is best across all load conditions. Driven by this observation, we developed *μTune*, a system that has two features:

(1) a novel framework that abstracts threading model implementation from application code, and (2) an automatic load adaptation system that curtails microservice tail latency by exploiting inherent latency trade-offs revealed in our taxonomy to transition among threading models at system runtime. We studied *μTune* in the context of *μSuite*'s web services to demonstrate up to 1.9× microservice tail latency reduction over static threading choices and state-of-the-art adaptation techniques.

We also described follow-on work, *Duplexity* [368], on hiding OS/network-induced microsecond-scale stalls identified in our threading analysis (detailed in Section 3.6). *Duplexity* is a heterogeneous server architecture that schedules latency-insensitive jobs when a microservice faces microsecond-scale stalls, improving data center performance and energy efficiency.

CHAPTER IV

Optimizing Commodity Server Architectures for Microservice Diversity at Hyperscale

The increasing user base and feature portfolio of web applications is driving precipitous growth in the diversity and complexity of the back-end services comprising them [285]. As described in Chapter I, there is a growing trend towards microservice implementation models [17, 7, 94, 477, 106], wherein a complex web application is decomposed into numerous, specialized, distributed microservices [286, 378, 448, 443]. This deployment model enables application components' independent scalability by ramping the number of physical servers/cores dedicated to each in response to diurnal and long-term load trends [477].

At global user population scale, important microservices can grow to account for an enormous installed base of physical hardware. Across Facebook's global server fleet, seven key microservices in four web service domains run on hundreds of thousands of servers, i.e., at hyperscale, and occupy a significant portion of the compute-optimized installed base. These microservices' importance begs the question: do our existing commodity server platforms serve them well? Are there common bottlenecks across microservices that we might address when selecting a future commodity server CPU architecture?

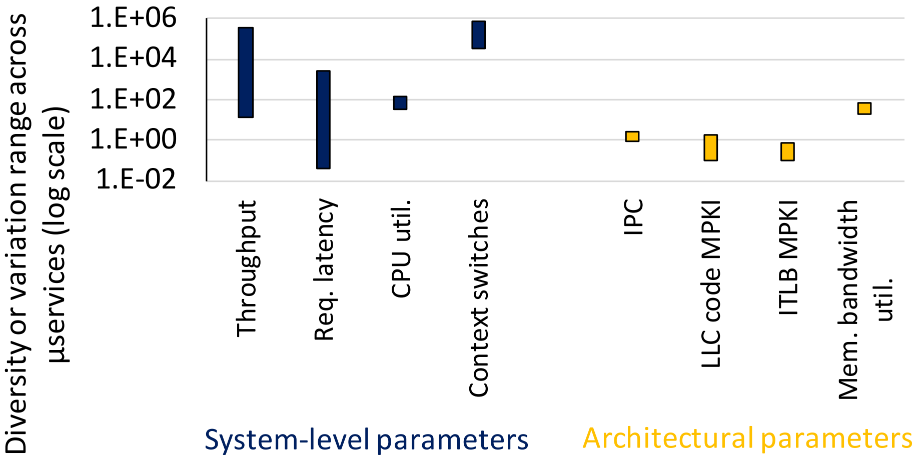


Figure 4.1: Variation in system-level & architectural traits across microservices: Facebook's microservices face extremely diverse bottlenecks.

As a part of this dissertation's hardware contributions, we[1] undertake comprehensive system-level and architectural characterizations of these microservices on Facebook production systems serving live traffic. We find that application functionality disaggregation across microservices has yielded enormous diversity in system and CPU architectural requirements, as shown in Fig. 4.1. For example, caching microservices [171] require intensive I/O and microsecond-scale response latency and frequent OS context switches can comprise 18% of CPU time. In contrast, a `Feed` [506] microservice computes for seconds per request with minimal OS interaction. Facebook's `Web` [388] microservice exhibits massive instruction footprints, leading to astonishing instruction cache and I-TLB misses and branch mispredictions, while other microservices exhibit much smaller instruction footprints. Some microservices depend heavily on floating-point performance while others have no floating-point instructions.

Such diversity might suggest a strategy to specialize CPU architectures to suit each microservice's distinct needs. Optimizing one or more of such production microservices to achieve even single-digit percent speedups can yield immense performance-per-watt

[1] Some of the work in this chapter was performed in collaboration with a researcher at Facebook, Abhishek Dhanotia, and my Ph.D. advisor, Thomas. F. Wenisch [446]. Therefore, I use the "we" pronoun in this chapter to acknowledge their involvement in this work. The lightning video of the related ISCA paper [446] is available at: `https://www.youtube.com/watch?v=m_SAiOQwu4w`

benefits and save millions of dollars [446, 298]. Indeed, we report observations that might inform future hardware designs. However, hyperscale enterprises have strong economic incentives to limit hardware platforms' diversity to (1) maintain fungibility of hardware resources, (2) preserve procurement advantages that arise from economies of scale, and (3) limit the overhead of qualifying/testing myriad hardware platforms. As such, there is an immediate need for strategies that enable a limited set of server CPU architectures (often called "SKUs," short for "Stock Keeping Units") to provide performance and energy efficiency over microservices with diverse characteristics.

Rather than diversify the hardware portfolio, we motivate the need for "soft SKUs," a strategy wherein we exploit coarse-grain (e.g., boot time) OS and hardware configuration knobs to tune limited hardware SKUs to better support their presently assigned microservice. Unlike data centers that co-locate services via virtualization, Facebook's microservices run on dedicated bare metal servers, allowing us to easily create microservice-specific soft SKUs [446]. As microservice allocation needs vary, servers can be redeployed to different soft SKUs through reconfiguration and/or reboot. Our OS and CPUs provide several specialization knobs; in this study, we focus on seven: (1) core frequency, (2) uncore frequency, (3) active core count, (4) code vs. data prioritization in the last-level cache ways, (5) hardware prefetcher configuration, (6) use of transparent huge pages, and (7) use of static huge pages.

Identifying the best microservice-specific soft-SKU configuration is challenging: the design space is large, service code evolves quickly, synthetic load tests do not necessarily capture production behavior, and the effects of tuning a particular knob are often small (a few percent performance change). To this end, we develop *μSKU*—a design tool that automates search within the seven-knob soft-SKU design space using A/B testing in production systems on live traffic. *μSKU* automatically varies soft-SKU configuration while collecting numerous fine-grain performance measurements to obtain sufficient statistical confidence to detect even small performance improvements. We evaluate a prototype of *μSKU* and demonstrate that the soft SKUs it designs outperform stock and expert-tuned production server configurations by up to 7.2% and 4.5% respectively, with no additional hardware requirement. Even such single-digit performance gains yield immense performance-per-watt benefits, saving millions of dollars and meaningfully reducing the global carbon footprint [446, 298, 299, 297].

The rest of this chapter is organized as follows: We describe and measure Facebook's seven production microservices' performance traits in Section 4.1. We motivate the need for Soft SKUs in Section 4.2. We describe *μSKU*'s design in Section 4.3 and we discuss the methodology used to evaluate *μSKU* in Section 4.4. We evaluate *μSKU* in Section 4.5, discuss limitations in Section 4.6, and compare against related work in Section 4.7. We also describe follow-on research in Section 4.9 and long-term impact potential in Section 4.8, before concluding in Section 4.10.

4.1 Understanding Microservice Performance

We identify software and hardware bottlenecks faced by Facebook's key production microservices to see if they share common bottlenecks that might be addressed in future server CPU architectures. In this section, we (1) describe each microservice, (2) explain our characterization methodology, (3) discuss system-level characteristics to provide insights into how each microservice is operated, (4) report on the architectural characteristics and bottlenecks faced by each microservice, and (5) summarize our characterization's most important conclusions. A key theme that emerges throughout this characterization is *diversity*; the seven microservices differ markedly in their performance constraints' time-scale, instruction mix, cache behavior, CPU utilization, bandwidth requirements, and pipeline bottlenecks. Unfortunately, this diversity calls for sometimes conflicting optimization choices, motivating our pursuit of "soft SKUs" (Section 4.2) rather than custom hardware for each microservice.

4.1.1 The Production Microservices

We characterize seven microservices in four diverse service domains running on Facebook's compute-optimized data center fleet. The workloads with longer work-per-request (e.g., Feed2, Ads1) might be called "services" by some readers; we use "microservice," since none of these systems is entirely stand-alone. We characterize on production systems serving live traffic. We first detail each microservice's functionality.

Web. Web implements the HipHop Virtual Machine, a Just-In-Time (JIT) compilation and runtime system for PHP and Hack [388, 493, 118], to serve web requests originating from end-users. Web employs request-level parallelism: an incoming request is assigned to one of a fixed pool of PHP worker threads, which services the request until completion. If all workers are busy, arriving requests are enqueued. Web makes frequent requests to other microservices, and the corresponding worker thread blocks waiting on the responses.

Feed1 and Feed2. Feed1 and Feed2 are key microservices in Facebook's News Feed service. Feed2 aggregates various leaf microservices' responses into discrete "stories." These stories are then characterized into dense feature vectors by feature extractors and learned models [506, 414, 142, 248]. The feature vectors are then sent to Feed1, which calculates and returns a predicted user relevance vector. Stories are then ranked and selected for display based on the relevance vectors.

Ads1 and Ads2. Ads1 and Ads2 maintain user-specific and ad-specific data, respectively [249]. When Ads1 receives an ad request, it extracts user data from the request and sends targeting information to Ads2. Ads2 maintains a sorted ad list, which it traverses to return ads meeting the targeting criteria to Ads1. Ads1 then ranks the returned ads.

Cache1 and Cache2. Cache is a large distributed-memory object caching service (like, e.g., [171, 474, 223, 88]) that reduces throughput requirements of various backing stores. Cache1 and Cache2 correspond to two tiers within each geographic region for this service. Client microservices contact the Cache2 tier. If a request misses in Cache2, it is forwarded to the Cache1 tier. Cache1 misses are then sent to an underlying database cluster in that

Table 4.1: Skylake18, Skylake20, Broadwell16's key attributes.

CPU features	Skylake18	Skylake20	Broadwell16
Microarchitecture	Intel Skylake	Intel Skylake	Intel Broadwell
Number of sockets	1	2	1
Cores/socket	18	20	16
SMT	2	2	2
Cache block size	64 B	64 B	64 B
L1-I$ (per core)	32 KiB	32 KiB	32 KiB
L1-D$ (per core)	32 KiB	32 KiB	32 KiB
Private L2$ (per core)	1 MiB	1 MiB	256 KiB
Shared LLC (per socket)	24.75 MiB	27 MiB	24 MiB

region.

4.1.2 Characterization Approach

We characterize the seven microservices by profiling each in production while serving real-world user queries. We next describe the characterization methodology.

Hardware platforms. We perform our characterization on 18- and 20-core Intel Skylake processor platforms [212], Skylake18 and Skylake20. Characteristics of each are summarized in Table 4.1. Web, Feed1, Feed2, Ads1, and Cache2 run on Skylake18. Ads2 and Cache1 are deployed on Skylake20. Both platforms support Intel Resource Director Technology (RDT) [103]. RDT facilitates tunable Last-Level Cache (LLC) size configurations using Cache Allocation Technology (CAT) [267] and allows prioritizing code vs. data in the LLC ways using Code Data Prioritization (CDP) [25].

Experimental setup. We measure each microservice in Facebook's production environment's default deployment—stand-alone with no co-runners on bare metal hardware. Therefore, there are no cross-service contention or interference effects in our data. We measure each system at peak load to stress performance bottlenecks and characterize the system's maximum throughput capabilities. Facebook's production microservice codebases evolve rapidly; we repeat experiments across updates to ensure that results are stable.

We collect most system-level performance data using an internal tool called Operational

Data Store (ODS) [168, 397, 123]. ODS enables retrieval, processing, and visualization of sampling data collected from all machines in the data center. ODS provides functionality similar to Google-Wide-Profiling [418].

To analyze microservices' interactions with the underlying hardware, we use myriad processor performance counters. We collect data with Intel's EMON [33]—a performance monitoring and profiling tool that time multiplexes sampling of a vast number of processor-specific hardware performance counters with minimal error. For each experiment, we use this tool to collect tens of thousands of hardware performance events. We report 95% confidence intervals on mean results.

We contrast our measurements with some CloudSuite [221], SPEC CPU2006 [253], SPEC CPU2017 [331], and Google services [140, 285] where possible. We measured SPEC CPU2006 performance on `Skylake20`. We reproduce selected data from published reports on SPEC CPU2017 [331], CloudSuite [221], and Google's services [140, 285] measured on Haswell, Westmere, and Haswell, respectively. These results are not directly comparable with our measurements as they are measured on different hardware. Nevertheless, they provide context for the greater bottleneck diversity we observe in Facebook's microservices relative to commonly studied benchmark suites.

We present our characterization in two parts. We first discuss system-level characteristics observed over the entire fleet. We then present performance-counter measurements and their implications on architectural bottlenecks.

4.1.3 System-Level Characterization

We first present key system-level metrics, such as request latency, achieved throughput, and path length (instructions per query), to provide insight into how the microservices behave and how these traits may impact architectural bottlenecks. Throughout, we call attention to key axes of diversity.

Table 4.2: Average request throughput, request latency, & path length across microservices: We observe great diversity across services.

μservice	Throughput (QPS)	Request latency	Instructions/query
Web	O (100)	O (ms)	O (10^6)
Feed1	O (1000)	O (ms)	O (10^9)
Feed2	O (10)	O (s)	O (10^9)
Ads1	O (10)	O (ms)	O (10^9)
Ads2	O (100)	O (ms)	O (10^9)
Cache1	O (100K)	O (μs)	O (10^3)
Cache2	O (100K)	O (μs)	O (10^3)

4.1.3.1 Request throughput, request latency, and path length

We report approximate peak-load throughput, average request latency, and path length (instructions per query) in Table 4.2. The amount of work per query varies by six orders of magnitude across the microservices, resulting in throughputs ranging from tens of Queries Per Second (QPS) to 100,000s of QPS with average request latencies ranging from tens of microseconds to single-digit seconds.

Microservices' differing time scales imply that per-query overheads that may pose major bottlenecks for some microservices are negligible for others. For example, microsecond-scale overheads that arise from accesses to Flash [120], emerging memory technologies like 3D XPoint by Intel and Micron [51, 235, 308], or 40-100 Gb/s Infiniband and Ethernet network interactions [476] can significantly degrade the request latency of microsecond-scale microservices [180, 145, 368, 369] like `Cache1` or `Cache2`. However, such microsecond-scale overheads have negligible impact on the request latency of seconds-scale microservices like `Feed2`. The request latency diversity motivates our choice to include several microservices in our detailed performance-counter investigation.

4.1.3.2 Request latency breakdown

We next characterize request latency in greater detail to determine the relative contribution of computation and queuing/stalls on an average request's end-to-end latency. We report

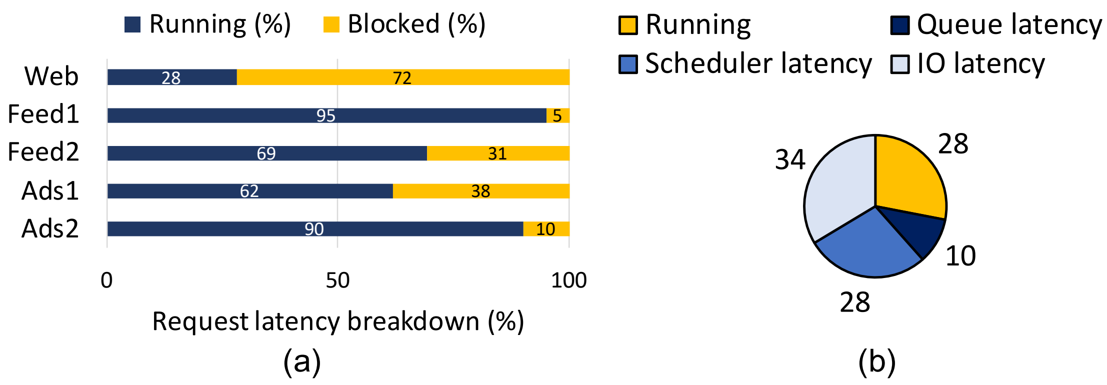


Figure 4.2: (a) A single request's latency breakdown for each μservice: Few μservices block for a long time, (b) Web's request latency breakdown: Thread over-subscription causes scheduling delays.

the average fraction of time a request is "running" (executing instructions) vs. "blocked" (stalled, e.g., on I/O) in Fig. 4.2 (a). We omit `Cache1` and `Cache2` from this measurement since their queries follow concurrent execution paths and time cannot easily be apportioned as "running" or "blocked".

`Feed1` and `Ads2` are almost entirely compute-bound throughout a request's life as they are leaves and do not block on requests to other microservices in the common case. They will benefit directly from architectural features that enhance instruction throughput. In contrast, `Web`, `Feed2`, and `Ads1` emit requests to other microservices and hence their queries spend considerable time blocked. These can benefit from architectural/OS features that support greater concurrency [449, 347], fast thread switching, and better I/O performance [450, 447].

We further break down `Web`'s "blocked" component in Fig. 4.2 (b) into queuing latency (while a query awaits a worker thread's availability), scheduler latency (where a worker is ready but not running), and I/O latency (where a query is blocked on a request to another microservice). Although `Web`'s scheduler delays are surprisingly high, these delays are not due to inefficient system design, and are instead triggered by thread over-subscription. To improve `Web`'s throughput, load balancing schemes continue spawning worker threads until adding another worker begins degrading throughput.

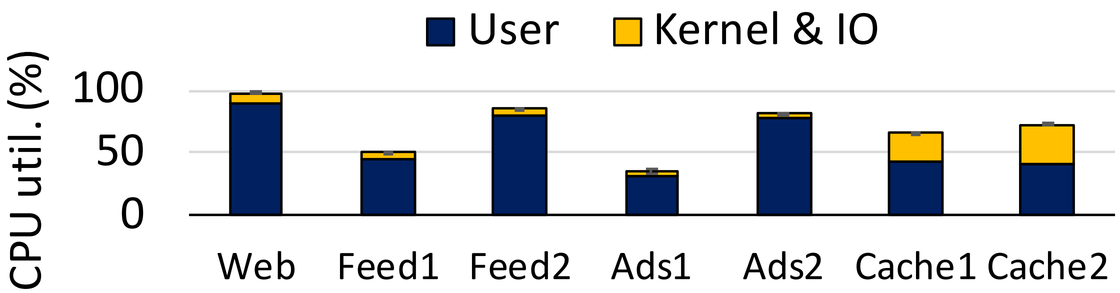


Figure 4.3: Max. achievable CPU utilization in user- and kernel-mode across μservices: Utilization can be low to avoid QoS violations.

4.1.3.3 CPU utilization at peak load

The microservices also vary in their CPU utilization profile. Fig. 4.3 shows the CPU utilization and its user- and kernel-mode breakdown when each microservice is operated at the maximum load it can sustain without violating Quality of Service (QoS) constraints. We make two observations: (1) CPU resources are not always fully utilized. (2) Most microservices exhibit a relatively small fraction of kernel/IO wait utilization. Each microservice faces latency, quality, and reliability constraints, which impose QoS requirements that in turn impose constraints on how high CPU utilization may rise before a constraint is violated. Our load balancers modulate load to ensure constraints are met. More specifically, `Cache1`, `Cache2`, `Feed1`, `Feed2`, `Ads1`, and `Ads2` under-utilize the CPU due to strict latency constraints enforced to maintain user experience. These services might benefit from tail latency optimizations, which might allow them to operate at higher CPU utilization. `Cache1` and `Cache2` exhibit higher kernel-mode utilization due to frequent context switches, which we inspect next.

4.1.3.4 Context switch penalty

We report the fraction of a CPU-second each microservice spends context switching in Fig. 4.4. We estimate context switch penalty by first aggregating non-voluntary and voluntary context switch counts reported by Linux's `time` utility. We then estimate upper and lower context switch penalty bounds using switching latencies reported by prior works [470, 325].

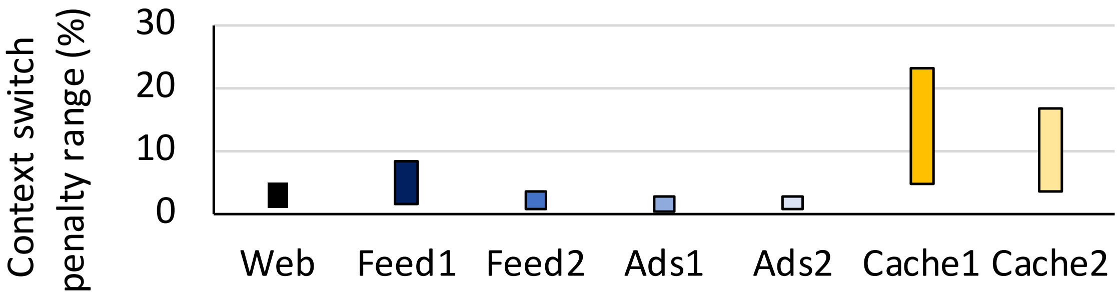


Figure 4.4: Fraction of a second spent context switching (range): Cache1 & Cache2 can benefit from context switch optimizations.

`Cache1` and `Cache2` incur context switches far more frequently than other microservices, and may spend as much as 18% of CPU time in context switching. Several context switches are non-voluntary context switches that arise due to a large number of I/O event notifications, resulting in a high kernel/IO wait utilization (as shown in Fig. 4.3). These frequent context switches also lead to worse cache locality, as we will show in our architectural characterization. Software/hardware optimizations [210, 274, 154, 153, 215, 318, 320, 165, 451] that reduce context switch latency or counts might considerably improve `Cache` performance. (In Chapter V, we use existing hardware mechanisms better to reduce this I/O notification latency.)

4.1.3.5 Instruction mix

We report Facebook microservices' instruction mix and contrast with SPEC CPU2006 benchmarks in Fig. 4.5. Instruction mix varies substantially across Facebook's microservices, especially with respect to store-intensity and the presence/absence of floating-point operations. The microservices that include ranking models that operate on real-valued feature vectors, `Ads1`, `Ads2`, `Feed1`, and `Feed2`, all include floating-point operations, and `Feed1` is dominated by them. These microservices can likely benefit from optimizations for dense computation, such as SIMD instructions.

Prior work has reported that key-value stores, like `Cache1` and `Cache2`, are typically memory intensive [171]. However, we note that `Cache` requires substantial arithmetic and

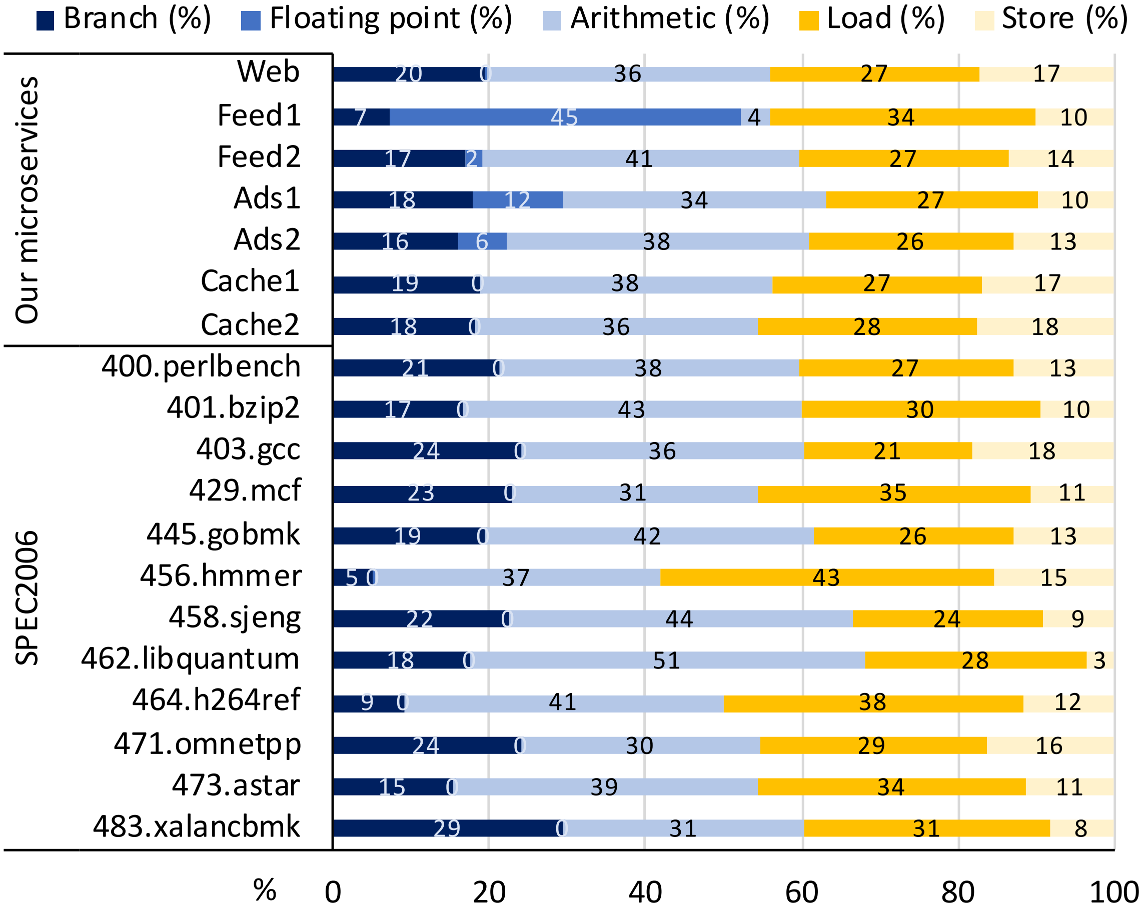


Figure 4.5: Instruction type breakdown across microservices: Instruction mix ratios vary substantially across microservices.

control flow instructions for parsing requests and marshalling or unmarshalling data; their load-store intensity does not differ from other services as much as the literature might suggest.

4.1.4 Architectural Characterization

We next turn to performance-counter-based analysis of the architectural bottlenecks of Facebook's microservice suite, and examine opportunities it reveals for future hardware SKU design.

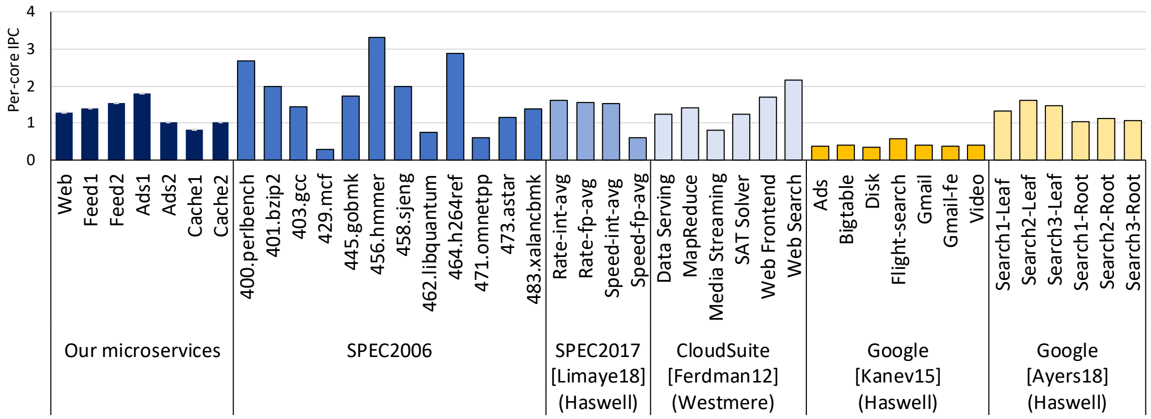


Figure 4.6: Per-core IPC across Facebook's microservices and prior work (IPC measured on other platforms): Facebook's microservices have a high IPC diversity.

4.1.4.1 IPC and stall causes

We report each microservice's overall Instructions Per Cycle (IPC) in Fig. 4.6. We contrast our results with IPCs for commonly studied benchmark suites [331, 221] and published results for comparable Google services [285, 140]. Prior works' IPCs are measured on other platforms as shown in Fig. 4.6; although absolute IPCs may not be directly comparable, it is nevertheless useful to compare variability and spreads.

None of Facebook's microservices use more than half of the theoretical execution bandwidth of a Skylake CPU (theoretical peak IPC of 5.0), and `Cache1` uses only 20%. As such, simultaneous multithreading is effective for these services and is enabled in our platforms. Relative to alternative benchmarks, Facebook's microservices exhibit (1) a greater IPC diversity than Google's services [285] and (2) a lower IPC than most widely-studied SPEC CPU2006 benchmarks. Given Facebook production workloads' larger codebase, larger working set, and more varied memory access patterns, we do not find this lower typical IPC surprising. When accounting for Skylake's enhanced performance over Haswell, we find the range of IPC values we report to be comparable to the Google services [140].

We provide insight into the root causes of relatively low IPC using the Top-down Microarchitecture Analysis Method (TMAM) [495] to categorize processor pipelines' execution

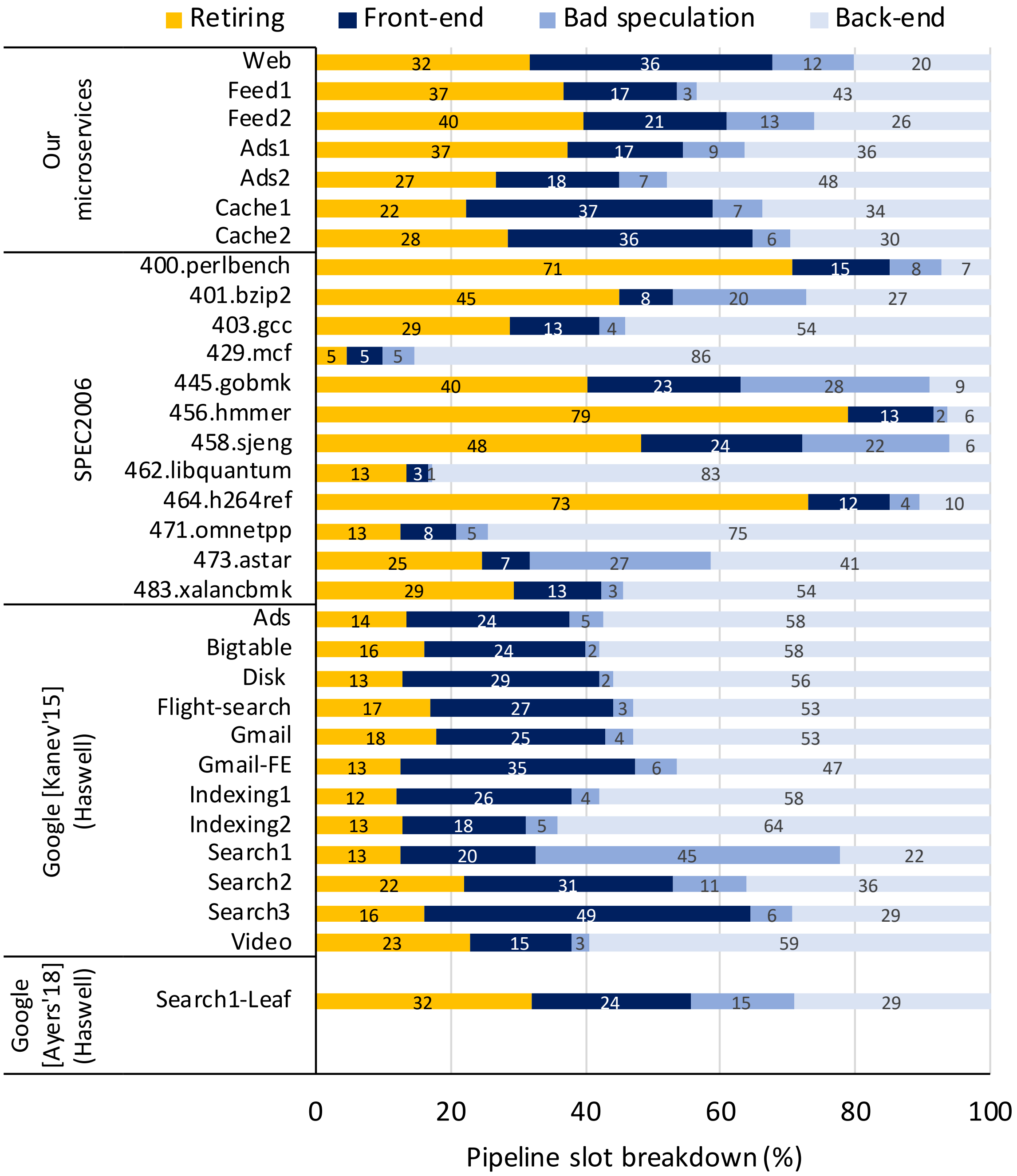


Figure 4.7: Top-down bottleneck breakdown: Several of Facebook's microservices face high front-end stalls.

stalls, as reported in Fig. 4.7. TMAM exposes architectural bottlenecks despite the many latency-masking optimizations of modern out-of-order processors. The methodology reports bottlenecks in terms of “instruction slots”—the fraction of the peak retirement bandwidth that is lost due to stalls each cycle. Slots are categorized as: *front-end* stalls due to instruction fetch misses, *back-end* stalls due to pipeline dependencies and load misses, *bad speculation* due to recovery from branch mispredictions, and *retiring* of useful work.

As suggested by the IPC results, Facebook’s microservices retire instructions in only 22%-40% of possible retirement slots. However, the nature of the stalls in Facebook’s applications varies substantially across microservices and differs markedly from the other suites. We make several observations.

First, Facebook’s microservices tend to have greater front-end stalls than SPEC workloads. In particular, `Web`, `Cache1`, and `Cache2` lose $\sim$37% of retirement slots due to front-end stalls; only Google’s `Gmail-FE` and `search` exhibit comparable front-end stalls. In `Web`, front-end stalls arise due to its enormous code footprint due to a rich feature set and the many URL endpoints it implements. In `Cache`, frequent context switches and OS activity cause high front-end stalls. As we will show, these microservices could benefit from larger I-cache and ITLB and other techniques that address instruction misses [177, 278]. In contrast, microservices like `Ads1`, `Ads2`, or `Feed1` do not stand to gain much from greater instruction capacity, leading to conflicting SKU optimization goals.

Second, mispredicted branches make up $3\% - 13\%$ of wasted slots. Branch mispredictions are more rare in data-crunching microservices like `Feed1` and more common when instruction footprint is large, as in `Web`, where aliasing in the Branch Target Buffer contributes a large fraction of branch misspeculations. SKU optimization goals diverge, with some microservices calling for simple branch predictors while others call for higher capacity and more sophisticated prediction.

Third, back-end stalls, largely due to data cache misses, occupy up to 48% of slots, implying that several microservices can benefit from memory hierarchy enhancements.

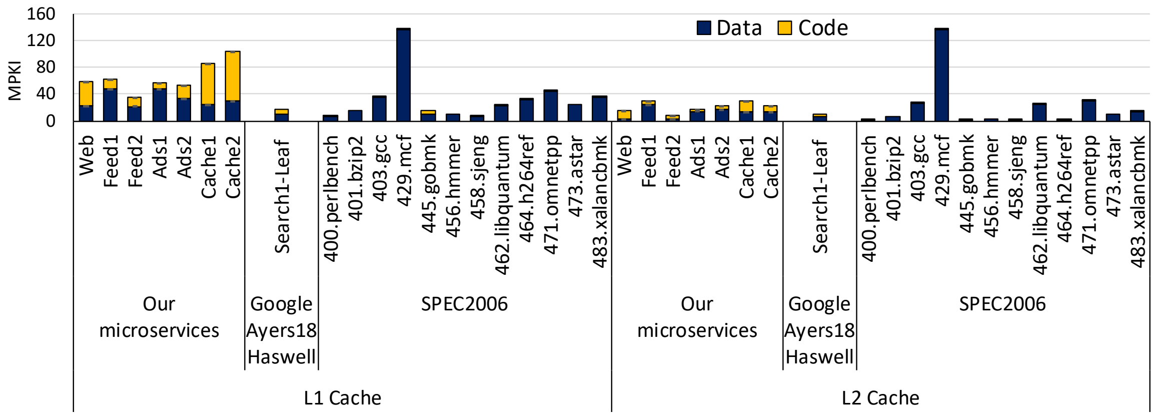


Figure 4.8: L1 & L2 code & data MPKI: Facebook's microservices typically have higher L1 MPKI than comparison applications.

However, microservices like `Web` or `Feed2`, which have fewer back-end stalls, likely gain more from chip area/power dedicated to additional computation resources rather than cache.

4.1.4.2 Cache misses

We provide greater nuance to our front-end and back-end stall breakdown by measuring instruction and data misses in the cache hierarchy. We present code and data Misses Per Kilo Instruction (MPKI) across all cache levels—L1, L2, and LLC in Figs. 4.8 and 4.9, to analyze the overall effectiveness of each cache level. We also show cache MPKI reported by prior work [140] for Google search and our measurements of SPEC CPU2006 on `Skylake20`.

We make the following observations: (1) Our L1 MPKI are drastically higher than the comparison applications, especially for code, and particularly for `Cache1` and `Cache2`. (2) LLC data misses are commonly high in all microservices, especially in `Feed1`, which traverses large data structures. (3) `Web` incurs 1.7 LLC instruction MPKI. These misses are quite computationally expensive, since out-of-order mechanisms do not hide instruction stalls. It is unusual for applications to incur non-negligible LLC instruction misses at all in steady state; few such applications are reported in the academic literature.

Prior works [221, 140, 285, 244] typically find current LLC sizes to be sufficient to encompass server applications' entire code footprint. In `Web`, the large code footprint and

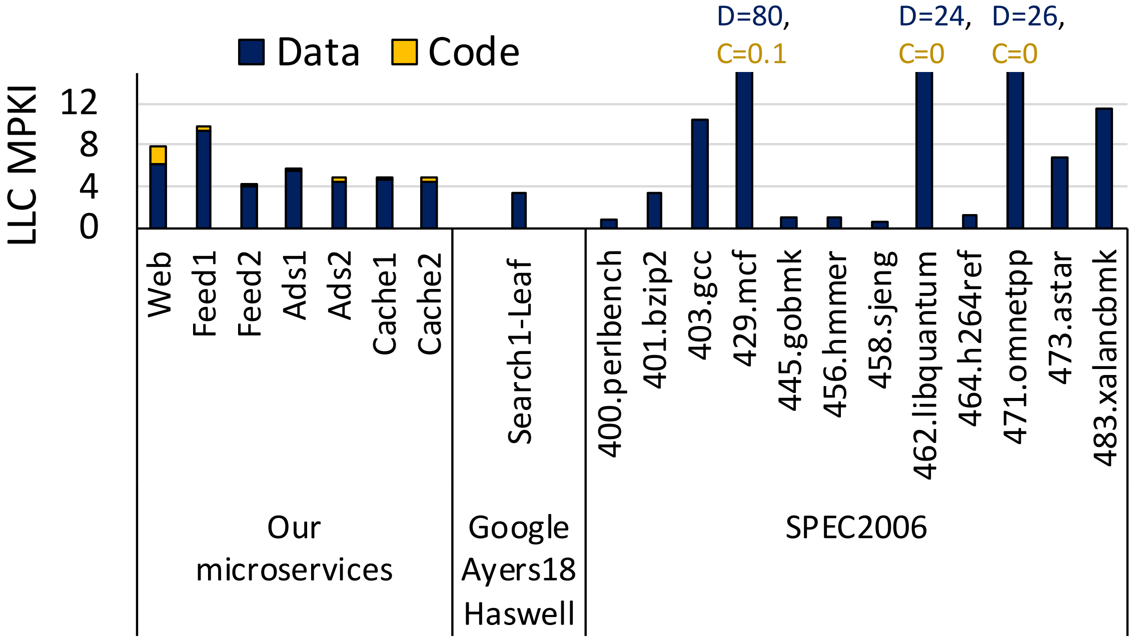


Figure 4.9: LLC code & data MPKI: LLC data MPKI is high across microservices and Web incurs a high code LLC MPKI.

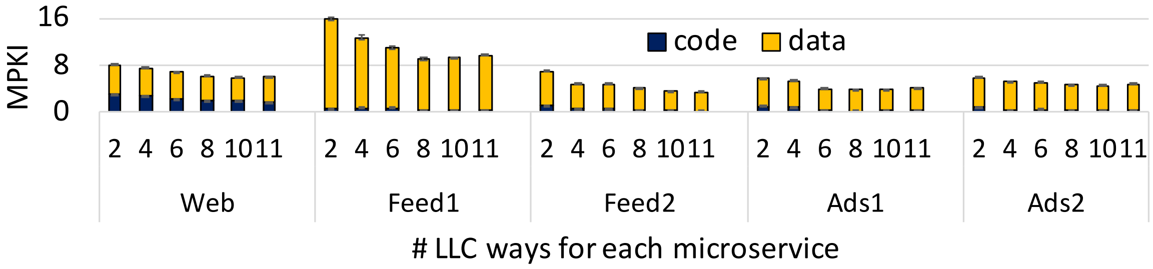


Figure 4.10: LLC code and data MPKI vs. LLC size: Some microservices may benefit from trading LLC capacity for more cores.

high instruction miss rates arise due to the large code cache, frequent JIT code generation, and a large and complex control flow graph. Cache1 and Cache2 incur frequent context switches (see Fig. 4.4) among distinct thread pools executing different code, which leads to code thrashing in L1 and, to a lesser degree, L2. We conclude many microservices can benefit from larger I-caches, instruction prefetching, or prioritizing code over data in the LLC using techniques like Intel's CDP [25, 393].

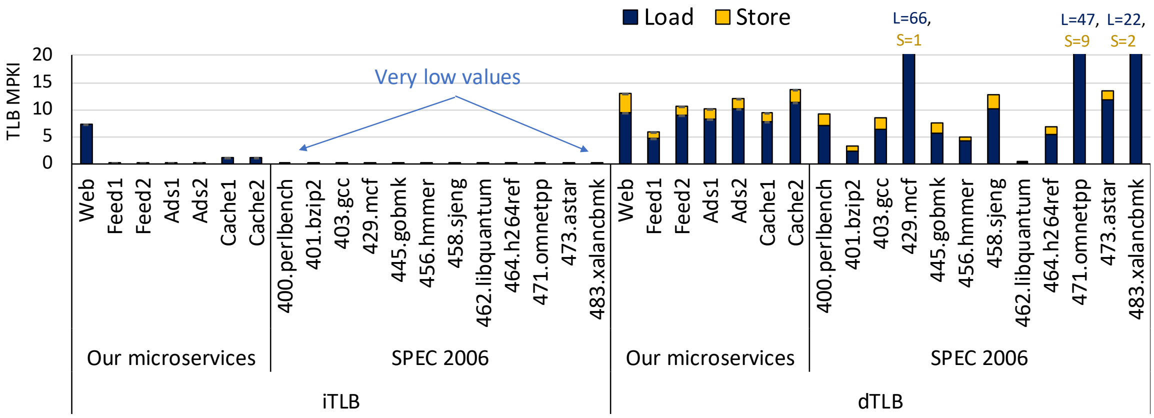


Figure 4.11: I-TLB & D-TLB (load & store) MPKI breakdown: Some microservices can benefit from huge page support.

4.1.4.3 LLC capacity sensitivity

Using CAT [25], we inspect sensitivity to LLC capacity. We vary capacity by enabling LLC ways two at a time, up to the maximum of 11 ways. We report LLC MPKI broken down by code and data in Fig. 4.10. We omit Cache as it fails to meet QoS constraints with reduced LLC capacity. For most microservices, a knee (8 ways) emerges where the LLC is large enough to capture a primary working set without degrading IPC, and further capacity increases provide diminishing returns. For some microservices (e.g., Ads2 and Feed1), the largest working set is too large to be captured. Hence, some services might benefit from trading LLC capacity for additional cores [345].

4.1.4.4 TLB misses

We report instruction and data TLB MPKI in Fig. 4.11. For the D-TLB, we break down misses due to loads and stores. The I-TLB miss trends mirror our LLC code miss observations: Web, Cache1, and Cache2 incur substantial I-TLB misses, while the miss rates are negligible for the remaining microservices. The drastically higher miss rate in Web illustrates the impact of its large JIT code cache.

D-TLB miss rates are more variable across microservices. They typically follow the LLC MPKI trends shown in Fig. 4.9 with the exception of Feed1—despite a relatively high LLC

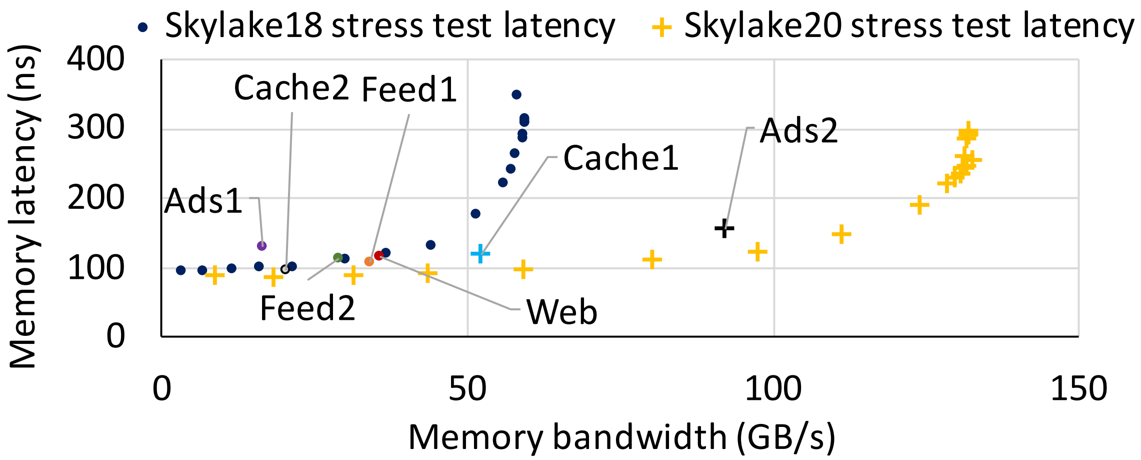


Figure 4.12: Memory bandwidth vs. latency: Microservices under-utilize memory bandwidth to avoid latency penalties.

MPKI of 9.3 it incurs a relatively low D-TLB MPKI of 5.8. `Feed1`'s main data structures are dense floating-point feature vectors and model weights, leading to good page locality despite a high LLC MPKI. However, the other microservices might benefit from software (like static or transparent huge pages) and hardware (e.g., [159, 312, 161, 186, 403, 404, 290]) paging optimizations.

4.1.4.5 Memory bandwidth utilization

We inspect memory bandwidth utilization and its attendant effects on latency due to memory system queuing for each microservice in Fig. 4.12. We first characterize the inherent bandwidth vs. latency trade-off of our two platforms—`Skylake18` in the blue dots and `Skylake20` in the yellow crosses—using a memory stress test [56]. These curves show the characteristic horizontal asymptote at the unloaded memory latency and then exponential latency growth as memory system load approaches saturation. We then plot each microservice's measured average latency and bandwidth, using dots and crosses, respectively, to indicate the service platform.

Microservices like `Web` or `Feed1` have high memory bandwidth utilization relative to the platform capability. Nevertheless, Facebook's microservices cannot push memory bandwidth utilization above a certain threshold—operating at higher bandwidth causes

Table 4.3: Summary of findings and suggestions for future optimizations.

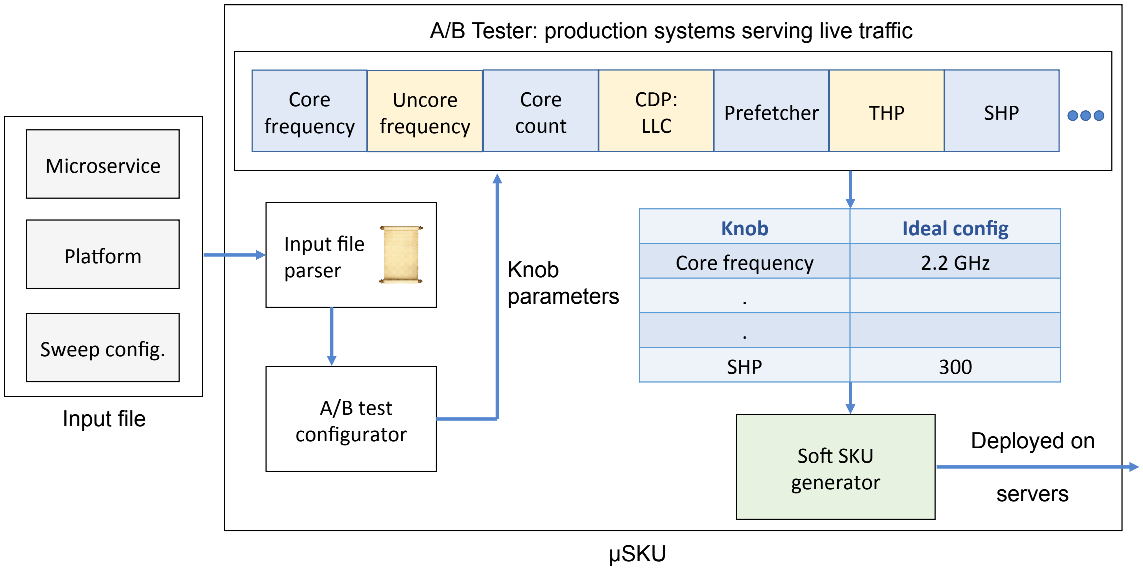
Finding	Opportunity
Diversity among microservices (§4.1.3, §4.1.4)	“Soft” SKUs
Some μservices are compute-intensive (§4.1.3.2)	Enhance instruction throughput (e.g., more cores, wider SMT)
Some μservices emit frequent requests (§4.1.3.2)	Features that support greater concurrency, fast thread switching, and faster I/O
CPU under-utilization due to QoS constraints (§4.1.3.3)	Mechanisms to reduce tail latency, enabling higher utilization
High context switch penalty (§4.1.3.4)	Coalesce I/O, user-space drivers, vDSO, in-line accelerators, thread pool tuning
Substantial floating-point operations (§4.1.3.5)	Optimizations for dense computation (e.g., SIMD)
Large front-end stalls & code footprint (§4.1.4.1, §4.1.4.2)	AutoFDO, large I-cache, CDP, prefetchers, ITLB optimizations, better decode
Branch mispredictions (§4.1.4.1)	“Wider” hardware branch predictors, sophisticated prediction algorithms
Low data LLC capacity utilization (§4.1.4.1, §4.1.4.2, §4.1.4.3, §4.1.4.5)	Trade-off LLC capacity for additional cores
Low memory bandwidth utilization (§4.1.4.5)	Optimizations that trade bandwidth for latency (e.g., prefetching)

exponential memory latency increase, triggering service latency violations. Ads1 and Ads2 operate at higher latency than the characteristic curve predicts due to memory traffic burstiness. The curves also reveal why it is necessary to run Cache1 and Ads2 on the higher-peak-bandwidth Skylake20 platform to keep memory latency low. Nevertheless, several microservices under-utilize available bandwidth, and hence might benefit from optimizations that trade bandwidth to improve latency, such as hardware prefetching [218].

We summarize our findings in Table 4.3.

4.2 “Soft” SKU

Our microservices exhibit profound diversity in system-level and architectural traits. For example, we demonstrated diverse OS and I/O interaction, code/data cache miss ratios, memory bandwidth utilization, instruction mix ratios, and CPU stall behavior. One way to address such distinct bottlenecks is to specialize CPU architectures by building custom hardware server SKUs to suit each service’s needs. However, such hardware SKU diversity



Knob	Ideal config
Core frequency	2.2 GHz
.	
.	
SHP	300

Figure 4.13: *µSKU*: System design.

is impractical, as it requires testing and qualifying each distinct SKU and careful capacity planning to provision each to match projected load. Given the uncertainties inherent in projecting customer demand, investing in diverse hardware SKUs is not effective at scale.

Data center operators aim to maintain hardware resource fungibility to preserve procurement advantages that arise from economies of scale and limit the effort of qualifying myriad hardware platforms. To preserve fungibility, we seek strategies that enable a few server SKUs to provide performance and energy efficiency over diverse microservices. To this end, we propose exploiting coarse-grain (e.g., boot time) parameters to create "soft SKUs", tuning limited hardware SKUs to better support their assigned microservice. However, manually identifying microservice-specific soft-SKUs is impractical since the design space is large, code evolves quickly, synthetic load tests do not necessarily capture production behavior, and the effects of tuning a single knob are often small (a few percent performance change). Hence, we build an automated design tool—*µSKU*—that searches the configuration design space to optimize for each microservice.

4.3 μSKU: System Design

μSKU is a design tool for quick discovery of performant and efficient “soft” SKUs. *μSKU* automatically varies configurable server parameters, or “knobs,” by searching within a predefined design space via A/B testing. A/B testing is the process of comparing two identical systems that differ only in a single variable. *μSKU* conducts A/B tests by comparing the performance of two identical servers (i.e., same hardware platform, same fleet, and facing the same load) that differ only in their knob configuration. *μSKU* collects copious fine-grain performance measurements while conducting automated A/B tests on production systems serving live traffic to search for statistically significant performance changes. Our goal is to ensure that *μSKU* has a simple design so that it can be applied across microservices and hardware SKU generations while avoiding operational complexity. Key design challenges include: (1) identifying performance-efficient soft-SKU configurations in a large design space, (2) dealing with frequent code changes, (3) capturing behavior in production systems facing diurnal or transient load fluctuations, and (4) differentiating actual performance variations from noise through appropriate statistical tests. We discuss how *μSKU*’s design meets these challenges.

We develop a *μSKU* prototype that explores a soft-SKU design space comprising seven configurable server knobs. *μSKU* accepts a few input parameters and then invokes its components—A/B test configurator, A/B tester, and soft SKU generator, as shown in Fig. 4.13. We describe each component below.

Input file. The user provides an input file with the following three input parameters.

(1) *Target Microservice.* Several aspects of *μSKU*’s behavior must be tuned for the specific target microservice. *μSKU* reboots the server while performing certain A/B tests (e.g., core count scaling). Some microservices may not tolerate reboots on live traffic and hence *μSKU* disables these knobs in such cases. Furthermore, *μSKU* disables knobs that do not apply to a microservice. For example, Statically-allocated Huge Pages (SHPs) are

inapplicable to Ads1, since it does not use the APIs to allocate them. Our current *μSKU* prototype estimates performance by measuring the Millions of Instructions per Second (MIPS) rate via EMON [33], which we have confirmed is proportional to several key microservices' throughput (e.g., Web and Ads1). However, we anticipate the performance metric that *μSKU* measures to determine whether a particular soft SKU has improved performance to be microservice specific. In particular, MIPS may be insufficient to measure Cache's throughput, since Cache's code is introspective of performance. (It executes exception handlers when faced with knob configurations that engender QoS violations, which make instructions-per-query vary with performance.) *μSKU* can be extended to perform A/B tests using microservice-specific performance metrics.

(2) *Processor platform.* The available settings in several *μSKU* design space dimensions, such as specific core and uncore frequencies, core counts, and hardware prefetcher options, are hardware platform specific.

(3) *Sweep configuration.* *μSKU*'s A/B tester measures the performance implications of sweeping server knobs either (1) *independently*, where individual knobs are scaled one-by-one and their effects are presumed to be additive when creating a soft SKU, or (2) *exhaustively*, where the design space sweep explores the cross product of knob settings. Note that some microservices receive code updates so frequently (O(hours)) that an *exhaustive* *μSKU* sweep cannot be completed between code pushes. In practice, the gains from *μSKU*'s knobs are not strictly additive. Nevertheless, the knobs do not typically co-vary strongly, so we have had success in tuning knobs *independently*, as the exhaustive approach requires an impractically large number of A/B tests.

A/B test configurator. The A/B test configurator sets up the automatic A/B test environment by specifying the sweep configuration and knobs to be studied.

A/B tester. The A/B tester is responsible for independently or exhaustively varying configurable hardware and OS knobs to measure ensuing performance changes. Our *μSKU* prototype varies seven knobs (suggested by our earlier characterization), but can be extended easily to support more. It varies (1) core frequency, (2) uncore frequency, (3) core count, (4) CDP in the LLC ways, (5) prefetchers, (6) Transparent Huge Pages (THP), and (7) SHPs.

The A/B tester sweeps the design space specified by the A/B test configurator. For each point in the space, the tester suitably sets knobs and then launches a hardware performance counter-based profiling tool [33] to collect performance observations. For each knob configuration, the A/B tester first discards observations during a warm-up phase that typically lasts for a few minutes to avoid cold start bias [363]. Next, the A/B tester records performance counter samples via EMON [33] with sufficient spacing to ensure independence. Finally, when the desired 95% statistical confidence is achieved, the A/B tester outputs mean estimates, which it records in a design space map. It then proceeds to the next knob configuration. The A/B tester typically achieves 95% confidence estimates with tens of thousands of performance counter samples (minutes to hours of measurement). If 95% confidence is not reached after collecting $\sim 30,000$ observations, μSKU concludes there is no statistically significant performance difference and proceeds to the next knob configuration. The final design space map helps identify (with a 95% confidence) the most performance-efficient knob configurations.

Soft SKU generator. The A/B tester's design space map is fed to the soft SKU generator, which selects the most performance-efficient knob configurations. It then applies this configuration to live servers running the microservice. Once the selected soft SKU is deployed, μSKU performs further A/B tests by comparing the QPS achieved (via ODS) by soft-SKU servers against hand-tuned production servers for prolonged durations (including across code updates and under diurnal load) to validate that the soft SKU offers a stable advantage.

4.4 Methodology

We discuss the methodology we use to evaluate μSKU.

Microservices. We focus our prototype μSKU evaluation on the `Web` service on two generations of hardware platforms and on the Ads1 microservice on a single platform. These two microservices differ drastically in our characterization results while both being amenable to the use of MIPS rate as a performance metric. Moreover, the surrounding infrastructure for these services is sufficiently robust to tolerate failures and disruptions we might cause with the *µSKU* prototype, allowing us to experiment on production traffic.

Hardware platforms. To evaluate *µSKU*, we run Web on two hardware platforms—Broadwell16 and Skylake18, and Ads1 on Skylake18 (see Table 4.1). We evaluate Web on both Skylake18 and Broadwell16 to analyze the configurable server knobs' sensitivity to the underlying hardware platform. Henceforth, we refer to Web running on Skylake18 as Web (Skylake) and Broadwell16 as Web (Broadwell).

Experimental setup. We compare *µSKU*'s A/B test knob scaling studies against default production server knob configurations. Some default knob configurations arise from arduous manual tuning, and therefore differ from stock server configurations. We next describe how *µSKU* implements A/B test scaling studies for each configurable knob.

(1) *Core frequency.* Our servers enable Intel's Turbo Boost technology [420]. *µSKU* scales core frequency from 1.6 GHz to 2.2 GHz (default) by overriding core frequency-controlling Model-Specific Registers (MSRs).

(2) *Uncore frequency.* *µSKU* varies uncore (LLC, memory controller, etc.) frequency from 1.4 GHz to 1.8 GHz (default) by overriding uncore frequency-controlling MSRs [241].

(3) *Core count.* *µSKU* scales core count from 2 physical cores to the platform-specific maximum (default), by directing the boot loader to incorporate the isolcpus flag [125] specifying cores on which the OS may not schedule. *µSKU* then reboots the server to operate with the new core count.

(4) *LLC Code Data Prioritization.* *µSKU* uses Intel RDT [25] to prioritize code vs. data in the LLC ways. Our servers' OS kernels have extensions that support Intel RDT via the Resctrl interface [58]. *µSKU* leverages these kernel extensions to vary CDP from one dedicated LLC way for data and the rest for code, to one dedicated way for code and the

rest for data. Default production servers share LLC ways between code and data without CDP prioritization.

(5) *Prefetcher.* Our servers support four prefetchers [32]: (a) *L2 hardware prefetcher* that fetches lines into the L2 cache, (b) *L2 adjacent cache line prefetcher* that fetches a cache line in the same 128-byte-aligned region as a requested line, (c) *DCU prefetcher* that fetches the next cache line into L1-D cache, and (d) *DCU IP prefetcher* that uses sequential load history to determine whether to prefetch additional lines. *μSKU* considers five configurations: (a) all prefetchers off, (b) all prefetchers on (default on `Web` (Skylake) and `Ads1`), (c) only *DCU prefetcher* and *DCU IP prefetcher* on, (d) only *DCU prefetcher* on, and (e) only *L2 hardware prefetcher* and *DCU prefetcher* on (default on `Web` (Broadwell)). *μSKU* adjusts prefetcher settings via MSRs.

(6) *Transparent Huge Pages (THP)*: THP is a Linux kernel mechanism that automatically backs virtual memory allocations with huge pages (2MB or 1GB) when contiguous physical memory is available and defragments memory in the background to coalesce free space [136]. *μSKU* considers three THP configurations (a) *madvise*—THP is enabled only for memory regions that explicitly request huge pages (default), (b) *always ON*—THP is enabled for all pages, and (c) *always OFF*—THP is not used even if requested. *μSKU* configures THP by writing to kernel configuration files.

(7) *Statically-allocated Huge Pages (SHP)*: SHPs are huge pages (2MB or 1GB) reserved explicitly by the kernel at boot time and must be explicitly requested by an application. Once reserved, SHP memory can not be repurposed. *μSKU* varies SHP counts from 0 to 600 in 100-step increments by modifying kernel parameters [228]. *μSKU* can be extended to conduct a binary search to identify optimal SHP counts.

Performance metric. *μSKU* estimates performance in terms of throughput by measuring MIPS rate via EMON [33]. We have verified that MIPS is proportional to `Web` and `Ads1`'s throughput (QPS). We do not measure QPS directly as QPS reported by ODS is not sufficiently fine-grained. We aim to eventually have *μSKU* replace tedious manual knob

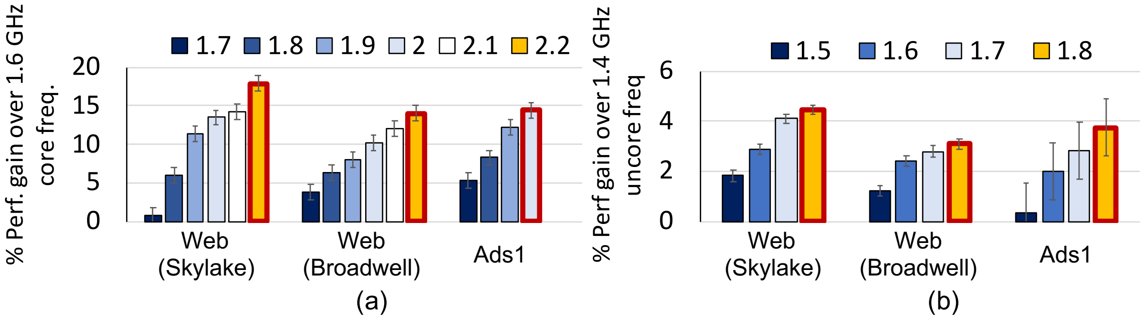


Figure 4.14: Performance trend with (a) core frequency scaling, (b) uncore frequency scaling: The maximum frequency offers the best performance.

tuning for each microservice. Hence, we evaluate *µSKU*-generated soft SKUs against (a) stock off-the-shelf and (b) hand-tuned production server configurations.

4.5 Evaluation

We first present *µSKU*'s A/B test results for all seven configurable server knobs. We then compare the throughput of "soft" server SKUs that *µSKU* discovers against (a) hand-tuned production and (b) stock server configurations.

4.5.1 Knob Characterization

We present *µSKU*'s A/B test results for each knob and compare it against the current production configuration, indicated by thick red bar/point outlines or red axis lines in our graphs. For each graph, we report mean throughput and 95% confidence intervals under peak-load production traffic. For the first three knobs, we find that *µSKU* matches expert manual tuning decisions. However, for the next four knobs, *µSKU* identifies configurations that outperform production settings.

(1) Core frequency. We illustrate *µSKU*'s core frequency scaling analysis in Fig. 4.14 (a). *µSKU* varies core frequency from 1.6 GHz to 2.2 GHz. We report relative throughput (MIPS) gains over cores operating at 1.6 GHz. Our production systems have a fixed CPU power budget that is shared between the core and uncore (e.g., LLC, memory and QPI controller, etc.) CPU components. The current production configuration enables Turbo Boost [420] and runs `Web` (Skylake and Broadwell) at 2.2 GHz and `Ads1` at 2.0 GHz (as indicated by the thick red bar outlines in Fig. 4.14 (a)). `Ads1` must operate at slightly lower frequency because its use of `AVX` operations consumes part of the CPU power budget.

µSKU (1) identifies whether there is a minimum core frequency knee below which throughput degrades rapidly and (2) diagnoses if core frequency trends suggest that the microservice may be uncore bound. `Web`'s and `Ads1`'s throughputs increase precipitously from 1.6 GHz to 1.9 GHz, beyond which *µSKU* reports continued but diminishing throughput gains. These microservices are all sensitive to core frequency, hence, operating at the maximum and enabling Turbo Boost are sensible tuning decisions. *µSKU* configures soft SKUs that operate at 2.2 GHz core frequency for `Web` (Skylake and Broadwell) and 2.0 GHz for `Ads1`, matching experts' tuning.

(2) Uncore frequency. *µSKU* varies the frequency of uncore CPU power domain (including LLC, QPI controller, and memory controller), from 1.4 GHz to 1.8 GHz. We report results normalized to 1.4 GHz uncore frequency (Fig. 4.14 (b)). Our default production configuration runs both microservices at 1.8 GHz uncore frequency. Uncore frequency indicates the degree to which applications are sensitive to access latency when memory and core execution bandwidth are held constant. Both of these microservices are sensitive to memory latency, though the sensitivity is greater in `Ads1`. As with core frequency, *µSKU* selects soft SKUs that operate at the maximum 1.8 GHz for both microservices, again matching the default production configuration.

(3) Core count. We present *µSKU*'s core count scaling results in Fig. 4.15, where we report throughput gain relative to execution on only two physical cores. The grey line indicates ideal linear scaling. *µSKU* scales `Web` (Skylake) to its maximum core count (18 cores) and `Web` (Broadwell) to its maximum (16). We exclude `Ads1` from Fig. 4.15 since its load balancing design precludes *µSKU* from meeting QoS constraints with fewer cores. *µSKU* observes that `Web`'s performance scales almost linearly up to ∼8 physical cores. As

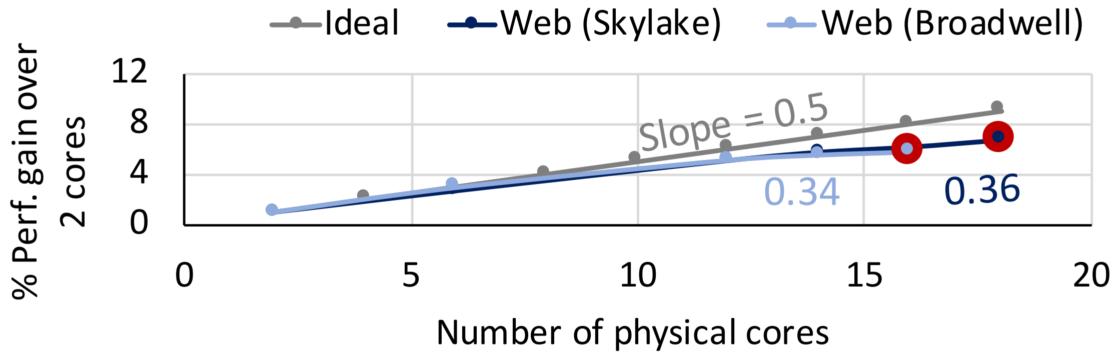


Figure 4.15: Performance trend with core count scaling: Web is core-bound.

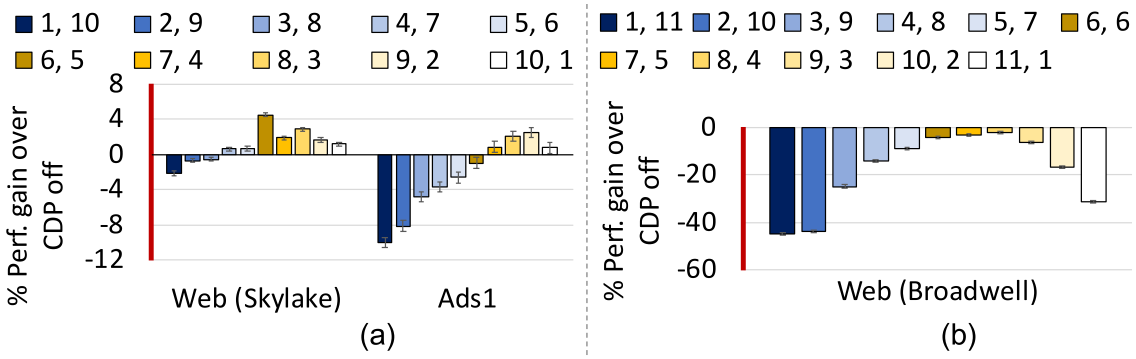


Figure 4.16: Performance trend with CDP scaling: (a) Web (Skylake) & Ads1 benefit due to lower code MPKI (b) Web (Broadwell) has no gains.

core count increases further, interference in the LLC causes the scaling curve to bend down. As with frequency, the best soft SKU selected by μSKU operates with all available cores.

(4) Code Data Prioritization (CDP) in LLC ways. In our earlier characterization (Fig. 4.9), we noted that Web exhibits a surprising number of off-chip code misses. Hence, μSKU considers prioritizing code vs. data in the LLC ways. We report throughput gains over the production baseline (where CDP is not used and code and data share LLC ways) for Web (Skylake) and Ads1 in Fig. 4.16(a) and Web (Broadwell) in Fig. 4.16(b). Skylake18 and Broadwell16 have 11 and 12 LLC ways, respectively. We label each bar with {LLC ways dedicated to data, LLC ways dedicated to code}.

Here we find that Web (Skylake) achieves up to 4.5% mean throughput gain with 6 LLC ways dedicated to data and 5 LLC ways dedicated to code, a configuration that degrades LLC data misses by 0.60 MPKI but improves code misses by 0.30 MPKI. Although this

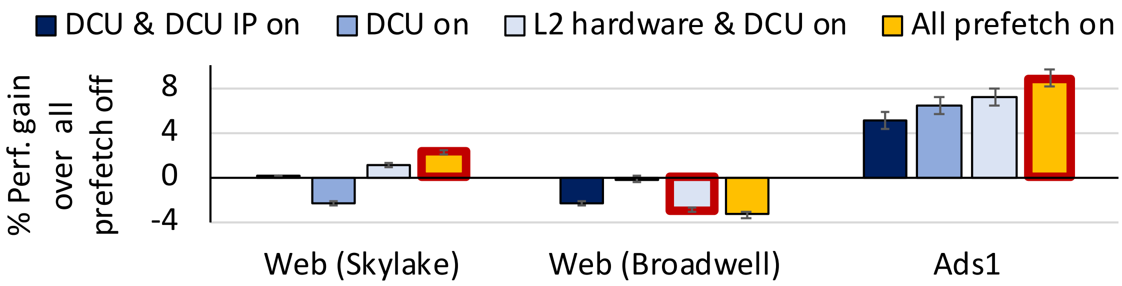


Figure 4.17: Performance trends with varied prefetcher configurations: Turning off prefetchers can improve bandwidth utilization in Web (Broadwell).

configuration increases net LLC misses by almost 0.30 MPKI, it still results in a performance win because the latency of code misses is not hidden and they incur a greater penalty. Similarly, `Ads1` achieves 2.5% mean throughput improvement with 9 LLC ways dedicated to data and 2 LLC ways dedicated to code, sacrificing 0.20 LLC data MPKI to improve LLC code MPKI by 0.06. *μSKU* observes no throughput improvement in `Web` (Broadwell) since it saturates memory bandwidth under all CDP configurations. Hence, *μSKU* can not trade-off increasing the net LLC MPKI to reduce LLC code misses. *μSKU* selects soft server SKUs for `Web` (Skylake) and `Ads1` such that they dedicate {6, 5} and {9, 2} LLC ways for data and code, respectively, improving over the present-day hand-tuned production configuration. *μSKU* does not enable CDP in `Web`'s (Broadwell) soft SKU.

(5) Prefetcher. We report *μSKU*'s results for prefetcher tuning in Fig. 4.17. Our production systems enable (1) all prefetchers on `Web` (Skylake) and `Ads1` and (2) only the L2 hardware prefetcher and DCU prefetcher on `Web` (Broadwell). On `Web` (Broadwell), *μSKU* reveals a $\sim$ 3% mean throughput win over the production configuration when all prefetchers are turned off. `Web` (Broadwell) is heavily memory bandwidth bound when prefetchers are turned on, unlike `Web` (Skylake) and `Ads1`. Turning off prefetchers reduces memory bandwidth pressure, enabling overall throughput gains. In contrast, `Web` (Skylake) and `Ads1` are not memory bandwidth bound, and hence do not benefit from turning off prefetchers.

(6) Transparent Huge Pages (THPs). In our earlier characterization (see Fig. 4.11), we found that `Web` suffers from significant I-TLB and D-TLB misses. Hence, *μSKU* explores

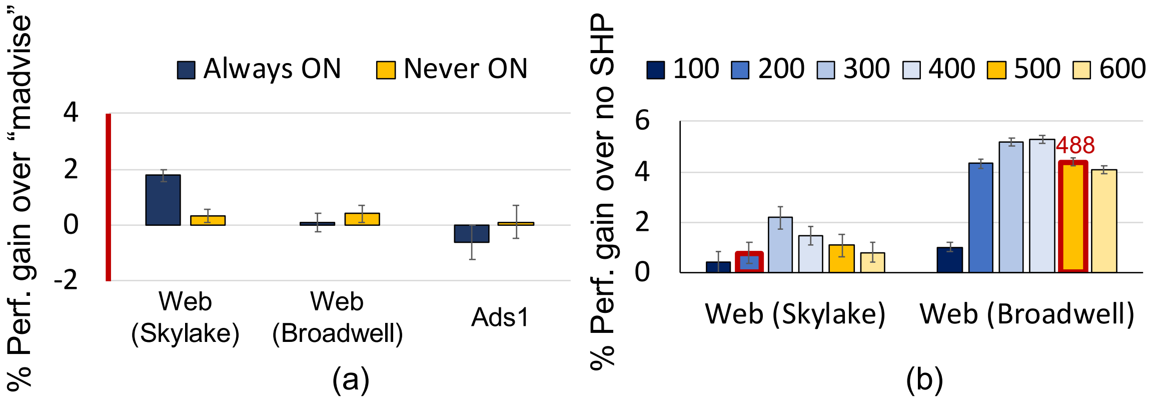


Figure 4.18: Performance trends with varied (a) THP: Web (Skylake) benefits from THP ON, (b) SHP: There is a sweet spot in optimal SHP count.

huge page settings to reduce TLB miss rates. The default THP setting on our production servers is *madvise*, where THP is enabled only for memory regions that explicitly request it. In Fig. 4.18(a), *µSKU* considers (1) always enabling huge pages (*always ON*) and (2) disabling huge pages even when requested (*never ON*), and compares with the default (baseline for the graph) *madvise* configuration.

µSKU identifies a mean 1.87% throughput gain on `Web` (Skylake) when THP is *always ON*, as it significantly reduces TLB misses compared to *madvise*. However, the *always ON* setting does not enhance `Ads1` and `Web` (Broadwell)'s throughput as their TLB miss rates do not improve. Throughput achieved with the *never ON* configuration is comparable with *madvise*, as few allocations use the *madvise* hint.

(7) Statically-allocated Huge Pages (SHPs). We report *µSKU*'s SHP sweep results in Fig. 4.18(b). *µSKU* excludes `Ads1` from this study as it makes no use of SHPs. Our production systems reserve 200 SHPs for `Web` (Skylake) and 488 SHPs for `Web` (Broadwell). *µSKU* shows that reserving 300 SHPs on `Web` (Skylake) and 400 SHPs on `Web` (Broadwell) can outperform our production systems by 1.4% and 1.0% respectively, due to modest TLB miss reductions.

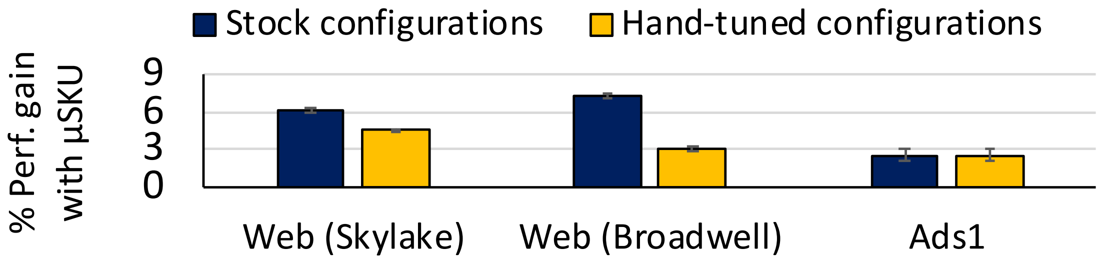


Figure 4.19: Performance gain with *µSKU* over stock and hand-tuned servers: *µSKU* outperforms even hand-tuned production servers.

4.5.2 Soft SKU Performance

µSKU creates microservice-specific soft SKUs by independently analyzing each knob and then composing their best configurations. In Fig. 4.19, we show the final throughput gains achieved by *µSKU*'s soft SKUs as compared to (1) hand-tuned production configurations and (2) stock server configurations (i.e., after a fresh server re-install). The stock configuration comprises (1) 2.2 GHz and 2.0 GHz core frequency for `Web` and `Ads1` respectively, (2) 1.8 GHz uncore frequency, (3) all cores active, (4) no CDP in LLC, (5) all prefetchers turned on, (6) *always ON* for THP, and (7) no SHPs. We listed the hand-tuned configurations in Sec. 4.5.1.

Since these services operate on hundreds of thousands of machines, achieving even single-digit percent speedups with *µSKU* can yield immense aggregate data center cost- and energy-efficiency benefits by reducing a service's provisioning requirement. *µSKU*'s soft SKUs outperform stock configurations by 6.2% on `Web` (Skylake), 7.2% on `Web` (Broadwell), and 2.5% on `Ads1` due to benefits enabled by CDP, prefetchers, THP, and SHP. Interestingly, *µSKU* also outperforms the hand-tuned production configurations by 4.5% on `Web` (Skylake), 3.0% on `Web` (Broadwell), and 2.5% on `Ads1`. We confirmed that the MIPS improvement reported by *µSKU*'s soft SKUs yields a corresponding QPS improvement over a prolonged period (spanning several code pushes) by monitoring fleet-wide QPS via ODS. The statistically significant throughput gains are a substantial win in data centers' efficiency.

µSKU's prototype takes 5-10 hours to explore its knob design space and arrive at the final soft-SKU configurations. Even for knob settings where *μSKU* identifies the same result as manual tuning by experts, the savings in engineering effort by relying on an automated system is significant. A key advantage of *μSKU* is that it can be applied to microservices that do not have dedicated performance tuning engineers.

4.6 Discussion

We discuss open questions and *μSKU* prototype limitations.

Future hardware knobs. Our architectural characterization revealed significant diversity in architectural bottlenecks across microservices. We discussed opportunities for microservice-specific hardware modifications and motivated how soft SKUs can be designed using existing hardware- and OS-based configurable knobs. However, in light of a soft-SKU strategy, we anticipate that hardware vendors might introduce additional tunable knobs. *μSKU* does not currently adjust knobs to address microservice differences in instruction mix, branch prediction, context switch penalty, and other opportunities revealed in our characterization.

QoS and perf/watt constraints. Our microservices face stringent latency, throughput, and power constraints in the form of Service-Level Objectives (SLO). *μSKU*'s prototype performs A/B testing in a coarse-grained design space and tunes configurable hardware and OS knobs to improve throughput. However, *μSKU* does not consider energy or power constraints. QoS constraints are only addressed insofar as we discard parts of the *μSKU* tuning space that lead to violations.

μSKU can be extended to consider a cluster's SLOs' full range. For example, `Cache` executes exception handlers when latency targets are violated, which makes MIPS an inappropriate metric to quantify `Cache` performance. With support for other performance metrics, *μSKU* can perform A/B tests that discount exception-handling code when measuring throughput. With support to also measure system power/energy, *μSKU* can be extended to perform energy- or power-efficiency optimization rather than optimizing only for performance. We leave such support to future work.

Exhaustive design-space sweep. We notice that throughput improvements achieved by individual knobs are not always additive when μSKU composes them to generate a soft SKU. This observation implies that knob configurations may have subtle dependencies on which we might capitalize. An exhaustive characterization that determines a Pareto-optimal soft SKU might identify global performance maxima that are better than those found by our independent search. However, performing an exhaustive search is prohibitive; better search heuristics (e.g., hill climbing [427]) may be required.

μSKU and co-location. Our production microservices run on dedicated hardware without co-runners. Co-location can raise interesting challenges for future work—scheduler systems that map service affinities can be designed in a μSKU-aware manner.

4.7 Related Work

Architectural proposals for cloud services. Several works propose architectures suited to a particular, important cloud service. Ayers et al. [140] characterize Google web search's memory hierarchy and propose an L4 eDRAM cache to improve heap accesses. Earlier work [146] also discusses microarchitecture for Google search. Some works [330, 272, 130] characterize low-power cores for search engines like Nutch and Bing. Trancoso et al. [468] analyze the AltaVista search engine's memory behavior and find it similar to decision support workloads; Barroso et al. [147] show that L2 caches encompass such workloads' working set, leaving memory bandwidth under-utilized. Microsoft's Catapult accelerates search ranking via FPGAs [413]. DCBench studies latency-sensitive cloud data analytics [276]. Studying a single service class can restrict the generality of conclusions, as modern data centers typically execute diverse services with varied behaviors. In contrast, we characterize diverse production microservices running in the data centers of one of the largest social medial providers. We show that modern microservices exhibit substantial system-level and architectural differences, which calls for microservice-specific optimization.

Other works [285, 311] propose architectural optimizations for diverse applications. Kanev et al. [285] profile different Google services and propose architectural optimizations. Kozyrakis et al. [311] examine Microsoft's email, search, and analytics applications, focusing on balanced server design. However, these works do not customize SKUs for particular services.

Academic efforts develop and characterize benchmark suites for cloud services. Most notably, CloudSuite [221] comprises both latency-sensitive and throughput-oriented scale-out cloud workloads. Yasin et al. [495] perform a microarchitectural characterization of several CloudSuite workloads. However, our findings on production services differ from those of academic cloud benchmark suite studies [221, 495, 505, 229, 350]. For example, unlike these benchmark suites, our microservices have large L2 and LLC instruction working sets, high stall times, large front-end pipeline stalls, and lower IPC. While these suites are vital for experimentation, it is important to compare their characteristics against large-scale production microservices serving live user traffic.

Hardware tuning. Many works tune individual server knobs, such as selective voltage boosting [261, 292, 409], exploiting multicore heterogeneity [243, 391, 202], trading memory latency/bandwidth [176, 139, 478, 484], or reducing front-end stalls [295, 505, 312]. In contrast, we propose (1) performance-efficient soft SKUs rather than hardware changes, (2) target diverse microservices, and (3) tune myriad knobs to create customized microservice-specific soft SKUs. Other works reduce co-scheduled job interference [327, 284, 355, 499, 492, 459] or schedule them in a machine characteristics-aware manner [353, 201, 494, 371]. Such studies can benefit from architectural insights provided here.

4.8 Long-Term Impact Potential

We discuss impact potential in terms of (1) long-term impact in industry and academia and (2) potential for follow-on research.

Demonstrated improvements in production systems. This work demonstrates performance improvements on real commodity hardware running production services deployed at hyperscale. The performance benefits from soft SKUs are significant enough at hyperscale to save millions of dollars and also meaningfully reduce the global carbon footprint [446, 5]. To quote an anonymous expert reviewer from the International Symposium on Computer Architecture (ISCA), "*It is a rare ISCA paper where I can get excited about a 7.2% improvement in some parameter, but when that performance improvement is on real hardware measured with production workloads, and is deployed at significant scale, it suddenly becomes very interesting."*

Soft SKUs enable cost-efficient fungible commodity hardware in data centers, while still reaping significant performance benefits. They reduce procurement, testing, time-to-market, upgrade, accelerator development, and energy costs. Additionally, μSKU improves performance engineers' productivity, reducing labor costs.

Influence in next generation commodity server designs. Our comprehensive system-level and architectural characterization of real-world production microservices has identified abundant opportunities for future impactful academic and industry research. Expert reviewers' comments include "*The workload characterization study is very detailed and offers many interesting insights into large scale production applications that has a potential to inspire many avenues of future work"* and *"This characterization collectively provides an excellent snapshot into the behavior of very important and widely deployed cloud services, and can extensively facilitate future research."*

Our characterization results suggest several directions to maintain the performance improvement rate for general-purpose, commodity servers, triggering a significant shift in the hardware industry. Specifically, several hardware vendors are actively pursuing hardware modifications based on our findings[2]. Moreover, in light of our soft-SKU strategy, hardware vendors are starting to incorporate additional configurable processor knobs that address microservice differences in instruction mix, branch prediction, context switch penalty, and

[2] Fun fact: The hardware engineering VP at Intel set up a call with Intel's engineers the very next day after our ISCA paper [446] was made public, to discuss hardware optimization opportunities based on our findings.

other opportunities revealed in our characterization. We also proposed new CPU knobs (e.g., Branch Target Buffer ways) that can be made configurable to create finer-grained soft SKUs. Several of these proposals, along with our study's key conclusions have influenced the design of the upcoming generation of server-class processors [5].

Enabling new architecture trends. In light of recent big architecture trends such as dark silicon, reliability and yield challenges with large chips, and shrinking technology nodes, soft SKUs can further improve the performance of a wide range of applications while still retaining hardware fungibility. For example, dark silicon can result in the creation of more performance and energy efficient soft SKUs by selectively enabling new processor knobs.

Benchmarking. There is immense value in validating commonly-used benchmarks with real-world application behaviors. Our characterization reveals the severity of hyperscale bottlenecks that are not often captured by open-source benchmarks [221]. Hence, our characterization drove hardware vendors to consider more representative benchmarks (in place of traditional ones they used for decades) when evaluating hardware designs[3]. We expect our comprehensive analysis to drive continued benchmarking efforts that represent the severity of overheads in production-grade software.

Industry impact. Our characterization and consequent SoftSKU proposal resulted in Facebook creating a team of engineers to investigate the fleet-wide impact of enabling further available processor knobs, such as SIMD width, Intel's Cache Allocation Technology, and Intel's Memory Bandwidth Allocation to achieve additional soft SKU performance benefits across the global fleet of cost-efficient, fungible commodity hardware [5].

Soft SKUs in diverse hyperscale production environments. Our conversations with several hyperscale enterprises revealed an interest in employing the SoftSKU strategy in their production data center environments since their applications also face a great diversity in system-level and architectural bottlenecks [285]. The μSKU design tool is simple and

[3] To quote an Intel researcher, "*We were driving blind until seminal works like these came along and told us to refocus our design efforts on more representative applications.*"

practical, and can be easily deployed at hyperscale in any data center where services run on dedicated hardware without co-runners (as with the production microservices studied in this chapter). However, some hyperscale enterprises co-locate microservices; we have had conversations with them about designing schedulers that map service affinities in a SoftSKU-aware manner.

Future research potential. Several researchers are already working on hardware and software optimizations based on bottlenecks identified in our characterization. (1) Researchers at the University of Michigan, University of Pennsylvania, UC Santa Cruz, Texas A&M, and Intel Labs are pursuing hardware and compiler optimizations to mitigate instruction misses in front-end microservices (e.g., `Web`). (2) Researchers at UT Austin, Texas A&M, and Intel Labs are working on mitigating branch mispredictions through various machine learning techniques. (3) Researchers at the University of Michigan, Georgia Tech, Harvard, and Carnegie Mellon University are pursuing hardware modifications to enable fast I/O. In Section 4.9, we highlight some of our own follow-on research.

Tech forums. Our work has triggered rich conversations amongst computer scientists [87, 5]. For example, our work generated discussions in the Real World Technologies forum [87] on topics such as new processor knobs, compiler effects, and hardware modifications.

4.9 Follow-On Research

In follow-on work [298, 299, 297][4], my co-authors and I mitigated the architectural bottlenecks in the frontend of the processor pipeline (e.g., instruction cache misses and branch mispredictions) that I found to be significant in microservices (detailed in Section 4.1). We used profile-guided optimization techniques to inform frontend operations (e.g., I-cache and BTB prefetching and replacement decisions) to achieve near-ideal frontend performance.

[4] This follow-on research was performed in collaboration with fellow graduate student Tanvir Ahmed Khan (who is the lead author) and other contributors [298, 299, 297]. Concepts summarized in Section 4.9 will be detailed in Khan's dissertation.

Mitigating I-cache misses with profile-guided prefetching [298]. In this work, we investigated the challenges of effective instruction prefetching in the I-cache. We used insights derived from our investigation to develop *I-SPY*, a novel profile-driven prefetching technique. *I-SPY* uses dynamic miss profiles to drive an offline analysis of I-cache miss behavior, which it uses to inform prefetching decisions.

Two key techniques underlie *I-SPY*'s design: (1) *conditional prefetching*, which only prefetches instructions if the program context is known to lead to misses, and (2) *prefetch coalescing*, which merges multiple prefetches of non-contiguous cache lines into a single prefetch instruction. *I-SPY* exposes these techniques via a family of light-weight hardware code prefetch instructions. We studied *I-SPY* in the context of nine data center applications and showed that it provides an average of 15.5% (up to 45.9%) speedup and 95.9% (up to 98.4%) reduction in instruction cache misses, outperforming the state-of-the-art prefetching technique proposed by Google [141] by 22.5%. We demonstrated that *I-SPY* achieves 90.5% of the performance of an ideal cache with no misses.

Mitigating I-cache misses with profile-guided replacement [299]. We investigated why existing I-cache miss mitigation mechanisms achieve sub-optimal performance for data center applications. We found that widely-studied instruction prefetchers fall short due to wasteful prefetch-induced cache line evictions that are not handled by existing I-cache replacement policies [124, 269, 270]. Existing replacement policies [124, 269, 270] are unable to mitigate wasteful evictions as they lack complete knowledge of a data center application's complex program behavior.

To make existing replacement policies [124, 269, 270] aware of eviction-inducing program behaviors, we presented *Ripple*, a novel software-only technique that profiles programs and uses program context to inform the underlying I-cache replacement policy about efficient replacement decisions. *Ripple* carefully identifies program contexts that lead to I-cache misses and sparingly injects "cache line eviction" instructions [489] in suitable program locations at link time. We evaluated *Ripple* using nine popular data center applications [298] and demonstrated that *Ripple* enables any I-cache replacement policy to achieve speedup that is closer to that of an ideal I-cache. Specifically, *Ripple* achieves an average performance improvement of 1.6% (up to 2.13%) over prior work [124, 269, 270] due to a mean 19% (up to 28.6%) I-cache miss reduction.

Mitigating BTB misses with profile-guided prefetching [297]. To overcome frontend stalls in the processor pipeline, modern server-class processors implement a decoupled frontend with Fetch Directed Instruction Prefetching (FDIP) [417, 455, 398, 421, 239]. We characterized the limitations of a decoupled frontend processor with FDIP and found that FDIP suffers from significant BTB misses. We also found that existing techniques (e.g., stream prefetchers [488, 487] and predecoders [296, 313]) are unable to mitigate these misses, as they rely on an incomplete understanding of a program's branching behavior.

To address the shortcomings of existing BTB prefetching techniques, we proposed *Twig*, a novel profile-guided BTB prefetching mechanism. *Twig* analyzes a production binary's execution profile to identify critical BTB misses and inject BTB prefetch instructions into code. Additionally, *Twig* coalesces multiple non-contiguous BTB prefetches to improve the BTB's locality. *Twig* exposes these techniques via a new BTB prefetch instruction. Since *Twig* prefetches BTB entries without modifying the underlying BTB organization, it is easy to adopt in modern processors. We studied *Twig*'s behavior across nine widely-used data center applications [298], and demonstrated that it achieves 20.86% (up to 145%) performance speedup over a baseline 8K-entry BTB, outperforming the state-of-the-art BTB prefetch mechanism [313] by 19.82% (on average).

Mitigating BTB misses with profile-guided replacement. We found that prior BTB prefetching techniques [296, 313] offer limited performance gains over FDIP, falling significantly short of a perfect BTB. We observed that the optimal Belady's replacement policy [152] significantly closes the performance gap, achieving near-ideal BTB performance. Hence, there is a need for a better BTB replacement policy to achieve near-ideal BTB performance.

Upon characterizing existing replacement policies [124, 269, 270], we noted that existing policies do not account for the access pattern bias among different program branches, inhibiting them from predicting and evicting the branch that is accessed furthest in the future. We proposed a novel, profile-guided BTB replacement mechanism called *HWC*, that accounts for the access pattern bias among different branches to make replacement decisions. *HWC* analyzes a production binary's execution profile to identify branch access pattern biases to inject BTB replacement instructions into code. We evaluated *HWC* for nine modern data center applications [298] and showed that *HWC* achieves near-ideal BTB performance.

4.10 Chapter Summary

We summarize our contributions as follows:

- **A comprehensive characterization of production microservices' system-level bottlenecks.** We presented a detailed analysis of the system-level bottlenecks experienced by key production microservices in one of the largest social media platforms today.

- **A detailed study of production microservices' architectural bottlenecks.** We presented a comprehensive characterization of shortcomings in commodity hardware architectures running hyperscale production microservices, highlighting potential hardware design optimizations. This characterization has (1) influenced the design of commercial server-class commodity processors and (2) driven more representative benchmarking efforts.

- **SoftSKU and μSKU:** We introduced a design approach and associated tool, SoftSKU and μSKU, that automatically tunes important configurable server parameters to enable existing commodity hardware to efficiently support diverse hyperscale microservices. Soft SKUs have the potential to maintain commodity processors' performance improvement rate (despite the decline in hardware performance scaling), and thereby trigger a significant shift in the hardware industry while enabling new architecture trends.

- **A detailed performance study of configurable server parameters tuned by μSKU.** We demonstrated that the SoftSKU approach and associated μSKU design tool significantly improves the performance efficiency of real-world, production microservices that service billions of users, saving millions of dollars and meaningfully reducing the global carbon footprint.

The variety and complexity of microservices in hyperscale data centers has grown precipitously over the last few years to support a growing user base and an evolving product portfolio. Despite accelerating microservice diversity, there is a strong requirement to limit diversity in underlying commodity server hardware to maintain hardware resource fungibility, preserve procurement economies of scale, and curb qualification/test overheads. As such, there is an urgent need for strategies that enable limited commodity server CPU architectures (a.k.a "SKUs") to provide performance and energy efficiency over diverse microservices. To this end, we undertook a comprehensive characterization of the top seven production microservices that run on the compute-optimized hyperscale data center fleet at Facebook.

Our characterization revealed profound diversity in OS and I/O interaction, cache misses, memory bandwidth utilization, instruction mix, and CPU stall behavior. Whereas customizing a CPU SKU for each microservice might be beneficial, it is prohibitively expensive. Instead, we motivated the need for "soft SKUs", wherein we exploited coarse-grain (e.g., boot time) configuration knobs to tune the platform for a particular microservice. We developed a tool, μSKU, that automates search over a soft-SKU design space using A/B testing in production systems and demonstrated how it obtains statistically significant gains (up to 7.2% and 4.5% performance improvement over stock and production servers, respectively) with no additional hardware requirements.

We also described follow-on work [298, 299, 297] on mitigating the architectural bottlenecks in the frontend of the processor pipeline (e.g., I-cache misses) that we found to be significant in microservices (detailed in Section 4.1). We developed a series of profile-guided optimizations that observe dynamic information from various frontend micro-architectural structures to inform these structures' decisions during runtime, thereby achieving near-ideal processor frontend performance.

CHAPTER V

Redesigning Commodity Server Architectures for Efficient Event Notification at Hyperscale

As described in Chapter I, a microservice typically communicates with numerous I/O devices and queues. For example, a microservice may receive network requests from tens to hundreds of microservices via the Network Interface Controller alone. In modern systems, a microservice typically receives work from an I/O queue by either (1) *spin-polling* for new events or (2) using *interrupts* via blocking system calls that yield the CPU if no work is available.

In Chapter IV, we observed that key microservices (e.g., Facebook's `Cache`) face high I/O notification latency and in Chapter VI, we will demonstrate that I/O notification can consume 52% of the total CPU cycles executed by Facebook's key microservices [444]. Hence, we ask the question: why do existing I/O notification paradigms fall short in the context of hyperscale microservices, resulting in high I/O event notification overheads?

We[1] comprehensively characterize widely-used I/O event notification paradigms such as *spin-polling*, *interrupts*, and `MWAIT`, and examine the overheads they induce. Surprisingly, we find that the latency of existing notification mechanisms, which used to be insignificant for monolithic services, can dominate microservice latency. For example, blocking incurs

[1] Some of the work in this chapter was performed in collaboration with my Ph.D. committee member, Margo I. Seltzer, and my Ph.D. advisor, Thomas. F. Wenisch [446]. Therefore, I use the "we" pronoun in this chapter to acknowledge their involvement in this work.

microsecond-scale overheads from thread wakeups, context switches, and unexpected hardware or OS actions, such as a slow transition from a low-power CPU mode. With microservices' microsecond-scale service times, the I/O software stack's latency becomes comparable to computation time.

Several microservices spin-poll on I/O queues [89] to avoid the I/O stack's microsecond-scale overheads. Although spin-polling avoids OS-induced delays [449], a spinning loop often polls empty queues before finding work in non-empty ones. Since some queues are inherently more frequently accessed than others, this traffic imbalance causes many queues to be empty at any given time. Checking empty queues wastes CPU cycles, decreasing peak throughput and increasing latency. We find that spinning through tens to hundreds of I/O queues increases tail latency by tens of microseconds. These findings suggest that widely-used I/O notification paradigms are not latency-efficient when tens (or more) microservices interact in complex ways. Hence, prior conclusions on I/O event notification paradigms must be revisited for hyperscale microservices.

As a part of this dissertation's hardware contributions, we present *μNotify*, a hardware-assisted shared-memory event notification paradigm that facilitates performance-efficient interactions with numerous I/O queues. Unlike interrupts and spin-polling, *μNotify* achieves low latency, queue scalability, and service priorities. When *μNotify* awaits notifications on an I/O queue, a work item written to the queue by an I/O device triggers a cache line invalidation coherence message in *μNotify*'s private cache. *μNotify*'s key idea is to observe these cache line invalidations and use them as low-overhead I/O event notification.

μNotify is composed of a programming model front-end and a hardware microarchitecture back-end that work in cooperation. *μNotify* dedicates a single CPU core to manage events from numerous I/O queues (as is done in prior work [389]). At a high level, it uses L1 cache invalidation information to detect new work item arrivals. First, the front-end loads a set of well-known I/O queue memory locations (called *doorbells)* to monitor in its L1 private cache. Each doorbell corresponds to an I/O queue. The back-end uses invalidations

to detect when a doorbell is “rung” and sets a suitable bit in a newly-introduced special hardware register, accessible to the front-end.

Our proposed hardware allows *μNotify* to identify I/O queues with new work in near-constant time. The main components of *μNotify*’s microarchitectural back-end are (1) a small extension to the cache coherence controller and (2) a special hardware register. The extended controller tracks write transactions, i.e., invalidations, that indicate new work arrival. The back-end uses the transaction’s address to index into a bit-readable/writable new hardware register, i.e., a *ready-vector*.

The front-end spin-polls on the ready-vector to detect a write. When a ready-vector bit is set due to work arrival, the front-end atomically reads and resets the ready-vector to re-arm notification. It uses existing instructions [40] to efficiently identify a “set” bit that corresponds to the queue with new work. It re-loads the invalidated doorbell to re-arm monitoring, and selects the next queue to service according to a pre-defined service policy. The front-end effectively functions as a task scheduler, sorting the order of ready queues to be serviced.

μNotify achieves queue scalability as the back-end reports which queues to process, rather than the software interrogating many empty queues. It is also immune to microsecond-scale I/O stack processing latencies, as it bypasses the OS.

We implement *μNotify* in a real system and evaluate it using some of the microservices from the μSuite benchmark suite [448] (detailed in Chapter II). We emulate *μNotify*’s back-end by extending the I/O device’s operation to set an appropriate bit in a shared memory bit-vector. A real system emulation allows comparing *μNotify*’s performance to state-of-the-art mechanisms—spin-polling and interrupts. Across our suite of microservices, we demonstrate that *μNotify* achieves a peak throughput and tail latency that is (on average) 15.6x and 14.2x respectively better than the state-of-the-art notification paradigms, with an overhead that is less than 500 ns.

5.1 Why do Widely-Used Notification Paradigms Fall Short?

We begin to uncover the challenges that microservices face with event notifications with a brief description of their key requirements. We then analyze how well widely-used I/O notification paradigms meet those requirements.

5.1.1 Microservice Requirements

Since web service requests frequently propagate serially through tens to hundreds of microservices [230], it is critical for each microservice to meet two key requirements.

Achieve microsecond-scale service latency. Each microservice must often achieve O(10 - 100μs)-scale service latencies to meet end-to-end O(100ms)-scale SLOs. A microservice must also receive and process work items from many other microservices with high throughput. At such low latencies and high throughputs, microsecond-scale system overheads from OS and I/O interactions that are insignificant for monoliths can dominate microservice performance. For example, the latency to access modern I/O devices such as emerging storage-class and disaggregated memories [119, 121, 145, 386], 100+ gigabit networking devices [163], and high-throughput accelerators [173, 407] is as low as single-digit microseconds. Hence, it is important to understand how microsecond-scale I/O interactions affect microservice performance.

Manage numerous I/O queues. Microservices must often manage communication with numerous I/O devices that each present multiple connections, such as Network Interface Controllers (NICs) [357], Solid State Drives (SSDs) [288, 326], persistent memory devices [57, 93], and accelerators [179]. For example, a microservice can listen on several network connections via the NIC when expecting requests from multiple clients. Since a microservice often interacts with tens to hundreds of microservices [230] and numerous I/O devices, it must handle a large number of I/O queues. Moreover, when many microservices are co-located on the machine [467], establishing communication between them is more efficient using shared memory queues than expensive network connections. Such shared memory queues further increase the number of work queues to monitor. Hence, it is critical for a microservice to efficiently manage events from tens to hundreds of work queues corresponding to several clients and I/O devices.

We now characterize the performance efficiency of widely-used I/O event notification paradigms when faced with events from numerous I/O queues. The two most widely-used I/O notification paradigms are OS interrupts and spin-polling. Prior work [449] has shown that the trade-off between these paradigms is intrinsic: polling reduces latency, while interrupts free a waiting CPU to perform other work. Hence, interrupts can induce higher latency as they additionally execute OS interrupt handling code, incurring context switches and thread wakeup delays [154, 339]. Other notification mechanisms include the Intel x86 `MWAIT` [343] or ARM `WFE` [496] instruction variants that halt CPU execution until the contents of a single memory address or address range change. We revisit these paradigms and examine their performance and scalability.

5.1.2 Interrupts

In interrupt-driven notification, threads await new work via blocking system calls, yielding the CPU if no work is available. Threads block on I/O interfaces (e.g., `select()` or `epoll()` system calls) awaiting work. Several real-world microservices such as Facebook's `Web` [444], Redis BLPOP [18], Azure SQL [16], Google Cloud SQL's Redmine [22, 323], and MongoDB replication [73] use interrupts for notification. Due to their wide adoption, interrupt notification has been extensively studied in prior work [213, 433]. We discuss four widely-used Linux user-level notification APIs that build upon kernel interrupt mechanisms—`select()`, `poll()`, `epoll()`, and `libevent()`—in the context of microservices.

Select(). `Select` allows waiting on events to multiple file descriptors. Before invoking `select`, a microservice creates file descriptors and bitmaps by asserting bits mapping to relevant file descriptors. On its return, `select` overwrites these bitmaps indicating which descriptors are "ready" with work. The service scans the bitmaps to discover ready descriptors.

`Select` faces a high overhead as it overwrites bitmaps—after each event, bitmaps are created, copied into the kernel, scanned, subsetted, copied out of the kernel, and then scanned by the service. These costs grow with an increase in the number of monitored descriptors [452]. A microservice might scan hundreds or thousands of descriptors for every event, as each descriptor can map to a client; scanning delays further exacerbate service performance. We conclude that `select` falls short in satisfying the microservice latency and scalability requirements.

Poll(). Unlike `select`, `poll`'s descriptor bitmap maps only to relevant descriptors and is not overwritten. But, `poll` has higher copy overheads than `select` with a large number of descriptors [80]. `Poll`'s work is also proportional to the monitored descriptors rather than to the number of events.

Epoll(). Unlike `select` and `poll`, `epoll` returns the ready file descriptors (obtained via `epoll_wait()`) and minimizes bitmap copy latency. Hence, many real world systems, including Remote Procedure Call libraries upon which microservices are built [45, 37], use `epoll`-driven notification. We also find `epoll` more scalable than `select` and `poll`. Hence, we next study interrupt notification challenges using `epoll`.

Experimental setup. To measure `epoll`'s user-mode notification latency, we develop a microbenchmark composed of a producer and consumer process. The producer represents events from many I/O devices or client microservices. The consumer represents a microservice that uses `epoll` to monitor I/O. We use shared memory queues to model the I/O the producer sends to the consumer. To mimic events from many I/O queues, the producer picks a random shared queue from a list of known queues and writes a ns-scale timestamp using RDTSC [392], generating an event. Upon receiving the event via `epoll`, the consumer notes the producer's "sent" timestamp and the current time to estimate the interrupt latency.

The producer operates in (1) a closed-loop by maintaining a fixed number of outstanding

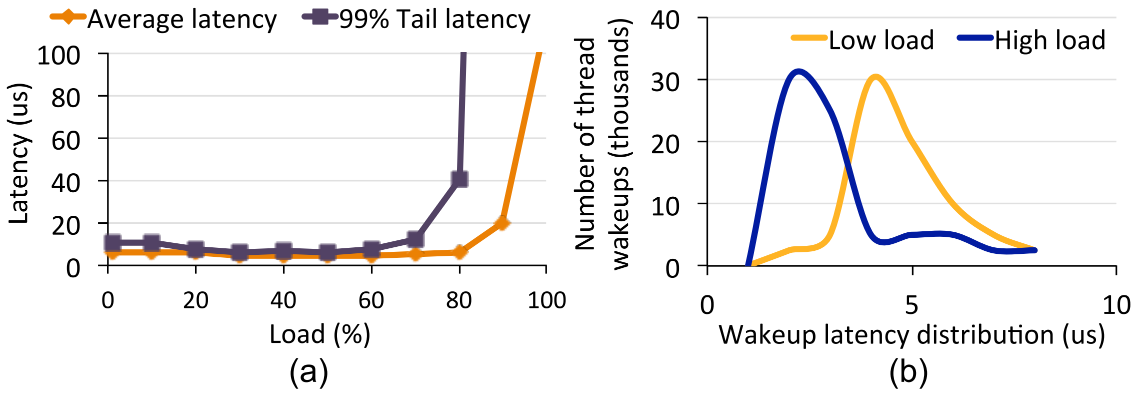


Figure 5.1: (a) Epoll latency with increasing load: Interrupts face μs-scale context switches & thread wakeups. (b) Thread wakeups at low & high load: Low-load wakeups are costlier.

work items [501] when measuring throughput and (2) an open-loop mode with Poisson inter-arrivals [172] when measuring latency. We map each process to a dedicated CPU core in a single-socket system, hence, RDTSC is synchronized across cores.

Interrupt notification latency. To report the bare-bones user-mode interrupt notification latency via `epoll`, we first measure the latency when the producer sends work via a single shared queue, mimicking events from a single I/O queue. In Fig. 5.1(a), we show the average and the $99^{th}\%$ tail latency across increasing load, i.e., work items/events sent per second.

We make the following observations. First, we note that the average interrupt notification latency costs $3-6\mu$s and the tail latency is $6-10\mu$s even at low (<20%) and intermediate (20% - 60%) load. These single-digit microsecond overheads primarily arise from two microsecond-scale context switches and consequent user thread wakeup delays.

Such microsecond-scale overheads are often insignificant for O(100ms)-SLO monolithic services (e.g., Lucene web search [359]). However, microsecond–scale microservices differ intrinsically: OS context switch and thread wakeup delays can dominate microservice latency [106]. For example, a single 10μs wakeup implies a 10% latency penalty for a request to a 100μs SLO microservice (e.g., McRouter [501]). Moreover, since tens to hundreds of microservices often communicate in series, the interrupt latency can quickly add up and dominate the end-to-end application latency. It is hence unsurprising that data center operators find that microservices can spend 52% of their cycles serving interrupts [285, 444].

Prior work proposed disabling interrupts to improve performance [210]. In such cases, for the interrupt handler to be fired, the service must wait for interrupts to be enabled again, degrading latency further. For example, with Virtual Machines (VMs), a VM-exit must occur for the hypervisor to serve an interrupt, precipitating overheads from the VM-exit, backing registers, and flushing TLBs, in addition to the bare-bones interrupt processing latency we demonstrate in Fig. 5.1(a).

Second, we notice a slightly higher average and tail latency at low load (<20% of saturation) compared to intermediate load. At low load, the OS might transition to a low-power mode that delays thread wakeups [261, 449]. In Fig. 5.1(b), we show thread wakeup delays (reported as latency histograms) gathered via the BPF run queue (scheduler) latency tool [66]. This data shows that most thread wakeups are slower ($3-5\mu$s) at low load than at higher load (<3μs). As load increases, the OS performance also tends to improve from better temporal and spatial locality caused by batching effects in the OS stack.

Third, at near-saturation load, interrupt latency spikes due to unbounded queuing delays (also reported in prior work [449]).

Notification latency with many queues is similar to data in Fig. 5.1. We conclude that with microservices' microsecond-scale service times, the I/O stack's latency becomes comparable to service latency and must be aggressively optimized.

Libevent. `Libevent` is a library that offers interrupt API portability across operating systems. Since `libevent` uses `epoll` under the hood, we find its performance comparable to Fig. 5.1. Hence, in the rest of this chapter, we report `epoll` performance when comparing against interrupt notification.

5.1.3 Spin-polling

Many microservices opt to spin-poll [89], solely to avoid context switches, thread wakeups, and unexpected hardware/OS actions, such as slow transitions to a low-power mode [148]. For example, real-world services such as Intel's DPDK Poll Driver [83], Redis replication [89], Redis LPOP [67], DoS attacks/defenses [380, 416, 442], and GCP Health Checker [97] use spin-polling. Similarly, Software Data Planes (SDP) [238] and kernel-bypass techniques [154, 274, 289, 329, 402, 408] also rely on spin-polling to deliver high performance. Since SDPs manage a large number of I/O queues, they are also susceptible to the problems we find with spin-polling.

Prior works [242, 277] demonstrated that when multiple spinning threads are required to sustain a high load, microsecond-scale thread contention can cause pathologically poor performance. Multiple spinning threads also (1) consume more cores, (2) consume more power, (3) induce faster processor aging, and (4) adversely affect microservices co-running on Simultaneously Multi-Threaded cores [352, 353]. Hence, several other works [291, 449] and existing systems [238] advocate using a single spinning thread to pick up I/O work items.

We find that a single spinning core lacks queue scalability. A spinning core must iterate through all I/O queues at full-tilt even when there are no work items in any queue. Spinning through empty queues in search of the next ready work item in a non-empty queue can cause long traversal delays. Traversal delays are more pronounced when I/O traffic is unbalanced, i.e., when a subset of queues are empty most of the time. Since the time required to process a work item is usually short (i.e, a few microseconds [230, 238]), particularly for middle-tier microservices [449], missing on numerous empty queue heads might take longer than processing a ready queue.

Throughput. To measure spin-polling's performance, we first study how a consumer microservice's peak throughput changes as a function of the queue count. We use the same experimental setup described above, but this time, the consumer spin-polls through I/O queues to find work. We use various I/O traffic patterns: *Fully Balanced (FB)*, where traffic passes through all the queues (represents all active queues); *Proportionally Concentrated (PC)*, where traffic passes through 20% of the queues all the time and through the rest with a 5% probability (represents when some queues become active based on an event);

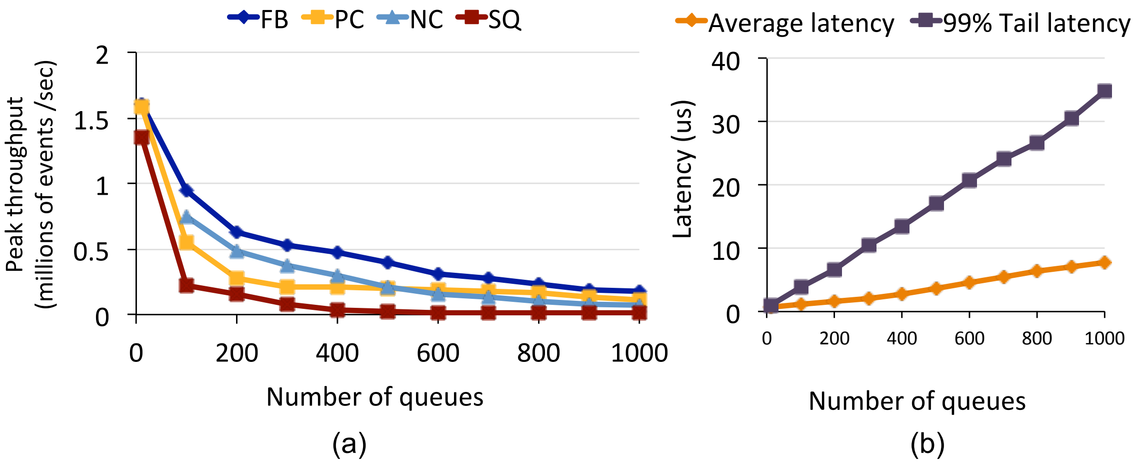


Figure 5.2: (a) Peak throughput when spin-polling many queues: Throughput reduces due to empty queue checks. (b) Latency when polling many queues: Long loop traversals dominate.

Non-proportionally Concentrated (NC), where traffic passes through 100 queues all the time and through the rest with a probability of 5% (represents a few active queues at a given time); *Single Queue (SQ)*, where traffic passes through one queue all the time and through the rest with a probability of 5% (represents a "one hot" queue).

In Fig. 5.2(a), we display the peak saturation throughput achieved with an increasing number of queues for the four different traffic shapes.

We make four observations. (1) We note a steep throughput decline with SQ traffic. This decline is caused by wasted spinning on empty queues, i.e., for each work item, the microservice performs unnecessary additional work by querying each queue head in its entire loop. (2) Although NC is a variant of SQ, the throughput drop with NC is milder since the ratio of non-empty to empty queues grows at a smaller rate compared to SQ. (3) With FB and PC, the ratio of non-empty to empty queues is constant (i.e., zero and four, respectively). Therefore, the throughput decline is less severe than NC and SQ. Nonetheless, FB and PC also face non-trivial throughput degradation due to additional queue head checks. (4) FB achieves a higher peak throughput than PC since the consumer has a higher probability of finding a non-empty queue in each spin. We conclude that microservice throughput is adversely affected when traffic is concentrated in a few queues, and the rest are usually empty, which is the common case [160].

Latency. We show how latency is affected by increasing the number of queues in Fig. 5.2(b). To avoid reporting queuing delays that show up at high load, we offer minimal load (<10% of saturation) in this test. Hence, the reported latency includes only the time it takes the spinning core to identify a non-empty queue and pick up a work item. We use the traffic pattern that achieves the best saturation throughput—FB.

We note that spin-polling indeed achieves low latency, but only with a few (<$\sim$10) I/O queues. The average and $99^{th}\%$ tail latency grow almost linearly with queue count as the time taken to interrogate non-empty queues begins to dominate. Tail latency grows with a higher slope than the median case, illustrating the worst-case scenario of a consumer having to poll through almost the entire loop before finding work in a ready queue. Average and tail latencies of tens of microseconds can cause pathologically poor performance in microsecond-scale microservices [429]. We conclude that spin-polling does not scale well with an increasing number of I/O queues.

5.1.4 MWAIT

Intel x86 MWAIT and ARM WFE instruction variants halt CPU execution until the contents of a single memory address or address range change. These instructions are currently usable only in the kernel code in server-class processors. Whereas `MWAIT` is more energy-efficient than spin polling, we find that `MWAIT` still falls short for the following reasons.

First, a microservice must be able to monitor all I/O queues simultaneously. `MWAIT` allows monitoring only a single queue at any given time. Second, a microservice might often have to service events based on pre-defined system priorities (e.g., receiving NIC items from an ads service might be less critical than receiving items from an indexing service). MWAIT variants do not allow receiving events based on explicit event priorities. Third, `MWAIT` cannot be executed in user-mode on current server-class processors (although `UMWAIT`, a user-mode MWAIT instruction has been announced, it is energy-optimized and is currently available only in embedded CPUs [113]). Due to the kernel-mode operation, we find `MWAIT` latency on par with interrupts' context switch penalty. Fourth, although user-mode `MWAIT` is available on a discontinued line of Intel's server-class CPUs [55], we find that this `MWAIT` variant still incurs a single-digit microsecond latency penalty as it is optimized for energy rather than performance efficiency [35].

In summary, we find that commonly-used notification mechanisms are not efficient when O(100) microservices interact in complex ways. Hence, prior conclusions on I/O notification must be revisited in the context of hyperscale microservices.

5.2 The μNotify Paradigm

Our characterization's main takeaways are that widely-used notification paradigms (1) do not scale well when monitoring tens to hundreds of I/O queues and (2) precipitate expensive OS-induced latencies that dominate microservice latencies. Hence, there is a critical need to re-design I/O notification for the microservice regime. We propose a novel performance-efficient I/O notification paradigm called *µNotify* that achieves both scalability and low latency. We now describe *µNotify*'s design goals and introduce its components.

5.2.1 Design Goals

Prior work [193, 389] proposed dedicating a core or a hardware unit [265] to manage notifications, since fast notifications are critical for modern, low-latency service paradigms such as microservices and serverless. We adopt this perspective to re-imagine the software and hardware design of a CPU core dedicated for monitoring, prioritizing, and receiving I/O. Such a design must achieve three goals.

First, to support microservices efficiently, we require a notification paradigm that bypasses the OS, since even a single context switch significantly degrades microservice performance. Our goal is to design a user-mode notification paradigm that achieves both

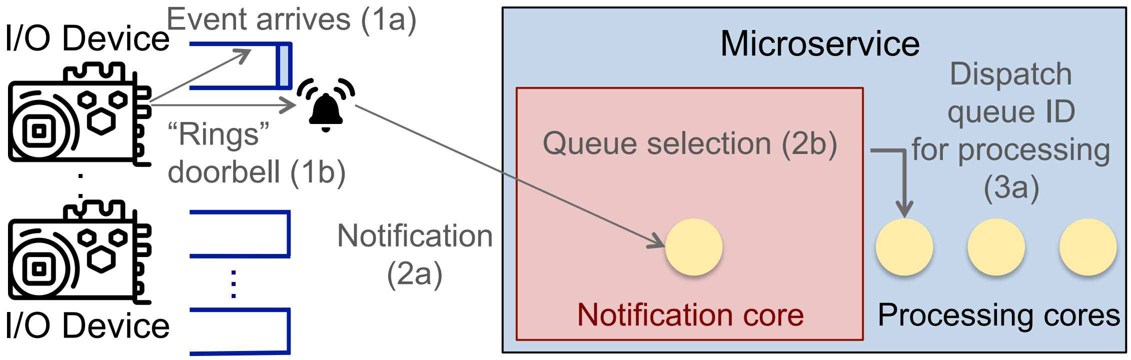


Figure 5.3: High-level system model.

low latency and high throughput.

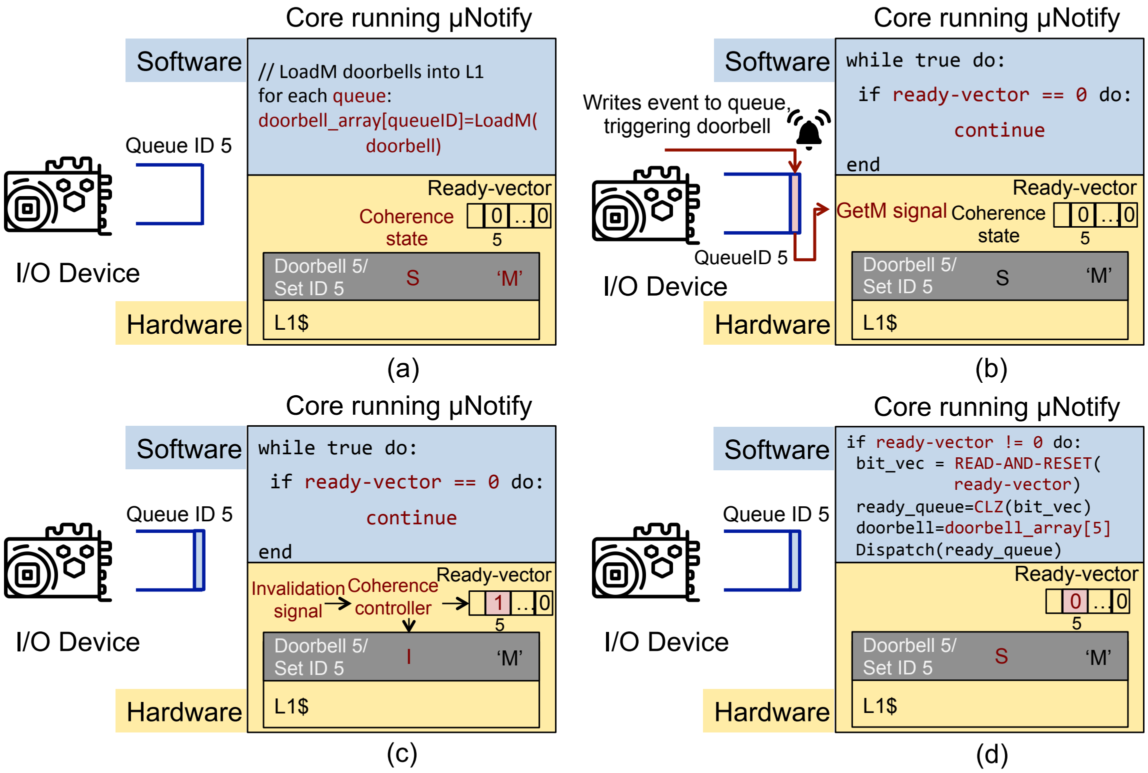
Second, we require a notification mechanism that does not iterate over empty queues to find work in ready ones, unlike conventional spin-polling. Our mechanism must be able to scale well in the presence of a large number of I/O queues.

Third, our design must be able to efficiently prioritize across numerous queues to determine which ready queue is most important. Distinguishing a high-priority queue from a low-priority one enables processing work items from latency-critical tasks before throughput-oriented batch jobs [368, 369].

5.2.2 System Model

We present our design in the context of a microservice system model shown in Fig. 5.3. A microservice's dedicated "notification core" waits for events on tens to hundreds of I/O queues. Each queue typically has a well-known memory location that indicates the arrival of new work in the queue (e.g., a queue's tail pointer). We refer to these well-known locations as *doorbells*. When a work item enters a queue (1a), the corresponding doorbell is "rung" (1b). The notification core must identify doorbell triggers (2a) and select the next ready queue to process based on a pre-defined policy (2b).

Since we dedicate a core for notification, i.e., a "notification core", received I/O items are not processed in-place, and are dispatched to another core for processing, i.e., a "processing



(a)

```
// LoadM doorbells into L1
for each queue:
doorbell_array[queueID]=LoadM(
    doorbell)
```

(b)

```
while true do:
  if ready-vector == 0 do:
    continue
end
```

(c)

```
while true do:
  if ready-vector == 0 do:
    continue
end
```

(d)

```
if ready-vector != 0 do:
 bit_vec = READ-AND-RESET(
              ready-vector)
 ready_queue=CLZ(bit_vec)
 doorbell=doorbell_array[5]
 Dispatch(ready_queue)
```

Figure 5.4: High-level diagram of μ*Notify*'s operations.

core" (3a). The notification core can use a lock-free shared buffer to tell the processing cores that there is work in a specific I/O queue. The processing cores can spin-poll as they must wait for events on a single buffer, rather than hundreds of queues. In contrast, the notification core must monitor hundreds of I/O queues simultaneously, and service them based on a predefined policy. Hence, the notification core cannot spin-poll as it would be highly unscalable as demonstrated in Sec. 5.1.

5.2.3 μNotify Overview

To meet these design goals, we propose a novel low-latency notification paradigm called μ*Notify* that facilitates notification and prioritization across hundreds of I/O queues. μ*Notify*'s key insight is to observe writes to queues by tracking hardware-generated cache line invalidation coherence signals triggered by an I/O device ringing its doorbell. Detecting invalidations serves as low-overhead notification.

μNotify's design is inspired by invalidation-driven mechanisms that manage critical section access in multi-threaded programs (e.g., Hardware Transactional Memory (HTM) [240] or Thread-Level Speculation [356, 481]). However, unlike these mechanisms [240, 356, 481] that abort execution upon receiving an invalidation, *μNotify* performs useful work by picking up a new I/O event. We first provide *μNotify*'s high-level overview and then detail its individual components. Throughout, we compare design similarities to existing Intel HTM TSX extensions [497] to demonstrate our design's implementation feasibility.

Fig. 5.4 illustrates *μNotify*'s operation. *μNotify* is composed of a front-end software programming model and a back-end hardware notification subsystem, i.e., a core dedicated for notification. First, the front-end loads the set of queue doorbells to monitor into its L1 (private) cache (5.4(a)) using a new load instruction—`LoadM`. `LoadM` loads an address and sets a "monitor" bit in the corresponding cache line. Hence, a coherence read transaction (e.g., GetS) is issued to ensure the cache line containing the doorbell has no owner and writes cannot be performed locally. For example, with a MESI coherence protocol, the doorbell's line enters a "shared" state.

`LoadM` is similar to existing line locking instructions [72] in that it sets a "monitor" bit in each doorbell's cache line. A monitor bit behaves like the "locked" bit used in prior works [356, 174, 170]—a set bit indicates that the line cannot be evicted due to replacement, but can be invalidated upon an external write, to maintain coherence. With such invalidations, the "monitor" bit remains set even though the line's coherence state is "invalid". If replacement is necessary, the cache controller will select a line not marked "monitor". If no evictable line is found, the core will access lower cache levels/memory. `LoadM` also sets the line's "monitor" bit in the directory to avoid directory-induced evictions.

When work is written to a monitored queue and its doorbell "rung", it generates a write transaction (e.g., GetM) to the doorbell's line (5.4(b)). With an invalidation coherence protocol [441], *μNotify*'s local copy of the doorbell's line must be invalidated. We extend the back-end coherence controller to atomically record write transactions generated by the

I/O device to the monitored doorbells, i.e., invalidation signals, in a special hardware register we introduce (5.4(c)). This step is similar to Intel's TSX [497] observing invalidations sent to monitored lines and recording the invalidation's reason (e.g., conflicting access, eviction, etc) in an abort register [206].

The extended controller uses the doorbell's cache set ID to index into this special bit-readable/writable hardware register bit-vector, henceforth referred to as "ready-vector" since it stores information about ready queues. The ready-vector's width is matched to the L1 private cache's size, one bit per set. With a doorbell assigned to an L1 set, the ready-vector's width corresponds to the number of distinct queues that can be monitored without aliasing. We discuss aliasing effects in greater detail when describing the back-end's operation.

μNotify's front-end spin-polls on the ready-vector's contents. Since polling has a low latency when spinning on a single location (see Sec. 5.1), *μNotify* does not face the scalability issues with polling numerous queues. After the cache controller sets the corresponding ready-vector's bit and invalidates the doorbell's line in *μNotify*'s L1, the front-end uses a proposed atomic `READ-AND-RESET` instruction to atomically read the ready-vector and reset it. It then uses existing instructions [40] to identify which bits were set in the ready-vector (i.e., which queue(s) are ready with new work) in near-constant time (5.4(d)). For each "set" bit, the front-end re-loads the "monitored" doorbell to re-arm write monitoring. It then dispatches the ready queue's ID to a processing core and selects the next ready queue to dispatch according to a pre-determined service policy. The front-end effectively functions as a task scheduler at non-trivial loads, sorting the ready queues' order of service.

5.2.4 μNotify's Back-end Microarchitecture

The back-end's operation is orchestrated by two small hardware extensions: an extended cache coherence controller and a ready-vector.

Extended coherence controller. The extended coherence controller detects work arrival by recording coherence write transactions, i.e., invalidation signals to monitored L1 doorbell

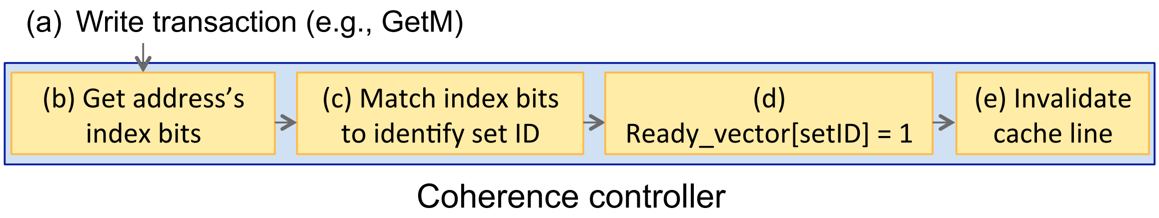

(a) Write transaction (e.g., GetM)
(b) Get address's index bits
(c) Match index bits to identify set ID
(d) Ready_vector[setID] = 1
(e) Invalidate cache line
Coherence controller

Figure 5.5: High-level cache coherence controller with *µNotify*.

lines. A coherence transaction that grants exclusive ownership of the monitored doorbell line to the requester causes the controller to indicate an arrival on the corresponding I/O queue (e.g., GetM transactions in generic protocols [441]) as shown in Fig. 5.5(a). *µNotify* requires I/O devices to provide sufficient control of mapping doorbell addresses so that the OS can use suitable placement schemes [372] to enforce mapping each doorbell to an individual L1 set (similar to prior work [205, 328, 337]). The controller can then look up the write transaction address's index bits (Fig. 5.5(b)) to identify which cache set it belongs to, i.e., which queue's doorbell (Fig. 5.5(c)).

After identifying the doorbell, the controller atomically (1) activates the associated doorbell's bit in the ready-vector (Fig. 5.5(d)) and (2) invalidates the corresponding cache entry (Fig. 5.5(e)). Atomicity between these two operations in real hardware can be established the same way as with TSX [497], where TSX atomically (1) observes an invalidation, (2) records the abort's reason in a register, and (3) invalidates the line.

The front-end adds a new I/O queue via a `LoadM`. An entry may later be removed by resetting its line's monitor bit via another instruction, `ResetM`. The controller's extension is independent of the coherence organization and can record invalidations snooped from the bus or directory.

Ready-vector. Our proposed ready-vector has a "set" bit for each cache set containing a doorbell that was written to. The ready-vector's width is matched to the L1's size, one bit per L1 set, determining the number of distinct queues that can be monitored without aliasing. For example, with a 32KB L1 direct-mapped cache (64B cache line), the ready-vector will have eight 64-bit words and can track 512 I/O queues. In this case, the ready-vector can itself fit within a single cache line.

μNotify supports two features to improve queue scalability. First, if the CPU supports an L2 private cache, the ready-vector's width can be matched to the larger L2's size (with inclusive caching) or the combined L1 and L2 sizes (with non-inclusive caching). Second, *μNotify* can afford to alias a few doorbells to an L1's single line (for a direct-mapped cache) or single set (for a set-associative cache). Even if two doorbells alias to the same line, we can monitor O(1000) doorbells with a 32KB direct-mapped L1. The front-end must then check two queues' status per event, which still has a lower overhead than spin-polling hundreds of queues (in Sec. 5.1, we show that spin-polling <10 queue heads has no noticeable performance difference). Since microservices require monitoring only hundreds of doorbells, aliasing can improve scalability.

The ready-vector also helps continue execution across context switches. A context switch "disarms" all monitored doorbells, as the L1 may be re-written. After *μNotify* is rescheduled, it must re-issue `LoadM`s to the doorbells to re-arm queue monitoring. A reserved bit in the ready-vector communicates the context switch. Before descheduling *μNotify*, the OS sets this reserved bit to indicate a context switch. When *μNotify* is re-scheduled, it checks this reserved bit to see if it must re-load doorbells for monitoring. We save and restore ready-vector state across context switches so that *μNotify* can process pending writes that occurred just before it was descheduled.

5.2.5 μNotify's Front-end Programming Model

Alg. 1 shows a simplified view of the front-end programming model. During initialization, the front-end issues a `LoadM` to each doorbell. Hence, a coherence read transaction (e.g., GetS) is issued to the "monitored" line to ensure it has no owner and writes cannot be performed locally (lines 1-5). The front-end then starts checking for ready queues by spinning on the ready-vector's contents (lines 6-11). When a ready-vector's bit is set (lines 12-13), the front-end performs the following operations.

Algorithm 1: A simplified view of *μNotify*'s programming model, assuming strict queue priorities.

```
1  // Initialization phase: Load & "monitor" doorbells into the L1 cache.
2  for all queues do
3      doorbell = allocate(doorbell_address_range);
4      doorbell_array[queue_ID] = LoadM(doorbell);
5  end
6  // Monitoring phase: Continuously track "rung" doorbells.
7  while true do
8      // If no event has occurred, continue monitoring.
9      if ready_vector == 0 then
10         continue
11     end
12     else
13         // Control enters here when a doorbell was rung.
14         /* Atomically read and reset ready_vector to re-arm notification, preventing races and
             missed writes.*/
15         bit_vector = READ-AND-RESET (ready_vector);
16         while bit_vector != 0 do
17             /* Identify ready queues: Count leading zeroes, i.e., first bit set in bit_vector.*/
18             ready_queue = CLZ (bit_vector);
19             /* Load doorbell into L1; a write between ready_vector reset & doorbell load enters a
                 known ready queue.*/
20             doorbell = doorbell_array[ready_queue];
21             bit_vector[ready_queue] = 0;
22             // Dispatch ready queue to another core.
23             Process(ready_queue);
24         end
25     end
26 end
```

Re-arm notification. When a ready-vector bit is "set", the front-end re-arms notifications to avoid data races. We propose an atomic READ-AND-RESET instruction to atomically read the ready-vector (into a local bit-vector) and reset it (lines 14-15).

Find ready queues. The front-end uses an existing instruction to find the indices of the "set" bits in the bit-vector, i.e., the ready queue IDs. In our implementation, we repeatedly use the Count Leading Zeroes (CLZ) instruction [40] to count the number of leading zeroes in the bit-vector and return the first set bit's position in near-constant time (lines 17-18). We reset the bit-vector's corresponding bit each time. CLZ is cheaper than an iterative design that loops through the bit-vector.

Re-arm monitoring. For each ready queue identified, *μNotify* re-loads its doorbell to bring it into the L1 in the “shared” state (lines 19-20) to re-arm monitoring invalidations. It then dispatches the ready queue IDs (i.e., the cache set IDs) to a processing core (lines 22-23) that ensures the corresponding queues are non-empty, to filter spurious activations from exclusive reads, false sharing [347], or doorbell writes that do not correspond to work arrivals. If the queue contains work, the processing core drains the queue and processes each item.

Avoid race. We now discuss how we avoid data race, particularly missed writes and consequent missed activations. *μNotify* performs four key operations that must be protected from race: (1) setting a ready-vector bit, (2) invalidating a line, (3) atomically reading and resetting the ready-vector, and (4) re-loading the doorbell. The hardware ensures that (1) and (2) are atomic as described earlier. (1)-(2) and (3) are atomic as the software checks whether the ready-vector is non-zero before atomically reading its value and resetting it. (3) and (4) need not be atomic, but have to occur in the described order to maintain the key invariant—a doorbell line cannot be “shared” when its corresponding ready-vector bit is “set”. Writes occurring between (3) and (4) to an invalidated doorbell will not generate an invalidation message or set a ready-vector bit. However, we will still know about these writes as they belong to the ready queues that *μNotify* identified (line 18). When the processing core receives these ready queues, it can process the newly-arrived work as well. Hence, *μNotify* ensures that there is no time window for actual work arrivals to be missed.

Handle service priority. The front-end can efficiently implement three common service policies [367]. With a *round-robin policy*, the Queue ID (QID) selected in a round must exhibit the lowest priority in the next round. In each round, *μNotify* stores the QID it selected first. For example, in the first round, *μNotify* would store the Most Significant Bit (MSB) that was set. In the next round, *μNotify* first checks QIDs that occur before the stored QID, stores them (if set) to process later, and then checks the next set QID to give the highest priority to the bit next to the one that was selected in the previous round.

The *weighted round-robin policy* generalizes round-robin, allowing a selected queue to be serviced for many consecutive rounds. Giving different weights to queues accommodates various microservices' differentiated arrival rates and QoS requirements. *μNotify* maintains a “weight” counter for each QID. Every time the queue is serviced, *μNotify* decrements its counter. When the counter reaches zero or the QID has not received any items within a time-out interval, the priority is passed to another QID by reloading its weight counter.

The *strict priority policy* fixes queue priorities such that QIDs mapped to the ready-vector's MSBs are always prioritized over the Least Significant Bits. Using the `CLZ` instruction inherently gives the QIDs mapped to the MSBs the highest priority. Priorities can also be inverted using the Count Trailing Zeros instruction [40]. However, this policy is typically not used in real applications as it would starve low-priority queues; instead, a weighted round-robin is often used, differentiating queue priorities while avoiding starvation.

5.3 Evaluation

5.3.1 Experimental Setup

We emulate *μNotify*'s back-end by having the I/O writers set a bit in a shared memory bit-vector (representing the ready-vector). This extension allows us to perform a real system evaluation to compare against existing interrupt and spin-polling paradigms. Since the software writes to the bit-vector instead of the hardware, our emulation has a higher overhead than the hardware design. Our implementation will incur additional invalidation overhead (for the bit-vector write) and hence produce conservative results.

Emulated I/O sources running on dedicated “producer” cores generate traffic with different shapes and loads. Traffic shapes are the same as those in Sec. 5.1. We offer load in a closed-loop when measuring peak throughput and in an open-loop with Poisson inter-arrivals [172] when measuring latency. We validated that the producer is not the bottleneck.

We run our experiments on an Intel Skylake machine with two sockets, 20 cores/socket, two-way SMT, 32 KiB L1-D$ (per core), 32 KiB L1-I$ (per core), 1 MiB private L2$ (per core), and 27 MiB shared LLC (per socket), with 64B cache lines; we map doorbells to 8192 available private L2 sets. *μNotify* runs on a dedicated core. The processing threads spin-poll a single lock-free work queue looking for *μNotify*'s requests. We found that the service policy has minimal impact on performance trends, so we report results for the round-robin policy, only. We report the average of ten trials.

5.3.2 Microservices

We consider four microservices, three of which are mid-tier microservices from the μSuite [448] benchmark suite (detailed in Chapter II), LSH, McRouter, and Recommend; the fourth, Word Stemming, is constructed using the same framework.

Locality Sensitive Hashing (LSH). This microservice uses locality-sensitive hashing to exponentially reduce a high dimensional search space. Upon receiving a search request (e.g., image search), the microservice probes an in-memory LSH table to gather potential nearest neighbor candidates.

McRouter. We use μSuite's consistent hashing microservice based on Facebook's McRouter [69], which is stateless and computes a consistent hash to suitably route KV-store requests.

Word Stemming. Stemming is a normalization process to reduce words to their root and is a core query-rewriting service in web search. We develop a stemming microservice based on Oleander's design of the Porter stemming algorithm [406, 405]. Word stemming is stateless; it hard-codes all stemming paths (prefixes, suffixes, etc.) into the program control-flow.

Recommend. This microservice performs user-based collaborative filtering on a pre-composed matrix of {user, item, rating} tuples to make rating predictions.

We now evaluate *μNotify*'s throughput, queue scalability, and latency, comparing it to two state-of-the-art paradigms—spin-polling and interrupts via `epoll`.

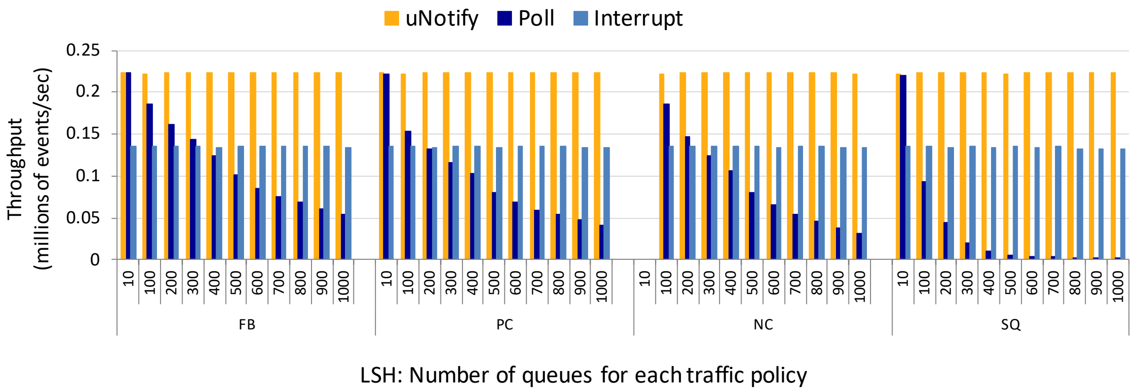


Figure 5.6: Peak throughput achieved by *μNotify* compared to spin-polling and interrupts across different queue counts and traffic shapes for LSH: *μNotify* consistently achieves a higher throughput compared to state-of-the-art paradigms.

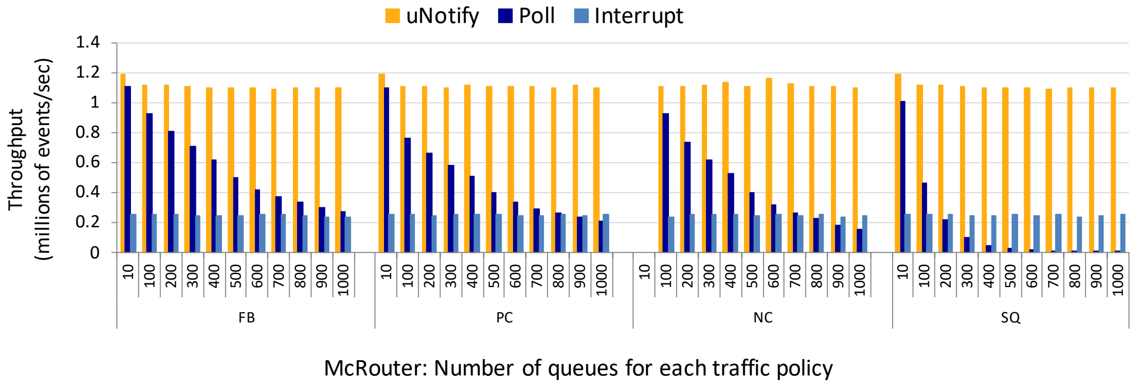


Figure 5.7: Peak throughput achieved by *μNotify* compared to spin-polling and interrupts across different queue counts and traffic shapes for McRouter: *μNotify* consistently achieves a higher throughput compared to state-of-the-art paradigms.

5.3.3 Peak Throughput

We analyze *μNotify*'s peak throughput and compare it to existing paradigms when receiving work from an increasing number of queues. Figs. 5.6, 5.7, 5.8, and 5.9 show throughput data for all microservices on the same traffic shapes used in Sec. 5.1.

Similar to the results in Sec. 5.1, we observe the most drastic throughput drop when spin-polling with SQ traffic, as the core needs to poll a larger number of empty queues to find work in the ready one(s); the drop is milder with NC. With FB and PC, the ratio of non-empty queues to empty queues is constant, so spin-polling achieves better throughput with these traffic patterns than with SQ and NC (PC is worse than NC until 500 queues as it

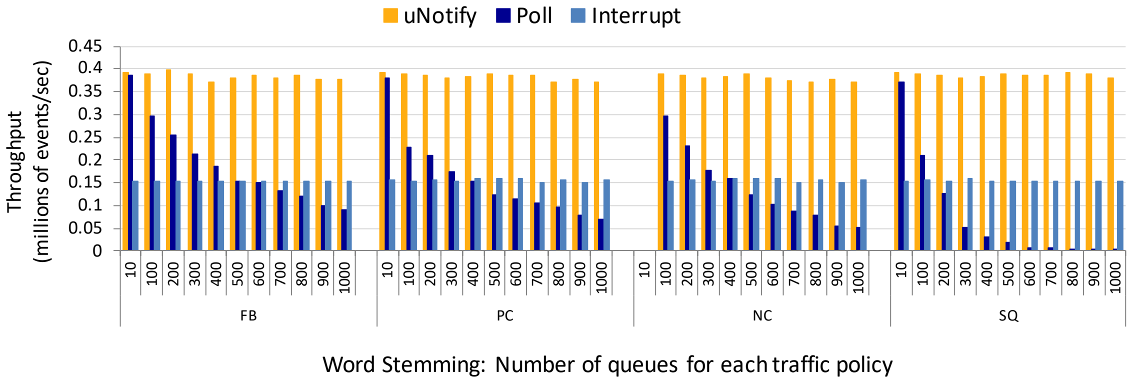


Figure 5.8: Peak throughput achieved by *µNotify* compared to spin-polling and interrupts across different queue counts and traffic shapes for Word Stemming: *µNotify* consistently achieves a higher throughput compared to state-of-the-art paradigms.

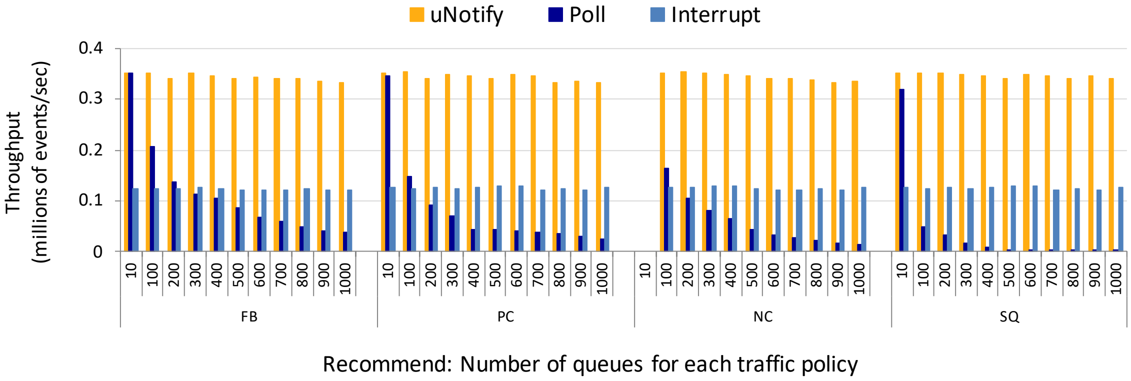


Figure 5.9: Peak throughput achieved by *µNotify* compared to spin-polling and interrupts across different queue counts and traffic shapes for Recommend: *µNotify* consistently achieves a higher throughput compared to state-of-the-art paradigms.

polls many more empty queues).

Regardless of the traffic pattern, we find that the work done per item continues to grow with an increasing number of queues (due to additional queue head polls), reducing throughput. Spin-polling achieves better throughput with FB than PC since FB performs fewer empty queue checks. Hence, in our successive experiments, we only consider FB traffic.

We note that interrupts achieve low throughput in general, since they perform more work to notify the microservice about an item arrival, i.e., they additionally execute large I/O stacks. However, throughput approximately remains constant across numerous queues, since

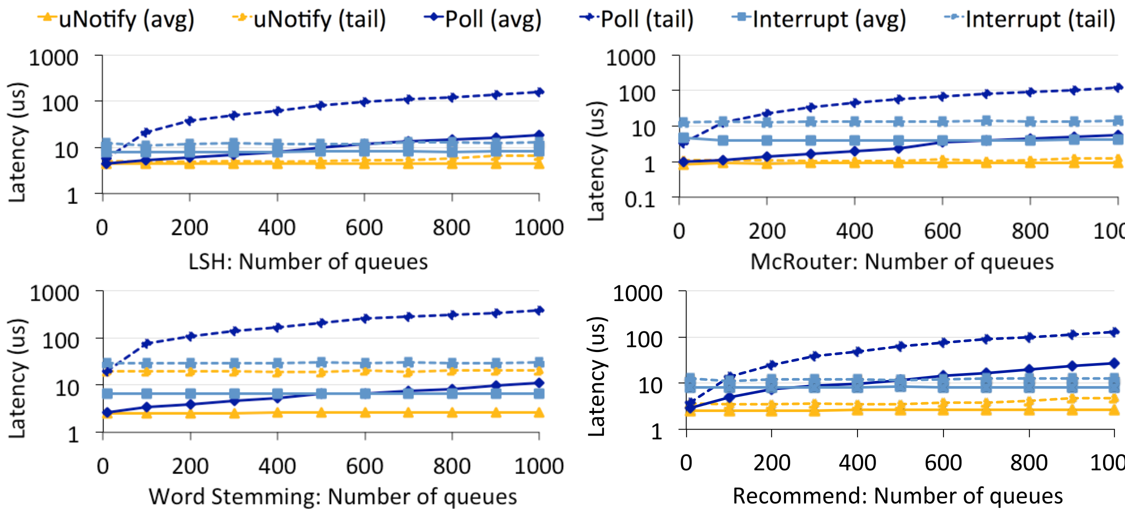


Figure 5.10: *μNotify*'s latency under light traffic with increasing queues (Y-axis is log-scale): *μNotify* achieves lower latency.

`epoll` can notify events received on many queues in a single wakeup/activation. Hence, spin-polling is less scalable than interrupts as the throughput with spin-polling eventually becomes much worse than interrupts.

In contrast, *μNotify* avoids the useless work of interrogating empty queues and does not execute I/O stacks. It recovers the throughput loss of spin-polling's empty queue checks and interrupts' execution of deep I/O stacks. Particularly, with the SQ and NC traffic, where spin-polling falls apart with more queues, *μNotify* maintains peak throughput even with 1000 queues. With various traffic shapes and queue counts, *μNotify* improves peak throughput by 15.63x, on average, compared to spin-polling and 2.815x, on average, compared to interrupts.

5.3.4 Queue Scalability

In Fig. 5.10, we report *μNotify*'s low-load latency across microservices as a function of increasing queue count. We offer a light FB traffic load (<10%) to prevent queuing delays. Note that the Y-axis is a log-scale to capture the order-of-magnitude range in measured latencies.

We make the following observations. First, with spin-polling, both the average and tail latencies grow linearly as the queue count is increased, since the core has to check more empty queues. Long poll-loop traversal delays severely degrade the tail latency in particular, since the iterator code has to traverse almost all queues before it reaches the ready one.

Second, the tail latency with spin-polling is particularly exacerbated when the microservice has greater request processing time variability (e.g., `Word Stemming`). `Word Stemming`'s tail latency degrades even further due to Head-of-Line (HoL) blocking——when the work item at the head of a queue takes longer than average to process, all items behind it experience long queuing delays, precipitating high tail latency.

Third, interrupts exhibit higher latency than *μNotify* due to the additional context switch and IO stack overhead. Nonetheless, interrupt latencies remain unchanged with increasing queues, outperforming spin-polling at higher queue counts.

Fourth, *μNotify* avoids the latency of both checking empty queues and executing I/O stacks. *μNotify* scales better across increasing queue counts, and the latency is unaffected with more queues. It retains both average and tail latency below 10 μs even at 1000 queues (except `Word Stemming` which has greater processing time variation), while spin-polling causes a tail latency of more than 100 μs for large queue counts. *μNotify* improves the tail latency by 14.2x and 2.72x, on average, compared to spin-polling and interrupts respectively.

5.3.5 Median and Tail Latency

In Fig. 5.11, we show *μNotify*'s latency across various FB loads, with a hundred queues. We note that: (1) Spin-polling has lower average latency than interrupts as the probability of finding a non-empty queue is higher in the average case. In contrast, polling exhibits higher tail latency due to worst-case traversal delays. (2) Interrupts have a slightly higher tail latency at low load due to longer thread wakeups, caused by OS scheduler actions and low-power mode transitions. (3) Interrupts continue to perform worse than *μNotify* due to IO stack overheads. Hence, apart from achieving a lower average and tail latency than both spin-polling and interrupts, *μNotify* is also able to sustain higher load, i.e., closer to

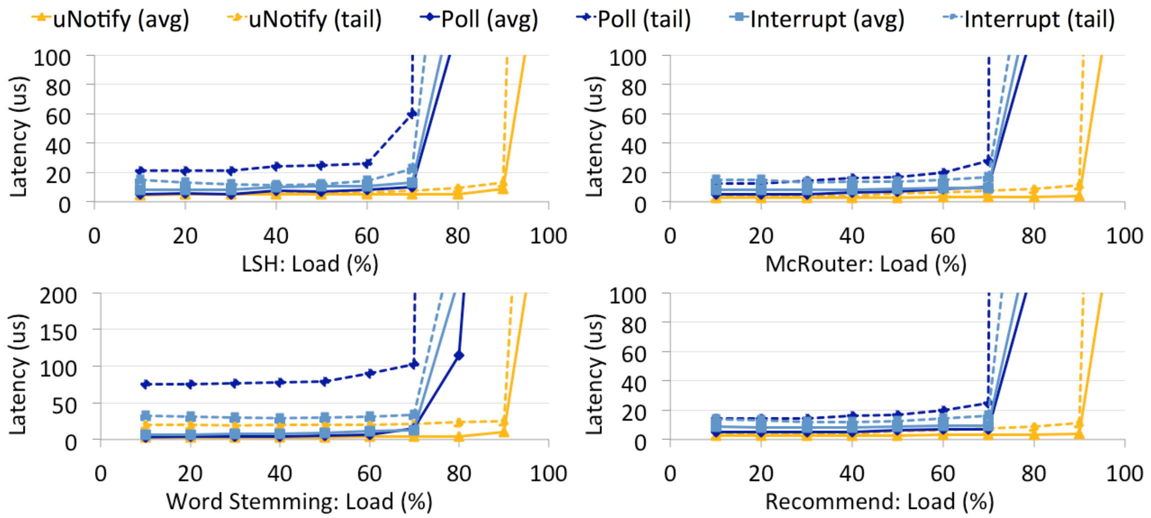


Figure 5.11: *µNotify*'s latency with increasing load compared to state-of-the-art: *µNotify* sustains higher load with low latency.

saturation, while the other techniques face unbounded queuing delays between 70 - 90% of saturation.

5.3.6 Overheads

µNotify incurs a 150ns-500ns overhead across various loads and queue counts due to (1) executing the CLZ instruction and (2) a few spurious doorbell checks. Our only hardware overhead comes from inserting a small logic in the cache controller and a hardware register whose width maps to the L1's size. These changes are trivial and do not add much complexity to our hardware design.

5.4 Discussion

We discuss designs that did not work as expected, lessons learned from the process, and *µNotify*'s limitations.

Sub-par designs. To identify coherence invalidations, we initially tried invalidation-driven techniques that manage critical section access amongst threads (e.g., Hardware Transactional Memory (HTM) [240] or Thread-Level Speculation [356, 481]). HTM was particularly promising, as it is available in existing CPUs. Our idea was to process received I/O when a critical section (i.e., queues) was written to, instead of “aborting” like these works [240, 356].

We designed a core to spin-poll on doorbells within an HTM transaction. When a doorbell is invalidated, i.e., “rung”, the spinning thread enters an abort handler where we could process new work. At first blush, it may seem that using available HTM extensions (e.g., TSX [240]) is an even better approach, but we identified several drawbacks. (1) When reading events within the abort handler or loading doorbells into the transaction each time, the service would miss new writes. Capturing these writes by checking the doorbells within the transaction would still precipitate the worst-case poll-loop latency (e.g., a single “hot” queue). (2) Existing HTM extensions also do not make the written address visible to the abort handler.

Another idea was to extend I/O device drivers to set a specific bit in a shared memory bit-vector for each write. But, sharing a bit-vector amongst multiple drivers (1) does not allow a service to control a third-party driver’s writes, (2) induces expensive false sharing [347], and (3) raises security/isolation concerns as devices/micro-servers may not trust each other.

Wimpy cores. Since *µNotify*’s primary purpose is to just track invalidations and a special register’s contents, we do not require a full-fledged power-hungry server-class core. Instead, a data center operator could dedicate a wimpy core in a heterogeneous CPU [346] to *µNotify* to optimize energy.

Software Data Plane (SDP). SDPs handle O(1000) queues when interacting with I/O devices and clients. *µNotify* can replace SDP’s poll-driven I/O [238] to improve performance.

5.5 Related Work

Memory monitoring. Several memory monitoring proposals for reliability and security [194, 379, 456, 473, 503] are not readily usable for microservices. We consider a general purpose design for a more detailed comparison: ECMon [379] can monitor various cache events (e.g., invalidation) to different address ranges, specified in entries of an event descriptor table. This table is a small structure that cannot efficiently support hundreds to thousands of events from various doorbells. Moreover, these proposals only monitor memory locations and do not offer solutions to establish priority among ready events.

Several queue-based locking schemes [364, 187, 349, 282] avoid spinning on a single lock by forming a FIFO queue of the requesting cores. In contrast, *μNotify* services numerous I/O queues based on defined policies rather than the FIFO order. Microservices must often sustain a high request arrival rate; the notification mechanism must be able to schedule requests at non-trivial loads, prioritizing queues' service order.

Interrupts. Prior works [231, 211, 316, 490] propose adapting between spin-polling and interrupts, user-level interrupts [170, 183, 377, 381, 41], and low-overhead interrupt solutions [257, 434]. In contrast, *μNotify* is a simple paradigm, achieves significantly lower latency, and prioritizes queues.

I/O software stacks. Kernel-bypass software stacks enable user processes to directly communicate with I/O. IX [154], Arrakis [402], ZygOS [408], Andromeda [195], and Snap [357] are specialized networking data planes with features such as task stealing [408], task preemption [281], virtualization [195, 402], while ReFlex [304] and PASTE [258] target storage devices. Demikernel [498] specifies I/O abstractions for a library OS. Systems such as Snap [357] and Shenango [389] deploy centralized software to orchestrate I/O. *μNotify*, as a notification paradigm, can benefit transport software implementations, especially with SDPs [238] and microkernel systems [357, 389].

5.6 Chapter Summary

We summarize our contributions as follows:

- **A comprehensive analysis of state-of-the-art I/O event notification paradigms.**

We presented a detailed characterization of the state-of-the-art I/O event notification paradigms used in real systems in the context of modern hyperscale microservices. This characterization paved the way for redesigning I/O event notification paradigms for the microservice regime.

- μ**Notify**: We presented a low-overhead, hardware-assisted I/O notification paradigm that capitalizes on cache coherence messages generated by commodity processors. μ*Notify* exemplifies how existing hardware mechanisms can intelligently be used to overcome new overheads (particularly from I/O notification) that are faced in the hyperscale microservice regime.

- **Near-constant time I/O notification.** We introduced a small hardware enhancement that enables μ*Notify* to convey information about ready I/O queues in near-constant time. Hence, μ*Notify* achieves both low-overhead I/O notification and scalability across numerous I/O queues.

- **A demonstration of μNotify's efficacy.** We presented an evaluation demonstrating μ*Notify*'s efficacy at improving I/O notification performance for modern microservices.

In Chapter IV, we identified that I/O event notification can critically affect microservice performance. (We will further detail I/O-induced overheads in Chapter VI). In this chapter, we first investigated why widely-used I/O event notification paradigms, such as interrupt, spin-polling, and MWAIT fall short in the microservice regime. We found that existing I/O notification paradigms suffer from expensive OS-induced overheads and lack scalability across numerous I/O queues.

To overcome these challenges, we presented a simple solution, μ*Notify*, that achieves low I/O notification latency, I/O queue scalability, and service priority. μ*Notify* is a hardware-assisted, shared-memory I/O event notification paradigm that facilitates scalable, performance-efficient communication with numerous I/O queues. μ*Notify*'s key idea is to observe cache coherence invalidation messages and use them as low-overhead I/O notification. We introduced a minor hardware modification (in the form of an extension to the coherence controller and a special hardware register) that allows *μNotify* to notify, prioritize, and service I/O events from many ready I/O queues in near-constant time. Finally, we demonstrated that *μNotify* improves microservice peak throughput by 15.63x and tail latency by 14.2x compared to state-of-the-art I/O notification techniques.

CHAPTER VI

Understanding Hardware Customization Opportunities at Hyperscale

At hyperscale, the microservice deployment model uses standardized RPC interfaces [45] to invoke several microservices to serve a user's query. Hence, upon receiving an RPC, a microservice must often perform operations such as I/O processing, decompression, deserialization, and decryption, before it can execute its core functionality (e.g., key-value serving, ML inference).

At global user population scale, important microservices can grow to account for an enormous installed base of physical hardware. As described in Chapter IV, across Facebook's global server fleet, seven key microservices in four diverse service domains run on hundreds of thousands of servers and occupy a large portion of the compute-optimized installed base. With the decline of hardware performance scaling [217, 462], successive server generations running these microservices exhibit diminishing performance returns.

To improve hardware efficiency, several architects today work on developing numerous specialized hardware accelerators for important microservice domains (e.g. Machine Learning tasks). However, hyperscale enterprises have strong economic incentives to limit hardware platform diversity to (1) maintain fungibility of hardware resources, (2) preserve procurement advantages that arise from economies of scale, and (3) limit the overhead of developing and testing on myriad specialized hardware platforms. Hence, an important question arises: *which microservice operations consume the most CPU cycles and are worth accelerating? Are there common overheads across microservices that we might address when designing future hardware?*

To answer this question, as a part of this dissertation's hardware contributions, we[1] undertake a comprehensive characterization of microservices' CPU overheads on Facebook production systems serving live traffic. Very few prior works study how CPU cycles are spent in data centers. Kanev et al. [285] investigate the "data center tax" across Google's server fleet by studying cycles spent in seven types of *leaf functions* invoked at the end of a call trace (e.g., `memcpy`()). However, a leaf function study alone does not holistically provide insight into whether acceleration might improve a *microservice functionality* (e.g., encryption).

To analyze *microservice functionalities*, we must comprehensively characterize a microservice's entire call stack to measure the CPU cycles spent in each phase of the microservice's operation after it receives a request. Characterizing microservice functionalities helps determine (1) whether diverse microservices execute common types of operations (e.g., compression, serialization, and encryption) and (2) the overheads such operations induce. Analyzing *both* leaf functions and microservice functionalities helps identify important acceleration opportunities that might inform future software and hardware designs. To this end, we characterize the CPU cycles spent by Facebook's production microservices in both *leaf functions* and *microservice functionalities*.

Our comprehensive characterization reveals that application logic disaggregation across microservices at hyperscale has resulted in significant leaf function and microservice functionality overheads. For example, several microservices spend only a small fraction of their execution time serving their main application logic (e.g., ML inference), squandering significant cycles facilitating the main logic via *orchestration* work that is not core to the main

[1] Some of the work in this chapter was performed in collaboration with a researcher at Facebook, Abhishek Dhanotia [444, 445]. Therefore, I use the "we" pronoun in this chapter to acknowledge Dhanotia's involvement in this work.

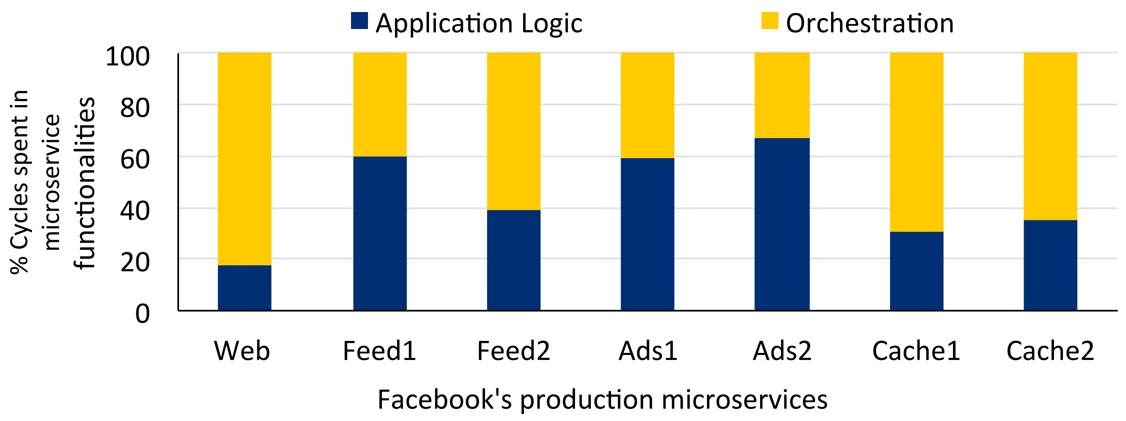


Figure 6.1: Breakdown of cycles spent in core application logic vs. orchestration work: Orchestration overheads can significantly dominate.

application logic (e.g., compression, serialization, and I/O processing), as shown in Fig.6.1. Accelerating main application logic alone can yield only limited performance gains—an important ML microservice can speed up by only 49% even if its ML inference takes no time. Facebook's `Web` microservice exhibits surprisingly high overheads from reading and updating logs. Caching microservices [171] can spend 52% of cycles sending/receiving I/O to support a high request rate and consequent I/O compression and serialization overheads dominate. Copying, allocating, and freeing memory can consume 37% of cycles, and kernel scheduler and network overheads are high with poor IPC scaling. Many microservices face common orchestration overheads despite great diversity in microservices' main application logic.

Driven by our characterization, we report acceleration opportunities that might inform future software and hardware designs. However, to build specialized hardware accelerators for key microservice operations, it is important to first identify which type of accelerator meets microservice requirements and is worth designing and deploying. Introducing hardware acceleration in production requires (1) designing new hardware, (2) testing it, and (3) carefully planning capacity to provision the hardware to match projected load.

Given the uncertainties inherent in projecting customer demand, deploying diverse custom hardware is risky at hyperscale as the hardware might under-perform due to performance bounds from the microservice's software interaction with the hardware (e.g., offload-induced overheads), resulting in high monetary losses. To make well-informed hardware decisions, it is crucial to answer the following question early in the design phase of a new hardware accelerator: *how much can the accelerator realistically improve its targeted microservice overhead?* To answer this question, there is a need for simple analytical models that identify performance bounds early in the hardware design phase to project realistic gains from accelerating microservice overheads.

We develop an analytical model for hardware acceleration, *Accelerometer*[2], that identifies performance bounds to project microservice speedup. Whereas a few prior models [127, 188] estimate speedup from acceleration, they fall short in the context of microservices as they assume that the CPU waits while the offload operates. However, for many microservice functionalities, offload may be asynchronous; the processor may continue doing useful work concurrent with the offload. We extend prior models [127, 188] to capture such concurrency-induced performance bounds to project microservice speedup from hardware acceleration.

We demonstrate *Accelerometer*'s utility using three retrospective case studies conducted on production systems serving live traffic. First, we analyze an on-chip acceleration strategy—a specialized hardware instruction for encryption, AES-NI [53]. Second, we study an off-chip accelerator—an encryption device connected to the host CPU via a PCIe link. In the final study, we analyze a remote acceleration strategy—a general-purpose CPU that solely performs ML inference and is connected to the host CPU via a commodity network. In all three studies, we show that *Accelerometer* estimates the real microservice speedup with an error that is less than or equal to 3.7%. Finally, we use *Accelerometer* to project speedup for the acceleration recommendations derived from three important common overheads identified by our characterization—compression, memory copy, and memory allocation.

The rest of this chapter is organized as follows: We describe and characterize the

[2] Available at https://github.com/akshithasriraman/Accelerometer and https://doi.org/10.5281/zenodo.3612797

production microservices in Section 6.1. We explain the *Accelerometer* analytical model in Section 6.2. We validate and apply *Accelerometer* in Section 6.3 and Section 6.4, compare against related work in Section 6.5, discuss long-term impact potential in Section 6.6, and conclude in Section 6.7[3].

6.1 Understanding Microservice Overheads

We identify how Facebook's important microservices spend their CPU cycles executing (1) leaf functions and (2) various microservice functionalities to determine software and hardware acceleration opportunities. We first characterize leaf functions (e.g., `memcpy()`) across diverse microservices. Whereas a leaf function study provides insight into common software building blocks, it does not reveal whether services share common functionalities that can be accelerated (e.g., compression). Hence, we additionally characterize service functionalities to identify common overheads that can benefit from acceleration. We also study Instructions Per Cycle (IPC) scaling for both the leaf and microservice functionality breakdowns to identify optimizations for overhead categories that scale poorly across CPU generations. In this section, we (1) describe each microservice, (2) explain our characterization approach, (3) characterize leaf functions, (4) report on microservice functionality breakdowns, and (5) summarize our characterization's key conclusions.

6.1.1 The Production Microservices

We study seven microservices in four diverse service domains that account for a large portion of Facebook's data center fleet. We study the seven Facebook microservices detailed in Chapter IV [446] (1) `Web`: a front-end microservice that implements PHP and Hack, (2) `Feed1` and `Feed2`: News Feed microservices that aggregate, rank, and display stories, (3)

[3] Fun fact: This chapter is personally a big milestone for me. It is the first work I took from start to finish without my Ph.D. advisor, Tom Wenisch. I'm grateful to Tom for continuously telling me that I could do it. If you are a graduate student struggling with imposter syndrome, I recommend that you try to independently see a project through. Your paper might not get in but, in my humble opinion, that is rather secondary. You grow to become *much* more confident about your abilities.

Table 6.1: GenA, GenB, and GenC CPU platforms' attributes.

CPU features	GenA	GenB	GenC
Micro-architecture	Intel Haswell	Intel Broadwell	Intel Skylake
Cores / socket	12	16	18 or 20
SMT	2	2	2
Cache block size	64 B	64 B	64 B
L1-I$ / core	32 KiB	32 KiB	32 KiB
L1-D$ / core	32 KiB	32 KiB	32 KiB
Private L2$ / core	256 KiB	256 KiB	1 MiB
Shared LLC	30 MiB	24 MiB	24.75 or 27 MiB

Ads1 and Ads2: advertisement microservices that compute user-specific and ad-specific data, and (4) Cache1 and Cache2: large distributed-memory object caching microservices. We characterize on production systems serving live traffic.

6.1.2 Characterization Approach

We characterize the seven microservices by profiling each in production while serving real-world user queries. We next describe the characterization methodology.

Hardware platforms. We characterize our microservices on 18- and 20-core Intel Skylake processors [212] (see Table 6.1). We run Web, Feed1, Feed2, and Ads1 on the 18-core Skylake, and Ads2, Cache1, and Cache2 on the 20-core Skylake. We study IPC scaling across three CPU generations (Table 6.1).

Experimental setup. We measure each microservice in our production environment's default deployment—stand-alone with no co-runners on bare metal hardware. There are no cross-service contention or interference effects in our data. We study each system at peak load to stress performance bottlenecks.

We characterize leaf functions by first using Strobelight [104] to measure instructions and cycles spent in microservices' key leaf functions. We then feed leaf functions and their cycle counts to an internal tool that tags each leaf function's category (e.g., tagging memcpy() as "memory"); the tool then aggregates cycles spent in each leaf category.

To characterize microservice functionality, we use Strobelight [104] to (1) collect all

Table 6.2: Categorization of leaf functions.

Leaf category	Examples of leaf functions
Memory	Memory copy, allocation, free, compare
Kernel	Task scheduling, interrupt handling, network communication, memory management
Hashing	SHA & other hash algorithms
Synchronization	User-space C++ atomics, mutex, spin locks, CAS
ZSTD	Compression, decompression
Math	Intel's MKL, AVX
SSL	Encryption, decryption
C Libraries	C/C++ search algorithms, array & string compute
Miscellaneous	Other assorted function types

function call traces of a microservice (e.g., a function call trace can be composed of a function sequence starting with cloning a thread and ending with a leaf function such as `memcpy()`) and (2) measure cycles and instructions spent in each call trace. We feed the function call traces and their cycle counts to an internal tool that buckets each function call trace into a microservice functionality category (e.g., I/O, serialization, and compression); it then aggregates cycles spent in each category. To determine a category's IPC, we determine the ratio of aggregated instruction and cycle counts for functions in that category. We contrast our measurements with some SPEC CPU2006 [253] benchmarks and Google services [285] where possible.

6.1.3 Leaf Function Characterization

We first present key leaf function breakdowns for our microservices and compare them with SPEC CPU2006 [253] and Google services [140, 285]. We then characterize a few key leaf functions in greater detail. Finally, we report IPC scaling measurements for `Cache1`'s leaf functions across three CPU generations.

We define each leaf function category in Table 6.2. We report the fraction of overall cycles spent in each leaf category in Fig. 6.2 (we omit bars that consume <1% of cycles). We also omit bars for 401.bzip2, 429.mcf, 445.gobmk, 456.hmmer, 458.sjeng, 462.libquantum, 464.h264ref, and 483.xalancbmk SPEC CPU2006 benchmarks since their leaves are

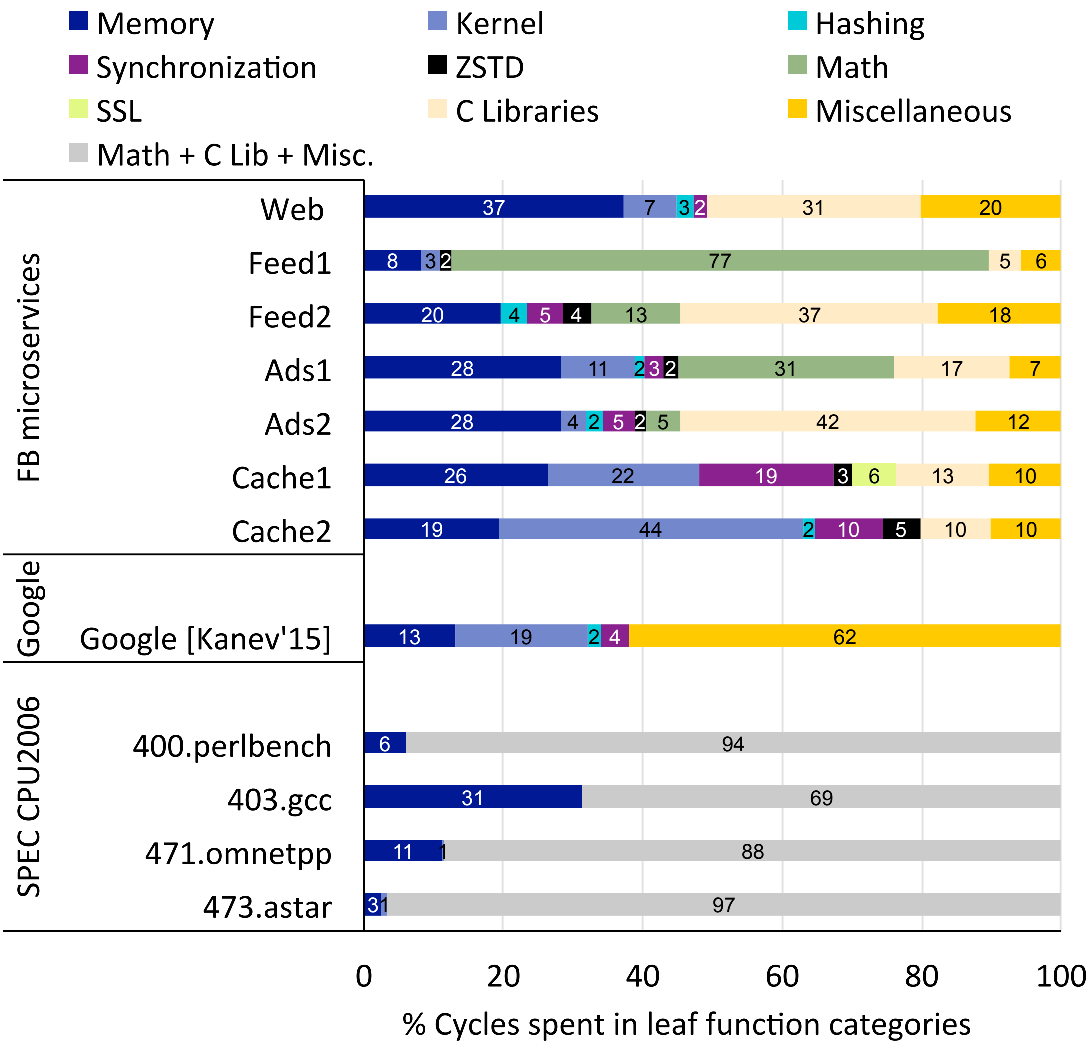


Figure 6.2: Breakdown of cycles spent in leaf functions: Memory functions consume a significant portion of total cycles.

composed of either math functions or C libraries.

We make several observations. First, most microservices spend a significant fraction of cycles on memory functions (e.g., copy and allocation) and kernel operations. Cache1 and Cache2 spend more cycles in the kernel as they frequently incur context switches due to a high service throughput [446]. We further break down the memory and kernel function categories (Subsections 6.1.3.1 and 6.1.3.2) to identify specific optimizations.

Second, ML microservices such as Ads2 and Feed2 spend only up to 13% of cycles on mathematical operations that constitute ML inference using Multilayer Perceptrons. We find that these services can also benefit from optimizations to C libraries, which we investigate further in Subsection 6.1.3.4.

Third, `Cache1` and `Cache2` spend significant cycles synchronizing frequent communication between distinct thread pools. Additionally, we find that `Cache1` spends 6% of cycles in leaf encryption functions since it encrypts a high number of Queries Per Second (QPS).

Fourth, Google's breakdown across their global server fleet [285] is similar to Facebook's leaf breakdowns. In contrast, SPEC CPU2006 [253] benchmarks do not capture key leaf overheads (e.g., memory and kernel) faced by our microservices; their functions primarily belong to the math, C libraries, and miscellaneous categories.

We conclude that many leaf function overheads are significant and common across services. We next investigate leaf functions in greater detail to identify acceleration opportunities.

6.1.3.1 Memory

In Fig. 6.3, we characterize cycles spent in various memory leaf functions as a fraction of total cycles spent in memory functions. The memory functions include memory copy, free, allocation, move, set, and compare. We compare our microservices with Google's services [285] and SPEC CPU2006 [253] benchmarks. Note that only the memory copy and allocation breakdowns are available for Google's services [285], and they account for 13% of total cycles (represented by an asterisk in Fig. 6.3).

We observe that memory copies are by far the greatest consumers of memory cycles. Google's services also spend 5% of total fleet cycles on memory copies [285]. Although `403.gcc` exhibits high memory overhead, it spends very few cycles in copying memory. Memory copy optimizations such as (1) reducing copies in network protocol stacks [114], (2) performing dense memory copies via SIMD [15], (3) moving data in DRAM [428], (4) minimizing I/O copies using Intel's I/O Acceleration Technology (IO AT) [54], (5) processing in memory [122], (6) optimizing memory-based software libraries [383, 411],

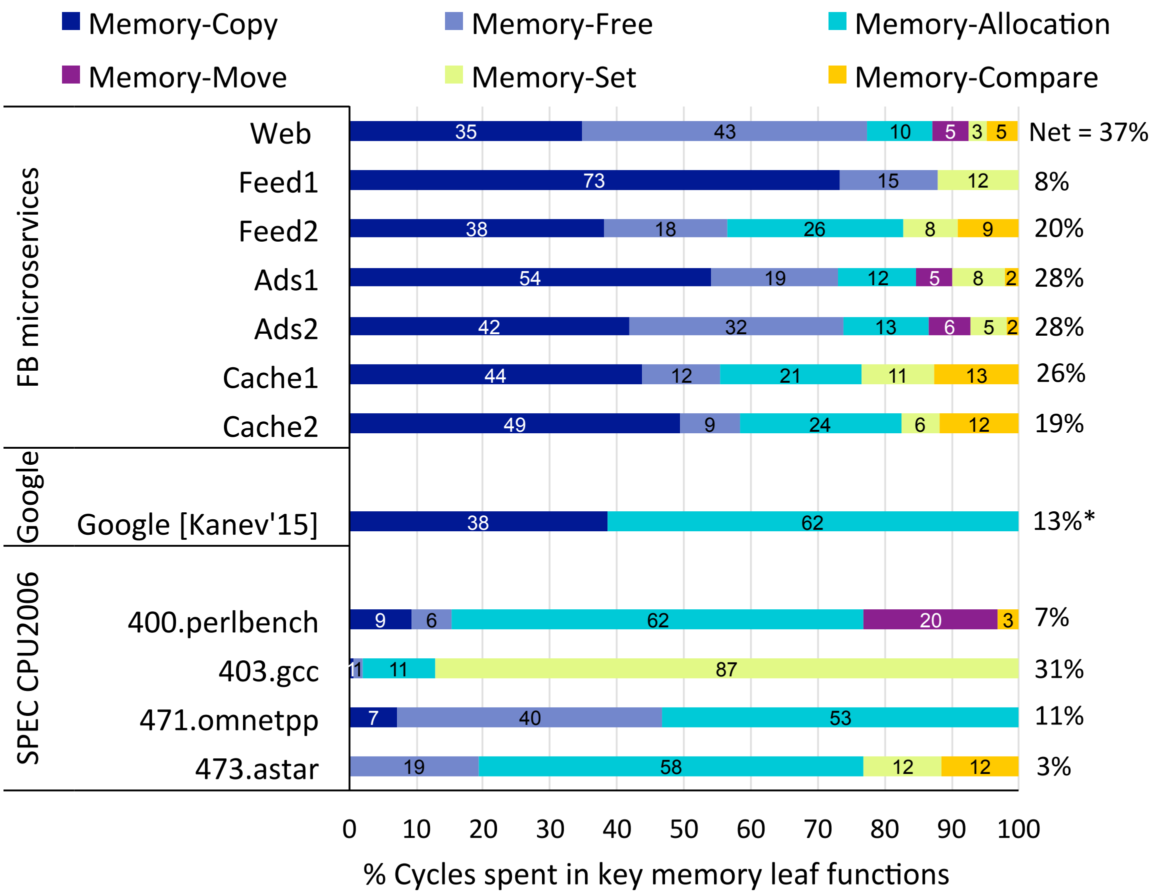


Figure 6.3: Breakdown of cycles spent in memory leaf functions as a fraction of total cycles: Memory copy, allocation, & free consume significant cycles.

and (7) building hardware accelerators (e.g., for memory-memory copies [351]) could minimize copy overheads. To identify optimizations, we also provide greater nuance to our memory copy characterization by attributing memory copies to various microservice functionalities.

We find that memory allocation can be a significant overhead despite using fast software allocation libraries [137]. Google's services incur a slightly greater allocation overhead. This observation suggests the need to continue to build software [264, 502, 305, 10, 319, 157, 366] and hardware optimizations [287, 303] for allocations. Of the SPEC CPU2006 [253] suite, `471.omnetpp` spends the most cycles on allocation ($\sim$5%).

Freeing memory incurs a high overhead for several microservices, as the `free()` func-

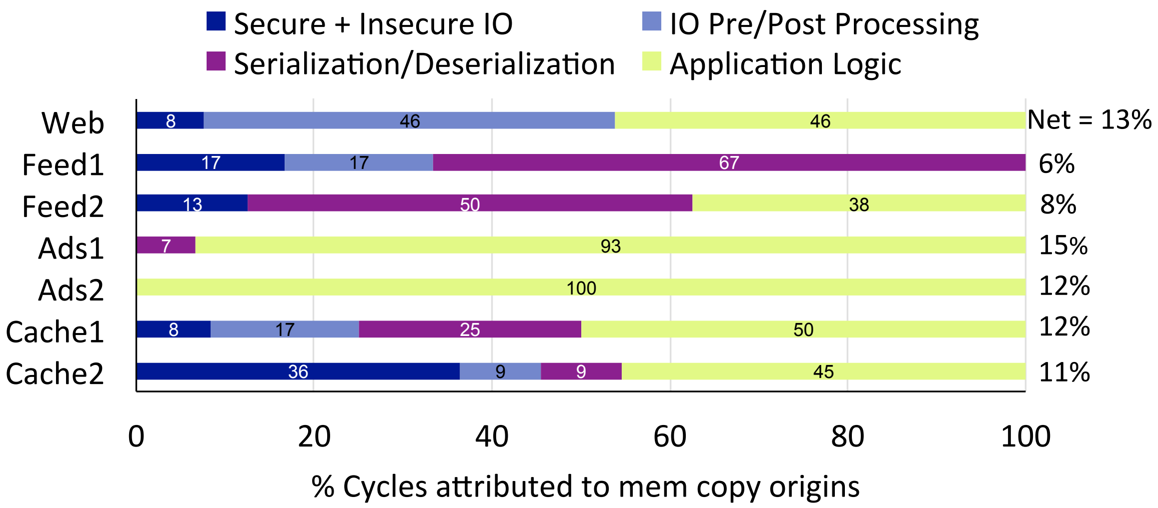


Figure 6.4: Breakdown of service functionalities that invoke memory copies: There is significant diversity in dominant functionalities that perform copies.

tion does not take a memory block size parameter, performing extra work to determine the size class to which to return the block [287]. TCMalloc performs a hash lookup to get the size class. This hash tends to cache poorly, especially in the TLB, leading to performance losses. Although C++11 ameliorates this problem by allowing compilers to invoke `delete()` with a parameter for memory block size, overheads can still arise from (1) removing pages faulted in when memory was written to and (2) merging neighboring freed blocks to produce a more valuable large free block [48]. While numerous prior works focus on optimizing allocations [285, 264, 303], very few recognize that optimizing `free()` can result in significant performance wins.

Memory copy origins. In Fig. 6.4, we attribute memory copies to microservice functionalities defined in Table 6.3. We find that memory is primarily copied during (1) I/O pre- or post-processing, (2) I/O sends and receives, (3) RPC serialization/deserialization, and (4) application logic execution (e.g., executing key-value stores in `Cache`). We observe significant diversity in dominant service functionalities that invoke copies across microservices. This diversity suggests a strategy to specialize copy optimizations to suit each microservice's distinct needs. For example, `Web` can benefit from reducing copies in I/O pre- or post-processing [428, 54], whereas `Cache2` can gain from fewer copies in

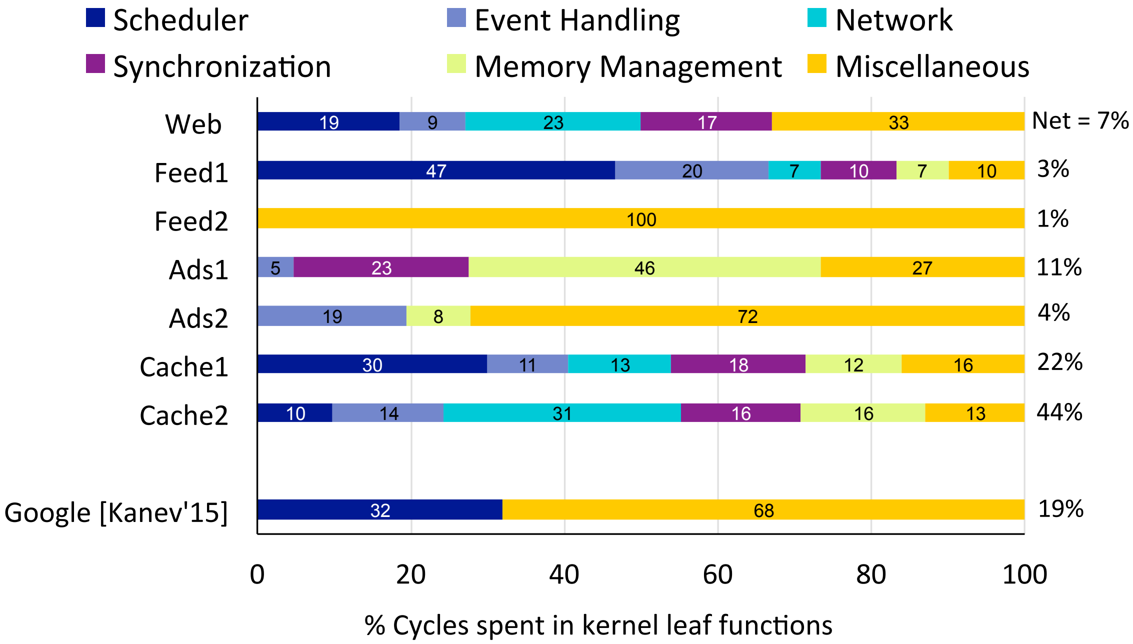


Figure 6.5: Breakdown of cycles spent in various kernel leaf functions: Kernel scheduler, event handling, and network overheads can be high.

network protocol stacks [114, 226].

6.1.3.2 Kernel

We depict the cycles spent in kernel leaf functions in Fig. 6.5. We make three observations: (1) Microservices with a high kernel overhead—`Cache1` and `Cache2`—invoke scheduler functions frequently. Software/hardware optimizations [210, 274, 154, 153, 215, 318, 320, 165, 451] that reduce scheduler latency (e.g., intelligent thread switching and coalescing I/O) might considerably improve `Cache` performance. (2) `Cache2` spends significant cycles in I/O and network interactions. Optimized systems [402, 289, 274, 154, 408, 329] that incorporate kernel-bypass and multi-queue NICs might minimize `Cache2`'s kernel overhead. (3) Prior work [285] reports only the kernel scheduler overhead for Google's services. They typically mirror overheads seen in `Cache1` and `Cache2`.

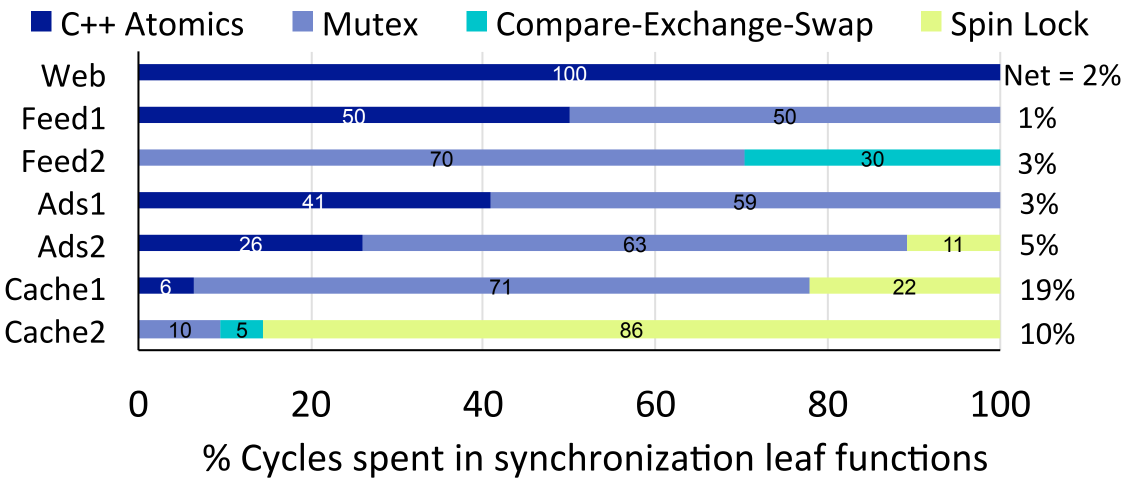


Figure 6.6: Breakdown of CPU cycles spent in synchronization functions: Cache frequently uses spin locks to avoid thread wakeup delays.

6.1.3.3 Synchronization

Microservices such as Cache over-subscribe threads to improve service throughput [446]. Hence, such microservices frequently synchronize various thread pools. We portray these synchronization overheads in Fig. 6.6. We find that Cache, which exhibits a high synchronization overhead, spends several cycles in spin locks that are typically deemed performance inefficient [131, 347]. However, Cache implements spin locks since it is a microsecond-scale microservice [446], and is hence more prone to microsecond-scale performance penalties that can otherwise arise from thread re-scheduling, wakeups, and context switches [448].

6.1.3.4 C Libraries

We characterize overheads from C libraries in Fig. 6.7. We observe that Feed2, Ads1, and Ads2 perform several vector operations as they deal with large feature vectors. Web spends significant cycles parsing and transforming strings to process queries from the many URL endpoints it implements. Web also performs several hash table look-ups to (1) maintain query parameters, (2) identify services to contact, and (3) merge responses. We conclude many microservices can benefit from optimizing vector operations [315], string computations [236, 436], and hash table look-ups [439, 454].

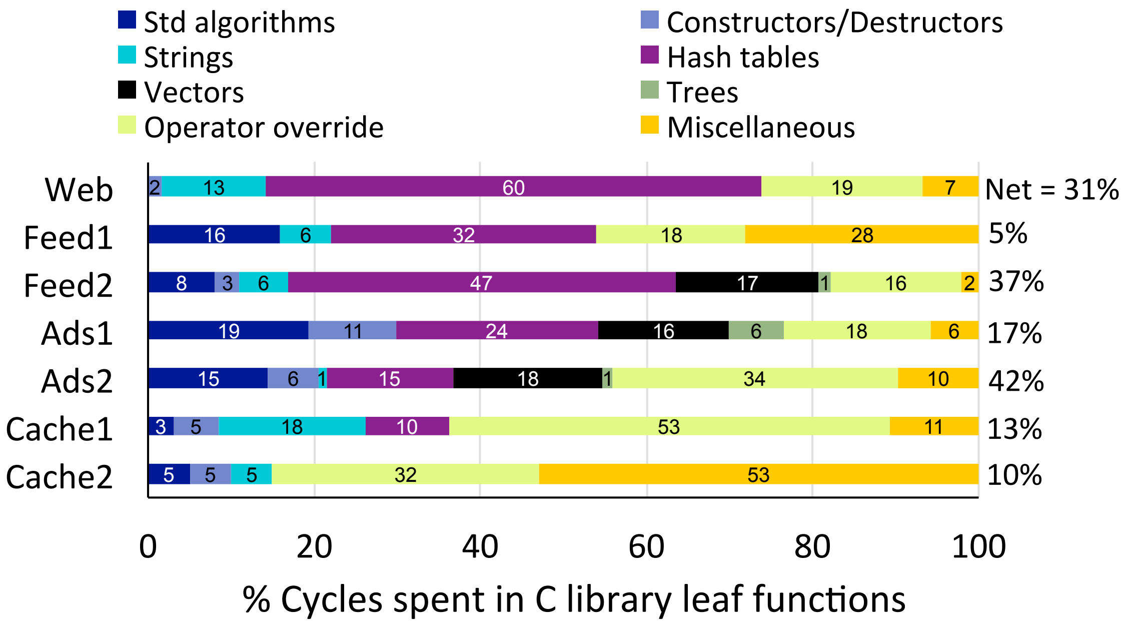


Figure 6.7: Breakdown of CPU cycles spent in C libraries: ML services perform several vector operations while dealing with large feature vectors.

6.1.3.5 IPC scaling

We show `Cache1`'s per-core IPC scaling for key leaf functions in Fig. 6.8. We report IPC across three CPU generations to see whether IPC scales as expected.

We make several observations: (1) Each leaf function type uses less than half of the theoretical execution bandwidth of a GenC CPU (theoretical peak IPC of 4.0). As such, simultaneous multithreading is effective for these microservices and is enabled in our CPUs. Given our production microservices' larger codebase, larger working set, and more varied memory access patterns, we do not find a lower typical IPC surprising. (2) Kernel IPC is typically low and also scales poorly. Accelerating the kernel is non-trivial as it is neither small, nor self-contained, and cannot be easily optimized in hardware. However, software optimizations that minimize scheduler, I/O, and network overheads can improve kernel IPC [210, 274, 154, 153]. (3) C libraries' IPC scales well across CPU generations. This observation is unsurprising as many hardware vendors primarily rely on open-source benchmarks [253] that heavily use C libraries (see Fig. 6.2) to make architecture design

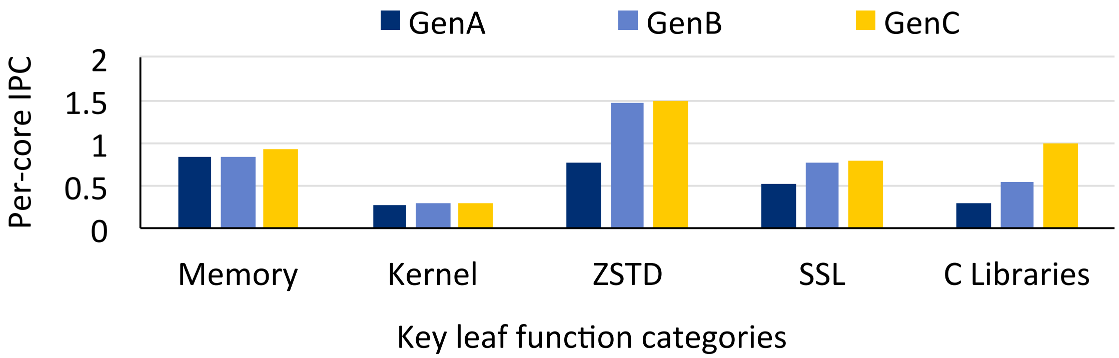


Figure 6.8: Cache1's IPC scaling across three CPU generations for key leaf functions: Kernel IPC is typically low & scales poorly.

Table 6.3: Categorization of microservice functionalities.

Functionality category	Examples of service operations
Secure and insecure I/O	Encrypted/plain-text I/O sends & receives
I/O pre/post processing	Allocations, copies, etc before/after I/O
Compression	Compression/decompression logic
Serialization	RPC serialization/deserialization
Feature extraction	Feature vector creation in ML services
Prediction/ranking	ML inference algorithms
Application logic	Core business logic (e.g., Cache's key-value serving)
Logging	Creating, reading, updating logs
Thread pool management	Creating, deleting, synchronizing threads

decisions. (4) Typically, we see only a small IPC gain from GenB to GenC. This trend suggests the need to specialize hardware for key leaf functions.

6.1.4 Service Functionality Characterization

We attribute CPU cycles to microservice functionalities in Fig. 6.9 to identify key microservice overheads (as motivated in Fig. 6.1). We define how we pool various functionalities in Table 6.3. Note that each functionality category typically includes several leaf function categories. For example, despite ML inference being heavy on math leaf functions, it can also comprise memory movement and C library leaves.

We make four observations: First, several microservices face significant orchestration overheads from performing operations that are not core to the application logic, but instead

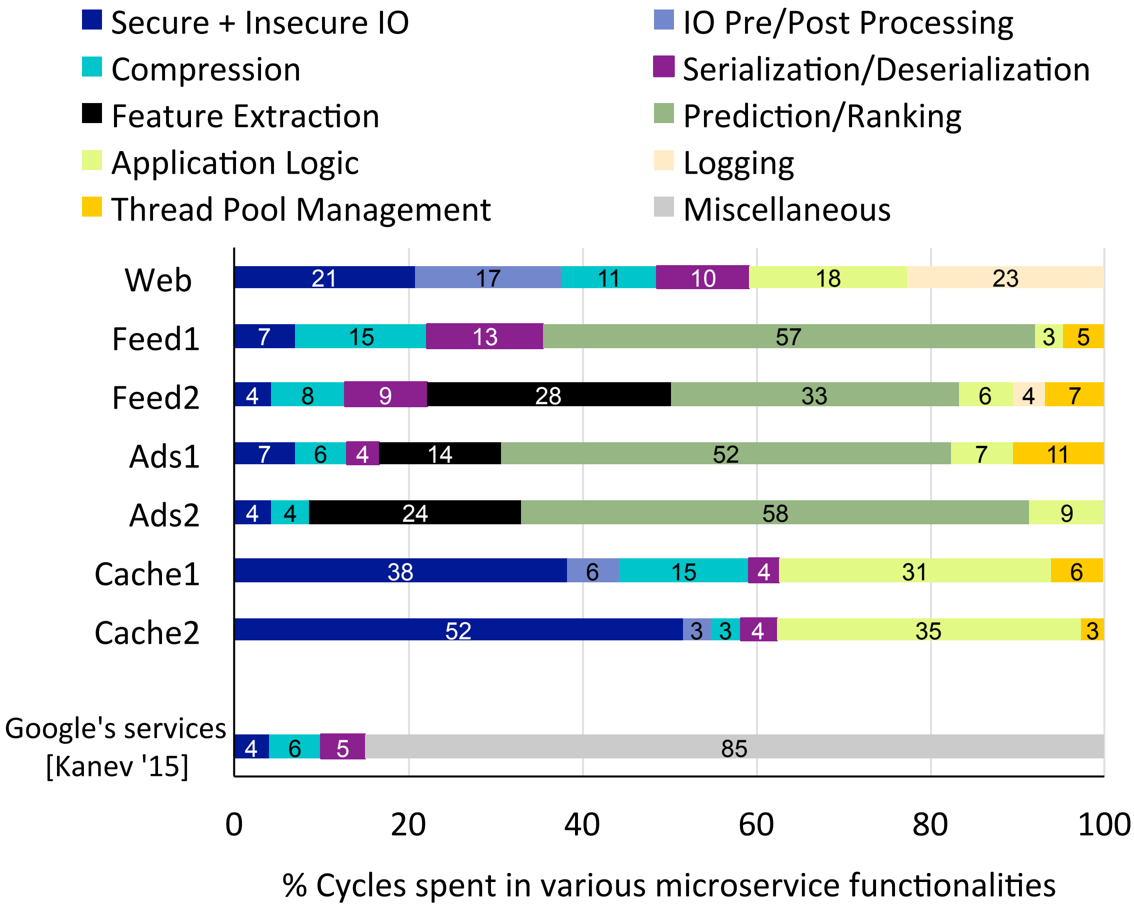


Figure 6.9: Breakdown of CPU cycles spent in various microservice functionalities: Orchestration overheads are significant & fairly common.

facilitate application logic such as compression, I/O, and logging. For example, the microservices that perform ML inference—`Feed1`, `Feed2`, `Ads1`, and `Ads2`—spend as few as 33% of cycles on ML inference, consuming 42% - 67% of cycles in orchestrating inference; (note that the "application logic" for these microservices includes core non-ML operations such as merging results). Hence, even if modern inference accelerators [279, 134, 491, 483] were to offer an infinite inference speedup, the net microservice performance would only improve by 1.49x - 2.38x. There is hence a great need for architects to accelerate the orchestration work that facilitates the core application logic.

Second, several orchestration overheads are common across microservices; accelerating them can significantly improve our global fleet's performance. For example, `Web`, `Cache1`, and Cache2 spend a significant portion of cycles executing I/O—i.e., sending and receiving RPCs. Web incurs a high I/O overhead since it implements many URL endpoints and communicates with a large back-end microservice pool. Cache1 and Cache2 are leaf microservices that support a high request rate [446]—they frequently invoke RPCs to communicate with mid-tier microservices. These microservices can benefit from RPC optimizations such as kernel-bypass and multi-queue NICs [402, 289, 274, 154, 408, 329]. Additionally, Web, Feed1, Feed2, and Cache1 spend several cycles in compression and serialization (similar to Google's services [285]); they can benefit from accelerating these common orchestration overheads [178, 435, 224, 128, 150, 301].

Third, Web spends only 18% of cycles in core web serving logic (parsing and processing client requests), consuming 23% of cycles in reading and updating logs. It is unusual for applications to incur such high logging overheads; only few academic studies focus on optimizing them.

Fourth, Ads1, Feed2, Cache1, and Feed1 incur a high thread pool management overhead. Intelligent thread scheduling and tuning [242, 412, 457, 485, 273, 302] can help these services.

We conclude that application logic disaggregation across microservices and the consequent increase in inter-service communication at hyperscale has resulted in significant and common orchestration overheads in modern data centers.

6.1.4.1 IPC scaling

In Fig. 6.10, we show Cache1's per-core IPC for key microservice functionalities across three CPU generations. We find that the I/O IPC remains low across CPU generations. Since I/O calls primarily invoke kernel functions, the low kernel IPC (see Fig. 6.8) contributes to the low I/O IPC. Additionally, there is little IPC improvement for key-value store operations. Since key-value stores are typically memory intensive [171], the low memory IPC (Fig. 6.8) results in a low key-value store IPC.

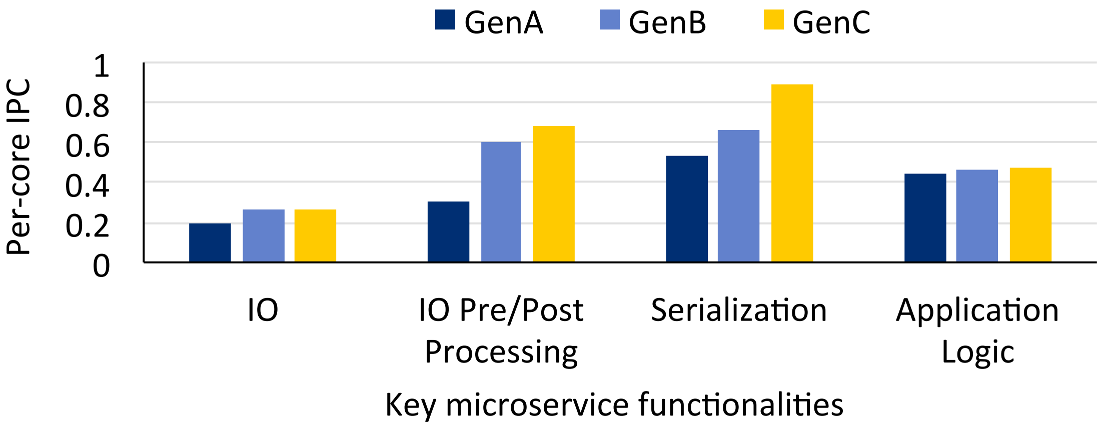


Figure 6.10: Cache1's IPC scaling across three CPU generations for key functionality categories: A low I/O IPC is primarily due to a low kernel IPC.

We summarize our characterization findings in Table 6.4.

6.2 The Accelerometer Model

Overheads identified by our characterization can be accelerated in the hardware via CPU optimizations (e.g., specialized hardware instructions) [53, 368, 369] or custom accelerator devices (e.g., ASICs). Investing in hardware acceleration often requires (1) designing new hardware, (2) testing it, and (3) carefully planning capacity to provision the hardware to match projected load. Given the uncertainties inherent in projecting customer demand, investing in diverse custom hardware is risky at scale, as the hardware might not live up to its expectations due to performance bounds precipitated by offload-induced overheads [127].

Simple analytical models enable better hardware investments by identifying performance bounds early in the design phase. However, existing models for hardware acceleration [127, 188] fall short in the context of microservices as they are oblivious to offload overheads induced by microservice threading designs such as synchronous vs. asynchronous offload to an accelerator. For example, existing models [127, 188] assume that the CPU waits while the offload operates i.e., offload is synchronous. However, for many functionalities, offload may be asynchronous; the CPU may continue doing useful work concurrent with

Table 6.4: Summary of findings and suggestions for future optimizations.

Finding	Acceleration opportunity
Significant orchestration overheads (§6.1.4)	Software and hardware acceleration for orchestration rather than just application logic
Several common orchestration overheads (§6.1.4)	Accelerating common overheads (e.g., compression) can provide fleet-wide wins
Poor IPC scaling for several functions (§6.1.3.5, §6.1.4.1)	Optimizations for specific leaf functions and service functionality categories
Memory copies & allocations are significant (§6.1.3, §6.1.3.1)	Dense copies via SIMD, copying in DRAM, Intel's I/O AT, DMA via accelerators, PIM
Memory frees are computationally expensive (§6.1.3, §6.1.3.1)	Faster software libraries for freeing memory, hardware support to remove pages
High kernel overhead and low IPC (§6.1.3, §6.1.3.5)	Coalesce I/O, user-space drivers, in-line accelerators, kernel-bypass
Logging overheads can dominate (§6.1.4)	Optimizations to reduce log size or number of log updates
High compression overhead (§6.1.3, §6.1.4)	Bit-Plane Compression, Buddy compression, dedicated compression hardware
`Cache` synchronizes frequently (§6.1.3, §6.1.3.3)	Better thread pool tuning and scheduling, Intel's TSX, coalesce I/O, vDSO
High event notification overhead (§6.1.3.2)	RDMA-style notification, hardware support for notifications, spin vs. block hybrids

the offload. Extending prior models [127], we develop *Accelerometer* to capture this concurrency and realistically model microservice speedup for various hardware acceleration strategies (e.g., on-chip vs. off-chip). In this section, we (1) describe the acceleration strategies *Accelerometer* models, (2) discuss system abstractions it assumes, (3) define *Accelerometer*'s model parameters, (4) detail how it models speedup for various threading designs, and (5) highlight *Accelerometer*'s applications.

6.2.1 Acceleration Strategies

Accelerometer models three kinds of hardware acceleration strategies to accelerate an algorithm or *kernel*—on-chip, off-chip, and remote.

On-chip. On-chip acceleration optimizes components on the CPU die (e.g., wider SIMD units [465], Intel's AES-NI hardware encryption instruction [53], and CPU modifications [368, 300]). Offload latencies are typically ns-scale.

Off-chip. Off-chip accelerators are typically contacted via PCIe and coherent interconnects [453] (e.g., GPUs, smart NICs, and ASICs). Offload latencies are $\sim \mu$s-scale [382].

Table 6.5: Description of the Accelerometer analytical model parameters.

Symbol	Parameter description	Units
C	Total cycles spent by the host to execute all logic in a fixed time unit	Cycles
g	Size of an offload	Bytes
n	Number of times the host offloads a kernel of lucrative size in a fixed time unit	N/A
o_0	Cycles the host spends in setting up the kernel prior to a single offload	Cycles
Q	Avg. cycles spent in queuing between host and accelerator for a single offload	Cycles
L	Avg. cycles to move an offload from host to accelerator across the interface, including cycles the data spends in caches/memory	Cycles
o_1	Cycles spent in switching threads (due to context switches and cache pollution) for a single offload	Cycles
A	Peak speedup of an accelerator	N/A
α	A constant ≤ 1	N/A
C_b	Cycles spent by the host per byte of offload data	Cycles

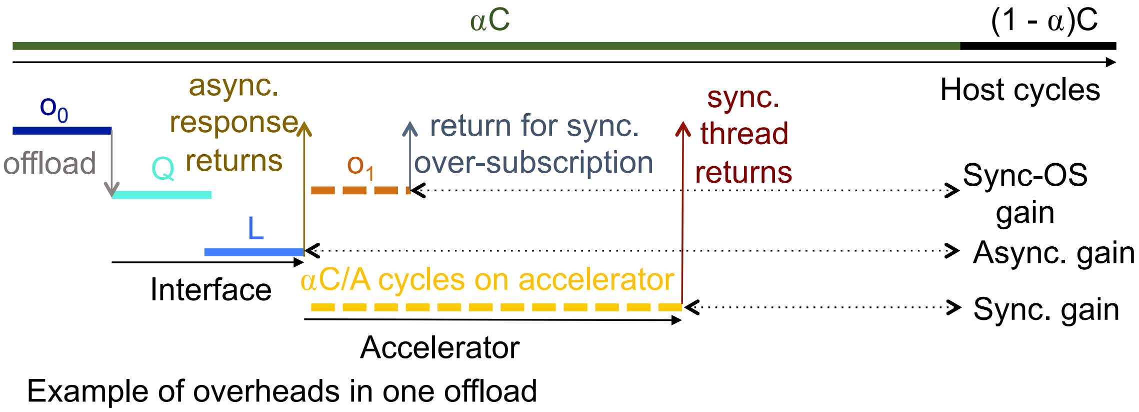


Figure 6.11: Example timeline of host & accelerator.

Remote. Remote accelerators are off-platform devices contacted via the network. Examples include remote ML inference units [248], network switches [426, 336], remote encryption units [158], and remote GPUs [214]. Offload latencies are typically ms-scale when using commodity ethernet [415].

6.2.2 System Abstraction

Accelerometer assumes an abstract system with three components (1) *host*—a general-purpose CPU, (2) *accelerator*—custom hardware to accelerate a kernel, and (3) *interface*—the communication layer between the host and the accelerator (e.g., a PCIe link). The interface helps define overheads from dispatching work to an accelerator (e.g., preparing the kernel for offload, offload latency, and queuing delays). Hence, the interface abstraction can easily help model speedup for diverse acceleration strategies. With these system abstractions, we build the *Accelerometer* model such that it abstracts the underlying hardware architecture using parameters defined below (see Table 6.5).

6.2.3 Parameter Definition

Accelerometer makes a few assumptions to retain model simplicity while still being able to estimate microservice speedup. Similar to LogCA [127], *Accelerometer* assumes that (1) the kernel's execution time is a function of *granularity g*—i.e., the data offload size and (2) the host and accelerator use kernels of the same complexity. It defines C as the total host cycles spent to execute both kernel and non-kernel logic in a fixed time unit; C is inversely proportional to the host's busy frequency for a time unit of one second. It uses Amdahl's law to define a constant $\alpha \leq 1$, such that the host spends ($\alpha * C$) cycles executing the kernel and ($(1-\alpha) * C$) cycles executing the non-kernel logic (as shown in Fig. 6.11). *Accelerometer* assumes that data offload is unpipelined (i.e., the accelerator requires the entire block to start operating); it considers the average latency of such an offload, L. The offload latency distribution can be found by multiplying the offload latency of a single byte with g for each offload. When data offload is pipelined, L is independent of g; we do not study pipelined offloads as our existing systems use unpipelined offloads. The peak achievable accelerator speedup factor, A, helps define cycles spent in the accelerator such that cycles spent on the host to execute the kernel is cut down by the acceleration factor, or $\frac{\alpha * C}{A}$.

6.2.4 Modeling Diverse Threading Designs

We develop *Accelerometer* to model the microservice throughput speedup (referred to as "speedup") and the microservice per-request latency speedup (referred to as "latency reduction") for the three acceleration strategies. Modeling both speedup and latency reduction helps ensure that acceleration enables a higher throughput (i.e., more QPS) without violating latency Service Level Objectives (SLOs). To model speedup, *Accelerometer* identifies

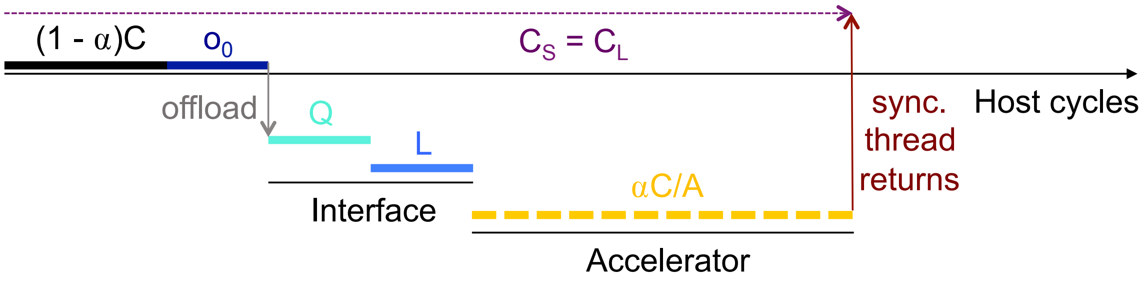


Figure 6.12: Modeling Sync C_S and C_L for one offload.

how many fewer host cycles are needed to execute the kernel when there is acceleration—spending fewer host cycles on the kernel frees up host cycles to do more work, improving throughput. It defines speedup as the ratio of total cycles spent by the host when there is no acceleration, C, to the total cycles spent by the host when the kernel is accelerated, C_S, or $\frac{C}{C_S}$. To model per-request latency reduction, it identifies the total cycles taken to execute a request when there is acceleration; spending fewer cycles for a request due to acceleration reduces per-request latency. It defines latency reduction as the ratio of C to the total cycles spent on the host and the accelerator, C_L, or $\frac{C}{C_L}$.

Unlike LogCA [127], we find that when data is offloaded to an accelerator, the speedup $\frac{C}{C_S}$ and latency reduction $\frac{C}{C_L}$ depend on the acceleration strategy as well as the threading design used to offload (e.g., synchronous vs. asynchronous offload). For example, in a synchronous offload the host waits for the accelerator's response before resuming execution (see Fig. 6.11), putting the accelerator's operation cycles ($\frac{\alpha * C}{A}$) in the critical path of the host's execution (i.e., C_S), impacting speedup. Conversely, in an asynchronous offload, the host continues doing useful work concurrent with the accelerator's operation on the offload, removing $\frac{\alpha * C}{A}$ from the critical path of C_S. We extend LogCA to model speedup and latency reduction for both synchronous and asynchronous offload.

Synchronous. When a host thread offloads work to an accelerator synchronously, it waits in the `blocked` state for the accelerator's response. If the microservice runs one thread per core, the host's core waits for the accelerator's response—we refer to this scenario as `Sync`. Hence, C_S and C_L will include cycles spent on the accelerator $\frac{\alpha * C}{A}$, as shown in Fig. 6.12. Moreover, the host can consume additional cycles to (1) prepare the kernel for offload, o_0, (2) transfer the kernel to the accelerator, L, and (3) wait in a queue for the accelerator to become available, Q. Hence, C_S and C_L can also include these additional overheads per offload. Considering n offloads occur in a given time unit, *Accelerometer* defines Sync speedup $\frac{C}{C_S}$ and latency reduction $\frac{C}{C_L}$ as: $\frac{C}{(1-\alpha)C + \frac{\alpha C}{A} + n(o_0 + L + Q)}$, where C_S and C_L comprise host cycles spent in (1) non-kernel logic, (2) waiting for the accelerator's response, and (3) offload-induced overheads across the n offloads. Making this equation appear similar to Amdahl's law with offload overheads (i.e., dividing by C),

$$\text{Sync } \frac{C}{C_S} or \frac{C}{C_L} = \frac{1}{(1-\alpha) + \frac{\alpha}{A} + \frac{n}{C}(o_0 + L + Q)} \tag{6.1}$$

In eqn. (6.1), $(n * Q)$ is the mean queuing delay for n offloads; Q enables projecting speedup based on accelerator load. Replacing $(n * Q)$ with $\sum_{i=1}^{n}(Q_i)$ models the queuing distribution. Net speedup is >1 when the host spends more cycles when unaccelerated—i.e., $(\alpha * C) > \frac{\alpha C}{A} + n(o_0 + L + Q)$. In eqn. (6.1), we consider n kernel offloads that each improve speedup (or reduce latency). To determine whether a kernel offload improves speedup, we consider the offload granularity, g, such that the host spends C_b cycles per byte of g. A single offload improves speedup when the cycles a host would spend in executing all bytes of a g-size kernel offload is greater than the cycles spent in accelerating the kernel (i.e., the sum of cycles spent on the accelerator executing the g-size offload and the offload overheads—$o_0 + L + Q$), or:

$$C_b * g > \frac{C_b * g}{A} + o_0 + L + Q \tag{6.2}$$

Eqn. (6.2) can be extended to model the kernel's complexity (e.g., sub-linear, linear, or super-linear) using g^{β} [127]. For example, $\beta = 1$ for a linear complexity kernel.

In reality, several microservices (e.g., `Web` and `Cache`) oversubscribe threads to improve throughput by having more threads than available cores. Oversubscription allows a host

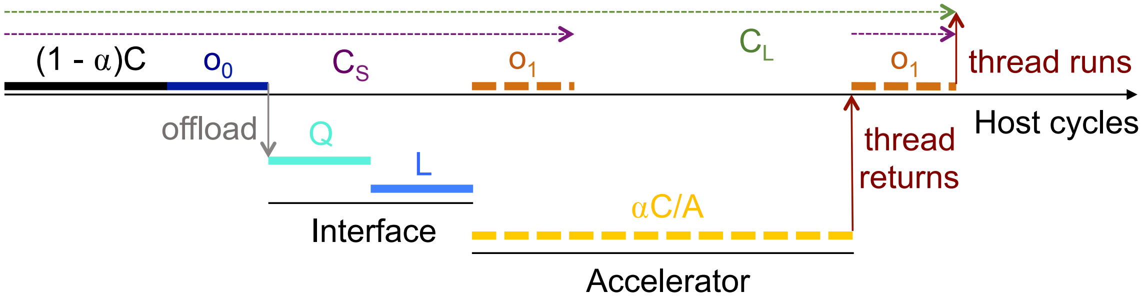


Figure 6.13: Modeling Sync-OS C_S and C_L for one offload.

to schedule an available thread to process new work, while the thread that offloaded work blocks awaiting the accelerator's response. The host continues to perform useful work instead of wasting cycles waiting for the accelerator's response; we refer to this synchronous thread Over-Subscription as Sync-OS. Hence, the accelerator's cycles $\frac{\alpha C}{A}$ do not affect C_S, as shown in Fig. 6.13. Instead, C_S is affected by the OS-induced overhead to switch to an available thread, o_1. The $(L+Q)$ overhead persists when the host's device driver synchronously waits for an offload acknowledgement from an off-chip accelerator before switching threads. However, $(L+Q)=0$ when (1) the device driver does not wait for the off-chip accelerator's acknowledgement or (2) the accelerator is remote. Hence, the speedup is:

$$\texttt{Sync-OS}\ \frac{C}{C_S} = \frac{1}{(1-\alpha)\ +\ \frac{n}{C}(o_0\ +\ L\ +\ Q\ +\ 2o_1)} \tag{6.3}$$

Speedup is >1 when: $(\alpha * C) > n(o_0+L+Q+2o_1)$. A single offload improves throughput speedup when the cycles a host would spend in executing that offload is greater than the offload-induced overhead—$o_0+L+Q+2o_1$, or:

$$C_b\ *\ g\ >\ o_0\ +\ L\ +\ Q+\ 2o_1 \tag{6.4}$$

The latency reduction remains the same as eqn. (6.1) (since the accelerated per-request latency, C_L, will include cycles spent on the accelerator), but must now account for o_1. The μs-scale o_1 overhead [470, 325] can dominate in μs-scale microservices such as

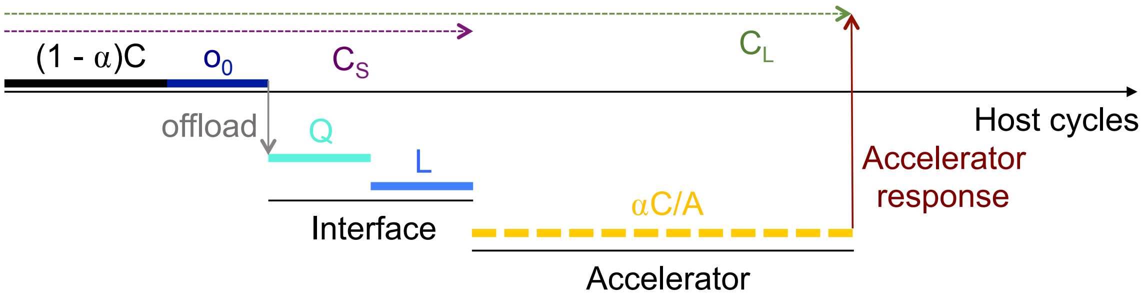


Figure 6.14: Modeling `Async` C_S and C_L for one offload.

`Cache` [446], making it feasible to incur a throughput gain at the cost of a per-request latency slowdown. Service operators can use the following latency reduction equation to ensure that the latency SLO is not violated.

$$\texttt{Sync-OS}\ \frac{C}{C_L} = \frac{1}{(1-\alpha) + \frac{\alpha}{A} + \frac{n}{C}(o_0 + L + Q + o_1)} \tag{6.5}$$

Latency is reduced when: $(\alpha * C) > \frac{\alpha C}{A} + n(o_0 + L + Q + o_1)$. A single offload reduces latency when the cycles a host would spend in executing the offload dominates accelerator cycles and offload overheads, or: $(C_b * g) > \frac{C_b * g}{A} + (o_0 + L + Q + o_1)$.

Asynchronous. After a host thread offloads work asynchronously, it continues to process new work without awaiting the accelerator's response. When the response arrives, it can be picked up by (1) the same thread that sent the request or (2) a distinct thread dedicated to pick up responses [449]. When a distinct thread picks up the response, the speedup equation is the same as (6.3) with only one thread switching overhead o_1. The latency reduction equation remains the same as (6.5). If the response is picked up by the same thread that sent the request, $o_1 = 0$ since the OS does not switch threads (see Fig. 6.14); we refer to this scenario as `Async`. Hence the speedup is:

$$\texttt{Async}\ \frac{C}{C_S} = \frac{1}{(1-\alpha) + \frac{n}{C}(o_0 + L + Q)} \tag{6.6}$$

Speedup is >1 when: $(\alpha * C) > n(o_0 + L + Q)$. A single offload improves speedup when:

$$C_b * g > o_0 + L + Q \tag{6.7}$$

Similarly, *Accelerometer* does not consider o_1 when modeling `Async` latency reduction:

$$\texttt{Async}\ \frac{C}{C_L} = \frac{1}{(1-\alpha) + \frac{\alpha}{A} + \frac{n}{C}(o_0 + L + Q)} \tag{6.8}$$

Latency reduces when: $(\alpha * C) > \frac{\alpha C}{A} + n(o_0 + L + Q)$. A single offload reduces latency when: $(C_b * g) > \frac{C_b * g}{A} + (o_0 + L + Q)$.

In some asynchronous designs, the host does not require the accelerator's response for further processing, eliminating o_1 (e.g., when a host sends requests to an encryption accelerator, which then sends encrypted requests to the next microservice). Hence, the speedup equation remains the same as eqn. (6.6). Latency reduction depends on whether acceleration is off-chip or remote since remote accelerator latencies $\frac{\alpha C}{A}$ will not affect a microservice's request latency and will instead show up in the overall application's end-to-end latency. We define the `Async` off-chip per-request latency reduction as eqn. (6.8) and the remote latency reduction as eqn. (6.6).

6.2.5 Accelerometer Use Cases

The *Accelerometer* model shows that speedup and latency reduction depend on the acceleration strategy and microservice threading design. We expect *Accelerometer* to have the following use cases: (1) Data center operators can project fleet-wide gains from optimizing key service overheads. (2) Architects can make better accelerator design decisions and estimate realistic gains by being aware of the offload overheads due to microservice design. *Accelerometer* can help determine trade-offs between various acceleration strategies (e.g., on-chip vs. off-chip) for microservice overheads. Indeed, we validate our models in production and then apply them to project gains for on-chip vs. off-chip recommendations (see Table 6.4) derived from key overheads identified by our characterization.

6.3 Validating the Accelerometer Model

We validate *Accelerometer*'s utility in production using three retrospective case studies. With these studies, we validate all three microservice threading scenarios—`Sync`, `Sync-OS`, and `Async`. Each study covers a distinct acceleration strategy—i.e., on-chip, off-chip, or remote. For each study, we first describe (1) the experimental setup, (2) how we derive model parameters, and (3) how we measure speedup on production systems. We then validate *Accelerometer* by comparing model-estimated speedup with real microservice speedup. We do not compare the latency reduction since our existing production infrastructure lacks necessary support to precisely measure a microservice's per-request latency.

6.3.1 Validation Methodology

We follow a five step process to validate the *Accelerometer* model: (1) we identify offload sizes g that improve speedup, (2) we determine the number of such offloads in one second, n, and the fraction of cycles they constitute, α, (3) we use the *Accelerometer* model to estimate speedup from these n offloads, (4) we compare *Accelerometer*-estimated speedup with real production speedup, and (5) we present a functionality breakdown for both the accelerated and unaccelerated microservices to show how throughput improves. We assume that we can use software to selectively accelerate only those kernel offloads that improve speedup (the kernel execution time can be dominated by overheads for very small offloads [127]).

6.3.2 Experimental Setup

We perform our case studies on Intel Skylake processor platforms (Table 6.1). For each case study, we first measure the real production speedup using an internal tool called Operational Data Store (ODS) [168, 397, 123]. We measure speedup via A/B testing. A/B testing is the process of comparing two identical systems that differ only in a single variable.

Table 6.6: Model parameters used to compare *Accelerometer*-estimated speedup with measured speedup on production systems.

Case Study	C (10^9 cycles)	α	n	o_0 (cycles)	Q (cycles)	L (cycles)	o_1 (cycles)	A	Est. Speedup	Real Speedup
AES-NI	2.0	0.165844	298,951	10	0	3	NA	6	15.7%	14%
Encryption	2.3	0.19154	101,863	0	0	2530	NA	NA	8.6%	7.5%
Inference	2.5	0.52	10	25x10^6	0	NA	12,500	NA	72.39%	68.69%

We conduct A/B tests by comparing the throughput (in QPS) of two identical servers (i.e., same hardware platform, same fleet, and facing the same load) that differ only in terms of whether they accelerate the kernel.

To determine the *Accelerometer*-estimated speedup, we assume a linear complexity kernel, since we cannot easily perform scaling studies on production systems to determine kernel complexity. We measure model parameters using (1) tools such as Strobelight [104], `bpftrace` [20], and `bcc-tools` [66], (2) roofline estimates from device specification sheets, and (3) micro-benchmarks that measure execution time on the host and the accelerator. Some host parameters, once calculated, can be re-used for different kernels on the same system. For each case study, we measure the unaccelerated host's busy frequency to calculate C for one second. To determine whether a specific offload improves speedup (using equations (6.2), (6.4), (6.7)), we use `bpftrace` [20] to measure g's size range and the number of invocations of each granularity. We compute n by aggregating invocations of those offload sizes that improve speedup. To determine α, we first use the service functionality breakdown (see Fig. 6.9) to estimate host cycles spent in the kernel under study. We then use n and these total host cycles to estimate the fraction of kernel cycles that must be offloaded, ($\alpha * C$). We assume an unpipelined interface when estimating L.

6.3.3 Case Study 1: AES-NI for Cache1

We study encryption in `Cache1` with Intel's AES-NI [53] instruction—an on-chip optimization. In this case, `Cache1` uses a `Sync` threading design. We use AES [8] from the OpenSSL [77] cryptography library to build micro-benchmarks to measure L, o_0, and A. We assume $Q = 0$, since the same host thread executes the AES-NI instruction. We

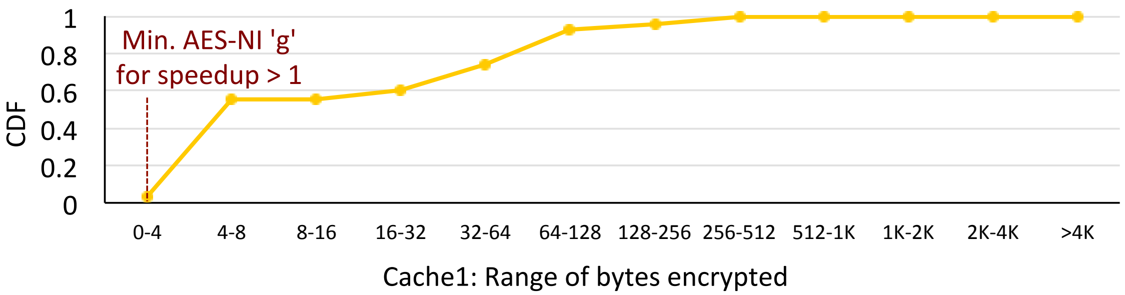


Figure 6.15: CDF of bytes encrypted in Cache1: <512B are frequently encrypted.

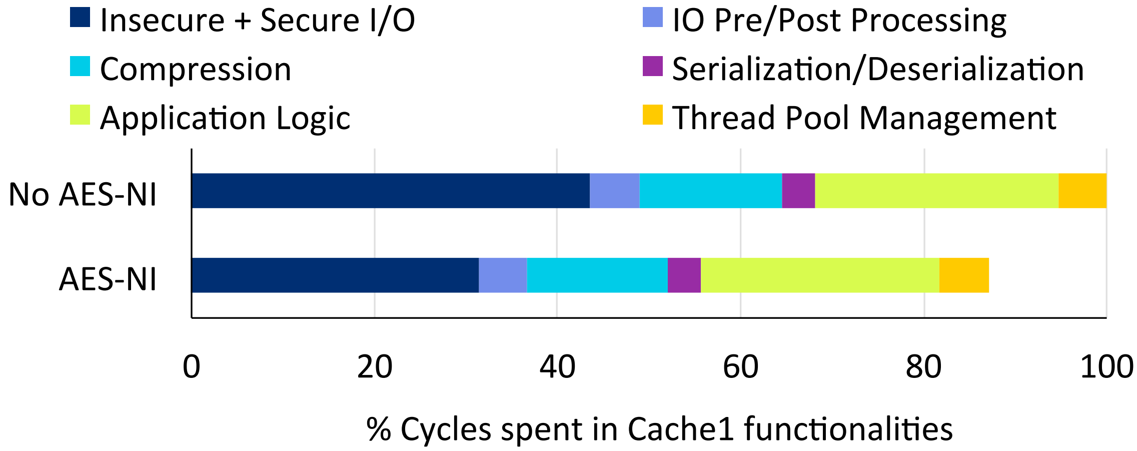


Figure 6.16: Breakdown of cycles spent in Cache1's functionalities for both the no-AES-NI (unaccelerated) & with-AES-NI (accelerated) cases: 12.8% of cycles are freed up with AES-NI.

show the Cumulative Distribution Function (CDF) of `Cache1`'s encryption granularities in Fig. 6.15. We use model parameters defined in Table 6.6 in eqn. (6.2) to determine that a specific offload improves net speedup when $g \geq 1$ Byte (B). Prior work [127] also sees wins with AES-NI for small offload granularities. From Fig. 6.15, we observe that `Cache1`'s encryption size is $\sim\geq 4$ B; hence, all offloads will improve speedup. We confirm that `Cache1` offloads all encryptions in a production system as well.

We then use Table 6.6's parameters in eqn. (6.1) to estimate a speedup of 15.7%. The real production speedup is 14% (as determined via A/B testing). Hence, the *Accelerometer*-estimated speedup differs from the real speedup by only 1.7%. We compare `Cache1`'s functionality breakdown with AES-NI in Fig. 6.16. We observe that AES-NI accelerates the "secure IO" functionality by 73%, saving 12.8% of `Cache1`'s cycles.

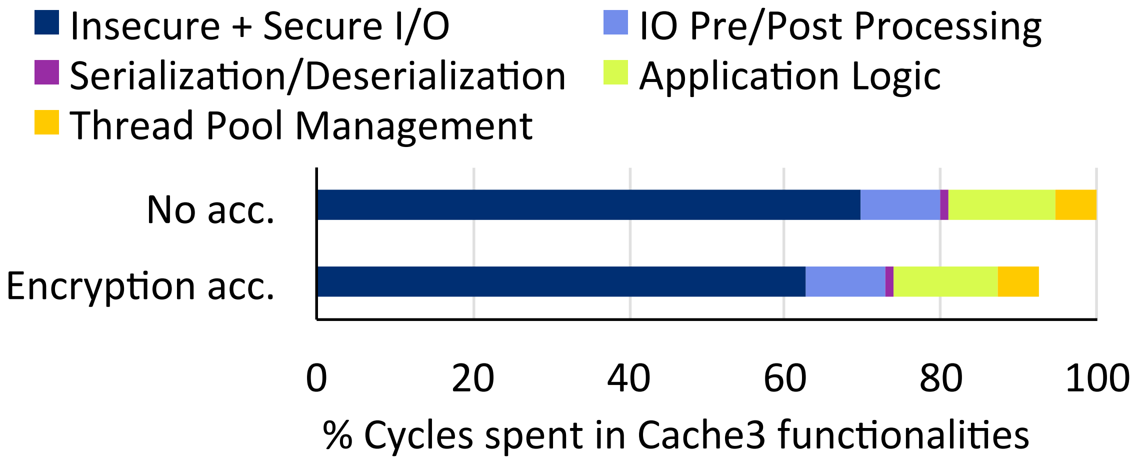


Figure 6.17: Breakdown of cycles spent in Cache3's functionalities when encryption is accelerated vs. not: Secure IO calls are optimized with acceleration.

6.3.4 Case Study 2: Encryption for Cache3

We accelerate encryption in a different microservice, `Cache3`, that is similar to `Cache1` and `Cache2`; we show `Cache3`'s functionality breakdown in Fig. 6.17. The encryption accelerator is off-chip—the host communicates with the accelerator via a PCIe link. The host offloads the encryption kernel to the accelerator asynchronously, and does not require the accelerator to respond (`Async`). However, after offloading a kernel, the host waits for the accelerator to acknowledge receipt. We use the accelerator's specification sheets to (1) estimate L with fair queuing Q and (2) assume $o_0 = 0$.

In this study, we assume that all encryption offloads will improve speedup, since `Cache3`'s software infrastructure does not support selectively offloading only those granularities that yield speedup. We use parameters defined in Table 6.6 in equation (6.6) to estimate speedup. We observe that the PCIe transfer latency is the dominant overhead. After A/B testing, we find that the model overestimates the real speedup by 1.1%.

In Fig. 6.17, we compare the functionality breakdown of an unaccelerated `Cache3` instance with a `Cache3` instance that accelerates encryption. We observe that acceleration improves the encryption (secure IO) overhead by 35.7%, improving `Cache3`'s throughput by 7.5%.

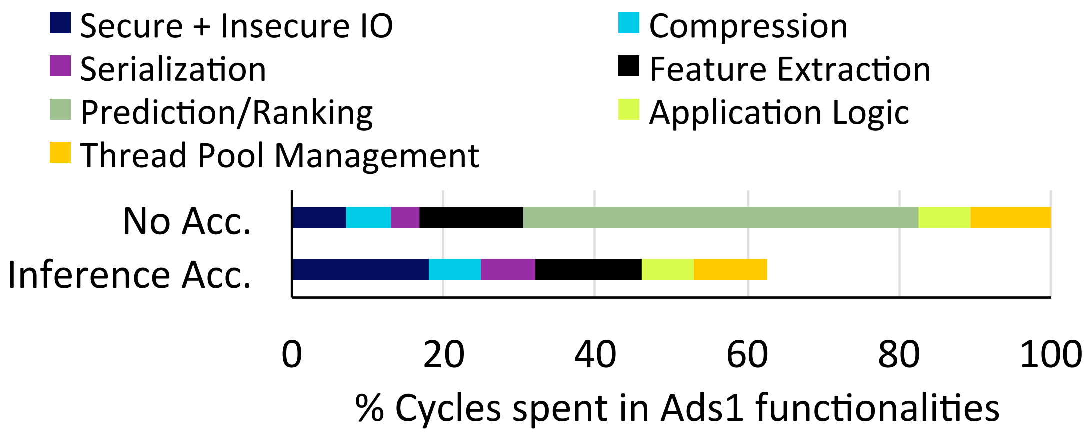


Figure 6.18: Breakdown of cycles spent in Ads1's functionalities for both the inference unaccelerated & accelerated cases: All inference cycles are freed up.

6.3.5 Case Study 3: Inference for Ads1

We deploy a remote Skylake CPU to perform `Ads1`'s ML inference. We note that the end-to-end service throughput decreases when inference is offloaded to a remote CPU (i.e., $A = 1$). However, we expect the host CPU running `Ads1` to incur a speedup, as it no longer does inference locally and uses asynchronous network APIs to offload inference to the remote "accelerator". We validate *Accelerometer* for remote acceleration using this case study.

The host picks up the accelerator's response with a distinct thread (same speedup as `Sync-OS` with a single thread switching overhead o_1). To estimate o_0, we use a micro-benchmark to measure (1) inference invocation counts and (2) feature vector sizes to estimate I/O overheads from offloading to a remote server. We use a micro-benchmark to measure o_1 using the BPF run queue (scheduler) latency tool [66]. We assume $L + Q = 0$ as the accelerator is remote.

We carefully batch inference operations and offload them to the remote CPU only when the batch size is large enough to overcome network overheads (as we cannot violate SLO on a production system). Hence, we assume that all of `Ads1`'s inference offloads improve speedup. We use parameters defined in Table 6.6 in equation (6.3) (with a single o_1) to estimate speedup. Since `Ads1` must invoke many more IO calls to offload inference, it incurs additional IO overheads (o_0). Due to these overheads, we estimate speedup as 72.39%. In reality, remote inference improves Ads1 throughput by 68.69%; our model over-estimates speedup by 3.7%.

In Fig. 6.18, we illustrate Ads1's functionality breakdown for both the remote inference and local inference cases. Although remote inference consumes additional IO cycles, it completely offloads the inference functionality, freeing up host CPU cycles to perform more work. Note that Ads1 achieves this throughput improvement at the expense of a per-request latency degradation since each request faces an additional $\sim$10 ms network traversal delay; we ensure that the per-request latency meets SLO constraints. This result shows that Ads1's latency can be improved if the remote inference CPU (with $A = 1$) is replaced with an inference accelerator with $A > 1$ to overcome network traversal delays.

6.4 Applying the Accelerometer Model

We apply the *Accelerometer* model to project speedup for the acceleration recommendations derived from three key common overheads identified by our characterization: compression, memory copy, and memory allocation (see Table 6.4). We first apply on-chip (Chen et al. [178]) and off-chip (Simek et al. [435]) compression acceleration with Sync, Sync-OS, and Async. We then apply on-chip memory copy (AVX [15]) and allocation acceleration (Kanev et al. [287]); off-chip faces several challenges (e.g., coherence). We apply on-chip offload only with Sync as we only assume CPU core optimizations. We do not see gains from remote acceleration.

We show the model parameters for each acceleration recommendation in Table 6.7. We assume that all on-chip offloads yield gains as we only consider core optimizations with negligible ($o_0 + L$) overhead. We assume $Q = 0$ in all cases.

Table 6.7: Parameters used to model speedup and latency reduction for a few acceleration recommendations from Table 6.4.

Overhead	Acceleration	C (10^9 cycles)	α	n	L (cycles)	o_1 (cycles)	A
Compression	On-chip: Sync	2.3	0.15	15,008	0	NA	5
Compression	Off-chip: Sync	2.3	0.15	9,629	2,300	NA	27
Compression	Off-chip: Sync-OS	2.3	0.15	3,986	2,300	5,750	27
Compression	Off-chip: Async	2.3	0.15	9,769	2,300	NA	27
Memory Copy	On-chip: Sync	2.3	0.1512	1,473,681	0	NA	4
Memory Allocation	On-chip: Sync	2.0	0.055	51,695	0	NA	1.5

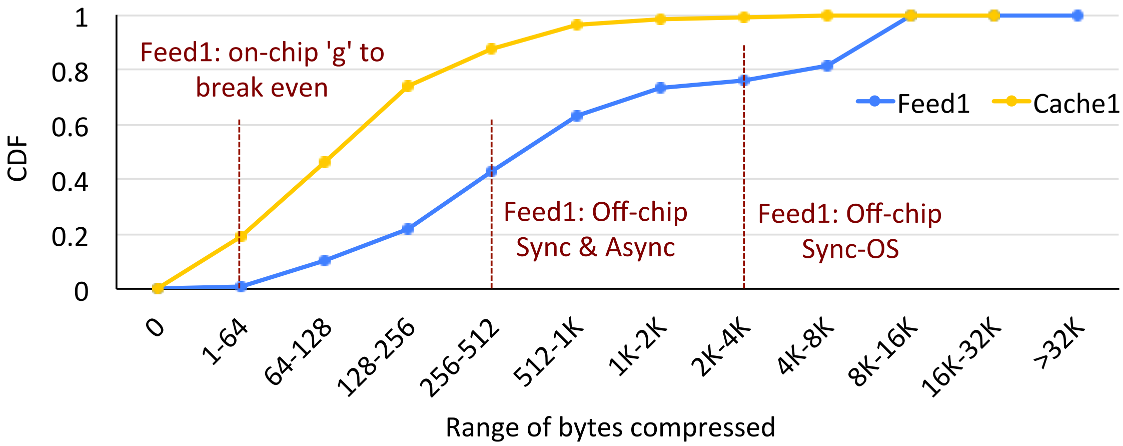


Figure 6.19: CDF of bytes compressed in `Feed1` and `Cache1`: `Feed1` often compresses large granularities.

6.4.1 Compression

In Fig. 6.19, we show the compression granularities' CDF for services with high compression overheads—`Feed1` and `Cache1`. `Feed1` compresses larger granularities than `Cache1`; we focus on `Feed1` in this study. Since `Feed1` spends 15% of cycles in compression, it can achieve an ideal speedup of 17.6%, as shown in Fig. 6.20.

On-chip. We apply Table 6.7's model parameters in eqn. (6.2) to find that an offload improves speedup when $g \geq 1$ B; all of `Feed1`'s compressions will improve speedup. We then use $n = 15{,}008$ in eqn. (6.1) to estimate a speedup of 13.6% as shown in Fig. 6.20,

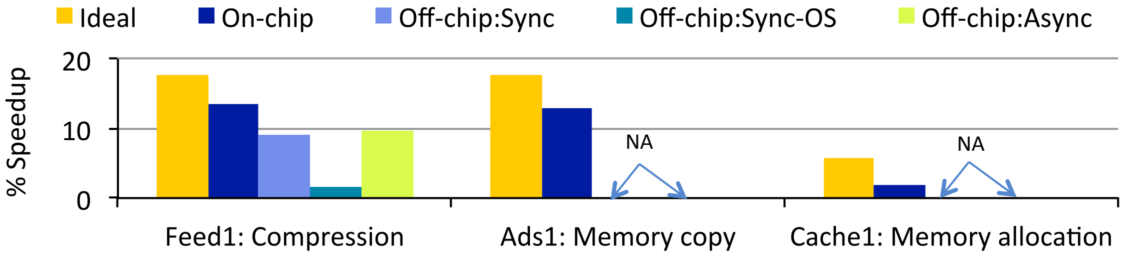


Figure 6.20: *Accelerometer*-estimated speedup for key overheads we identified: Performance bounds from accelerator offload limit achievable speedup.

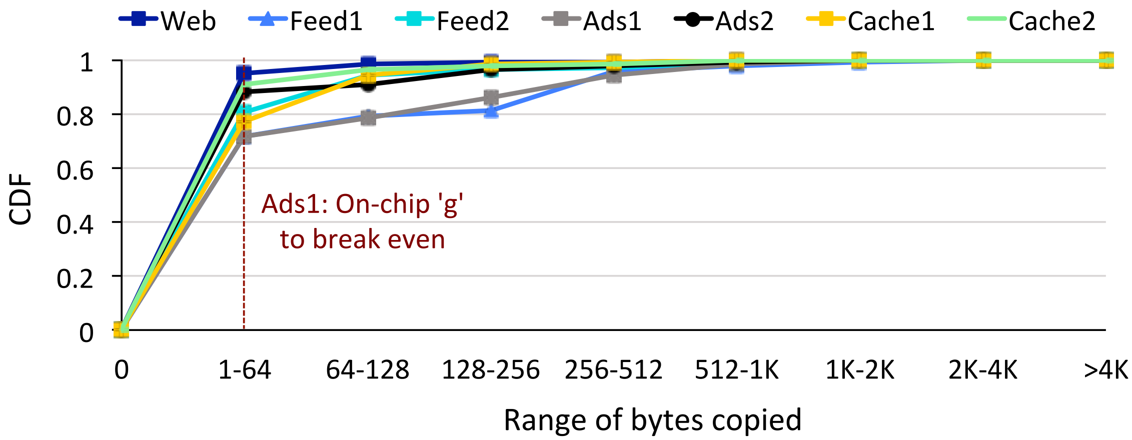


Figure 6.21: CDF of memory copies across microservices: Most microservices frequently copy small granularities.

implying a latency reduction of 13.6%.

Off-chip. From Table 6.7 and eqn. (6.2), we find that a Sync offload improves speedup when $g \geq 425$ B. We note that 64.2% of compressions are ≥ 425 B (Fig. 6.19). Offloading these compressions improves speedup (and reduces latency) by 9% (Fig. 6.20). Similarly, Sync-OS and Async offloads yield speedups of 1.6% and 9.6% respectively, reducing latency by 1.4% and 9.2%. Even though on-chip yields a higher speedup, there might be value in off-chip acceleration as it is easier to design than modifying CPUs. For example, off-chip encryption accelerators can be extended to perform compression to leverage improving two kernels for the price of one offload.

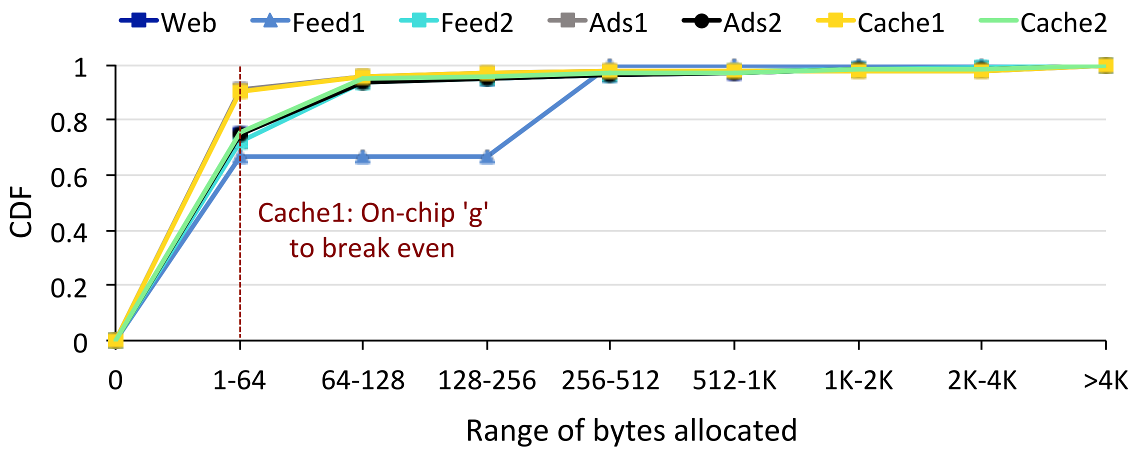


Figure 6.22: CDF of memory allocations across microservices: Most microservices frequently allocate small granularities.

6.4.2 Memory Copy

Fig. 6.21 shows memory copy granularities' CDF across services. We observe that several services often copy $<$ 512 B (smaller than a 4K page). We apply on-chip acceleration [15] for `Ads1` as it incurs the highest copy overhead. We apply Table 6.7's parameters in eqn. (6.1) to project a speedup and latency reduction of 12.7% (Fig. 6.20). Hence, an on-chip copy optimization [15] can yield significant gains.

6.4.3 Memory Allocation

We show the CDF of memory allocations in Fig. 6.22. Most microservices perform small allocations (typically $<$ 512 B). We analyze the microservice with the highest memory allocation overhead—`Cache1`. We find that offloading all of `Cache1`'s 51,695 memory allocations to an on-chip accelerator [287], will result in a 1.86% speedup and latency reduction (Fig. 6.20).

6.5 Related Work

We discuss two categories of related work.

Data center overheads. Very few prior works study how cycles are spent in modern data centers. Kanev et al. [285] investigate the "data center tax" or the performance impact of seven types of leaf functions across Google's server fleet. Mars et al. [354, 352, 353] use key factors that impact available heterogeneity in CPUs to improve warehouse-scale performance. In contrast, we provide a deep-dive into Facebook's important microservices via leaf function, as well as service functionality breakdowns.

Analytical models. Altaf et al. developed the LogCA [127] model to estimate gains from hardware acceleration. We extend LogCA [127] to support various microservice threading designs to estimate throughput and latency improvements.

Several works develop analytical models for heterogeneous architectures. Chung et al. [182] model custom logic, FPGAs, and GPGPUs. Hempstead et al. [251] propose Navigo to determine accelerator area requirements to maintain performance trends. Nilakantan et al. [384] estimate communication costs in heterogeneous architectures. Kumar et al. identify performance-efficient data offload granularities. These models use several parameters to accurately determine performance improvements. *Accelerometer* uses a small parameter set to build simple models for microservice speedup and latency reduction.

Several models are architecture-specific [500, 259, 260, 440, 365, 192]. Song et al. [440] predict performance and power trade-off in GPUs. Hong et al. model GPU execution time [259] and power requirements [260]. Daga et al. [192] discuss communication overheads in APUs and GPUs. Meswani et al. [365] develop models for high performance applications. The *Accelerometer* model abstracts the underlying architecture and can be used across various accelerator types.

Apart from LogCA [127], *Accelerometer*'s simplicity is similar to the Roofline model [342]. Extensions to the Roofline model [387, 344] target specific architectures such as mobile SoCs [255], GPUs [275], vector processing units [423], and FPGAs [191]. While the Roofline model aims to aid programmability, our models seek to expose performance bounds from an accelerator's interface for hyperscale microservices.

6.6 Long-Term Impact Potential

This work has been recognized for its long-term impact potential with an IEEE Micro Top Picks distinction (one of 12 total computer architecture papers to receive this recognition in 2020) [445]. To quote an anonymous IEEE Micro Top Picks reviewer, "*This paper is clearly solid work that advances the state-of-the-art in multiple directions: more realistic profiling of data center applications, modeling of different acceleration offloading strategies, and ideas for potential accelerators. Its immediate impact is a roadmap for accelerator development and deployment for hyperscalers, but it will also trigger further research in improving the model and the offloading and communication techniques between host and multiple accelerators.*"

Additionally, this work won the "*Best Presentation Award*"[4] at the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2020 and has received technical press coverage [12, 90, 5, 11]. We discuss long-term implications, highlighting the impact this work has already had.

Accelerometer in production. As microservices evolve, *Accelerometer*'s generality makes it even more suitable in determining new hardware requirements early in the design phase. Since we validated *Accelerometer* in production and made it open-source [4], we are happy to report that it has been adopted by multiple hyperscale enterprises (e.g., with developing their encryption and compression accelerators) to make well-informed hardware decisions [445]. We expect *Accelerometer* to trigger research in developing more complex models that account for overheads induced by offloading to specific accelerators (e.g., software batching implications on FPGA memory bandwidth vs. latency).

Influence on commercial hardware designs. In this work, we took a step back and answered the Amdahl's Law question of: which overheads prevail even after offloading a

[4] Presentation video available at: `https://www.youtube.com/watch?v=H1a6FPFKG4A`.
Fun fact: ASPLOS 2020 was the first systems/architecture conference that was held virtually during the COVID-19 pandemic. Attempting to create a live audience during this challenging time, I presented my entire ASPLOS talk to my dog, Po Gopal. Watch the video to see his adorable reactions to this work!

microservice's main functionality to an accelerator? Our comprehensive study of real-world microservices (detailed in Section 6.1) definitively indicates the need for a qualitatively different approach to future accelerator efforts. So far, data center hardware acceleration efforts have primarily focused on the most costly operations of a few "killer" applications (e.g., ML inference [279]). However, accelerating orchestration overheads can offer greater benefits as they are significant and common across microservices.

As web service architectures grow more fragmented and granular (e.g., deeper microservice pipelines and serverless architectures), it becomes more critical to optimize the increasingly ubiquitous orchestration overheads. However, accelerating orchestration overheads is non-trivial as (1) orchestration libraries are already well-optimized in software and (2) orchestration function invocations are frequent, involve small data granularity, and are interspersed between other microservice code. Hence, accelerating orchestration overheads will require different techniques than those used in throughput-based specialization blocks with coarse-grained offloads (e.g., video processing).

Although *Accelerometer* provides the first step in determining required acceleration strategies, we expect significant academic and industrial interest in rethinking accelerators for fine-grained orchestration operations. Already, a few hardware vendors have used our study's insights to influence hardware customization for orchestration operations [445].

Characterization approach and tool. While it is relatively simple to measure the CPU cycles spent in leaf functions, it is extremely difficult to categorize every path's functionality in a microservice's entire call stack, as microservices have deep, complex software stacks that are hard to parse and classify. We developed a methodology to systematically classify each call trace path: we applied expert insights to identify service functionality classification rules that we then used to categorize cycles spent in various microservice functionalities [445].

We integrated this characterization tool into Facebook's fleet-wide performance monitoring infrastructure; it currently assimilates statistics from hundreds of thousands of servers from around the world to help developers visualize the performance impact of their code changes at hyperscale [445]. With the decline of hardware performance scaling, there is a greater need for researchers to develop such tools for performance monitoring and optimization at all levels of the systems stack.

Industry-academia collaborative benchmarking efforts. Many hardware vendors rely on open-source benchmarks such as SPEC that heavily use C libraries to make architecture decisions [445]. Hence, in our characterization in Section 6.1, we observe that only C libraries' IPC scales well across CPU generations, but the other overheads (e.g., memory movement and encryption) show little to no improvement.

There is immense value in validating commonly-used benchmarks with real-world application behaviors. Our characterization drove hardware vendors to consider more representative benchmarks (in place of traditional ones they used for decades) when evaluating hardware designs [445]. To quote an Intel researcher, "*We were driving blind until seminal works like these came along and told us to refocus our design efforts on more representative applications*". This work has resulted in an industry-academia joint collaborative effort [5] to design and open-source scale-out data center benchmarks that represent the hyperscale behaviors identified in our characterization. We expect our comprehensive study to drive continued benchmarking efforts that represent the severity of overheads in production-grade software.

End-to-end thinking in accelerator design. Oftentimes, when designing accelerators, computer architects tend to miss the end-to-end picture, i.e., overheads that might arise from other system parts [333]. When trying to adopt these accelerators at hyperscale, several hyperscale enterprises often find that these accelerators degrade performance due to overlooked microservice software-induced overheads (e.g., offload-induced overheads due to microservice threading design) [445].

Accelerometer is a simple, powerful tool to help architects analytically estimate software-induced overheads that arise from the end-to-end path, projecting realistic gains early in

[5] Facebook awarded research grants to researchers at Cornell, UT Austin, and MIT, to develop benchmarks that represent the hyperscale behaviors we identified in Section 6.1 [445].

the hardware design phase. To quote an anonymous expert reviewer for the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) 2020, "*Given the time and cost for a full acceleration effort, good models that can inform early design choices are extremely valuable. To my knowledge, there is no alternative model for microservice execution that could serve this purpose.*"

Driving future research. Our characterization in Section 6.1 revealed many new significant overheads. For example, it is unusual for applications to incur such high overheads from logging and memory frees; very few academic studies focus on optimizing them. Our rich characterization of overheads will enable industry and academic researchers to work on mitigating them. Expert ASPLOS reviewers' comments include, "*The data set of microservice overheads presented in this work and the detailed breakdowns will serve as excellent motivation for future research in acceleration of infrastructure operations*" and "*Comprehensive and extremely insightful study of production microservices*".

6.7 Chapter Summary

We summarize our contributions as follows:

- **A comprehensive characterization of microservice leaf function overheads.** We presented a systematic characterization of leaf function overheads experienced by production microservices at Facebook: one of the largest social media platforms today.

- **A detailed study of microservice functionality breakdowns.** We presented detailed microservice functionality breakdowns, identifying orchestration overheads and providing a systematic understanding of hardware acceleration opportunities at hyperscale. Both the leaf function and microservice functionality studies have received recognition in academia and industry, having (1) influenced commercial hardware design, (2) enabled industry-academia joint benchmarking efforts, and (3) improved the software development process [445].

- **Accelerometer**: We introduced an analytical model to project realistic microservice speedup early in the hardware design phase for various hardware acceleration strategies. *Accelerometer* is an analytical alternative to ad hoc hardware customization approaches that helps hyperscale enterprises to make well-informed hardware investments.

- **A detailed demonstration of Accelerometer's utility in production.** We demonstrated *Accelerometer*'s utility in Facebook's production microservices using three retrospective case studies, showing how *Accelerometer* projects realistic performance gains from hardware acceleration.

At global user population scale, important microservices in hyperscale data centers can grow to account for an enormous installed base of servers. With the decline of hardware performance scaling (detailed in Chapter I), successive server generations running key microservices exhibit diminishing performance returns. Hence, it is imperative to understand how important microservices spend their CPU cycles to determine hardware acceleration opportunities across the global server fleet. To this end, we undertook a comprehensive characterization of the top seven microservices that run on the compute-optimized data center fleet at Facebook.

Our characterization revealed that microservices spend as few as 18% of CPU cycles executing the main application logic (e.g., performing an ML inference operation); the remaining cycles are spent in common operations that are not core to the main application logic (e.g., I/O processing, logging, and compression). Accelerating such common building blocks can greatly improve data center performance. Whereas developing specialized hardware acceleration for each building block might be beneficial, it becomes risky at hyperscale if these hardware accelerators do not yield expected gains due to performance bounds precipitated by a microservice's software interaction with the hardware (e.g., offload-induced overheads due to microservice software threading design). To identify such performance bounds early in the hardware design phase, we developed an analytical model for hardware acceleration, *Accelerometer*, that projects realistic speedup in microservices. We validated *Accelerometer*'s utility in production using three retrospective case studies and demonstrated that it estimates the real, production speedup with an error that is less than or equal to $\leq 3.7\%$. We then used *Accelerometer* to project gains from accelerating important common building blocks identified in our characterization.

CHAPTER VII

Future Work and Conclusions

7.1 Future Directions

My long-term research vision is to radically redesign the entire systems stack for computing systems that serve billions of users around the world while facing stringent performance, power, and cost requirements. This dissertation is the first step towards achieving my vision. There are many exciting avenues of future work that follow from the research presented in this dissertation, involving interdisciplinary collaborations with researchers working on algorithms, programming languages, Machine Learning, Human-Computer Interaction, device technologies, embedded systems, computer networks, software systems, and computer architecture. Some of these future research ideas are summarized in this section (broadly sorted from short-term to long-term).

7.1.1 Enabling Cross-Stack Designs for Emerging Web Service Paradigms and Application Domains

This dissertation demonstrates the benefit of cross-stack design to enable the modern web service paradigm of microservices. Apart from microservices, modern web systems are increasingly being built with new service paradigms such as serverless architectures. Each new paradigm introduces unique overheads that affect data center efficiency. For example, unlike microservices, serverless systems introduce additional inefficiencies from container launch and warm-up delays, increased communication, and greater scalability issues. Using some of the techniques developed in my dissertation, future work has the potential to redesign improved hardware and software design primitives to support continuously emerging service paradigms.

Apart from service paradigms, I notice several emerging application domains that are starting to require hyperscale computation. For example, I anticipate the cloud or edge data centers to increasingly start processing data from self-driving cars, connected vehicles, and the Internet of Things (IoT) devices. Efficiently supporting such emerging applications will require rethinking data center software and hardware design. Similar to the work done in this dissertation, I envision the need for future research to analyze how such emerging application classes will begin to use data centers and the overheads they might impose, to design efficient cross-stack optimizations to support them.

7.1.2 Rethinking Hardware-Software Co-Design for System Overheads that Arise at Hyperscale

My characterization of real-world production microservices revealed several system overheads that particularly arise at hyperscale (detailed in Chapters IV and VI). In this dissertation, I presented techniques to mitigate a few predominant overheads such as I-cache misses (Chapter IV) and I/O event notification (Chapter V). As immediate future work, I see myself continuing to systematically analyze the cause of each significant overhead I identified, to develop efficient solutions to mitigate them. As one example, apart from improvements to I/O event notification (Chapter V), I envision an end-to-end I/O processing path that also incorporates hardware-software optimizations to efficiently (1) receive/send a large number of I/O, (2) operate the CPU when waiting for an I/O and (3) process large I/O just as well as small I/O transfers.

7.1.3 Mitigating the Killer Microsecond Problem in Modern Web Services

As my dissertation has shown, modern servers are equipped with mechanisms to effectively hide nanosecond-scale stalls (e.g., OoO cores) and millisecond-scale stalls (e.g., context switching), but lack efficient support to hide microsecond-scale stalls that can critically affect modern web service efficiency. To mitigate these microsecond-scale stalls (often called the "killer microsecond" [368]), I will characterize the impact of various microsecond-scale accesses (e.g., modern networking, non-volatile memories, and accelerator accesses) on application latency and resource efficiency.

To hide killer microseconds, I will design an end-to-end solution spanning the systems stack. First, I will design "microsecond-aware" systems stacks that have reduced lock contention, fast interrupt handling, efficient spin-polling, and improved job scheduling (drawing on techniques from Chapters III and V). Second, I will develop techniques that will keep the CPU busy during a microsecond-scale stall by making the hardware seamlessly switch hardware threads in a super-wide processor. Third, I will design cross-stack solutions that prevent the CPU from being idle when there is insufficient Thread Level Parallelism, by self-navigating fine-grained sleep states at runtime to enable a core to stop consuming power when a microsecond-scale access is outstanding and shift that power to cores not blocked on accesses.

7.1.4 Redesigning Software Stacks for Emerging Hardware Accelerators

As systems researchers, we continue to struggle with abstraction primitives, suggesting that the era of abstraction design innovation is not over. For example, every few years we invent new isolation and abstraction mechanisms, such as processes, virtual machines, trusted execution environments, and containers (detailed in Chapter I). I find that we design these new abstractions more as an afterthought to an emerging application paradigm (e.g., developing containers to suit the needs of the microservice and serverless paradigms).

Rather than redefine OS primitives for each service paradigm, I will identify correct primitives that allow software to seamlessly trade-off isolation with the ability to share data and computational resources. As a starting point, I find that defining such abstractions is particularly pertinent in the modern-day era of designing specialized hardware accelerators for an increasing number of application domains. During my internships at hyperscale companies, I observed that software stack developers spent several months to years developing custom, hardware-specific software stacks for individual accelerators. Rather than redefining software abstractions for each hardware accelerator, I am interested in building OS abstractions for small components (e.g., using program synthesis techniques) that can then be assembled into custom software stacks for myriad novel hardware accelerators. This approach allows for software design exploration and innovation, allowing experimenting with different primitives without building entirely new software stacks.

7.1.5 Designing Systems to Support Emerging Device Technologies

With the decline in hardware performance scaling, recent hardware innovations include a particular focus on emerging device technologies such as Non-Volatile Memory (NVM) [399, 309, 308], 3D memory blocks [322], and optical computing [268]. Each of these device technologies' variants have diverse device properties. As one example, NVM technologies (e.g., CTT, RRAM, STTRAM, and PCM) exhibit a significant diversity in efficiency metrics such as endurance, retention, fault-tolerance, storage density, read latency, write latency, and energy consumption.

As these technologies begin to be introduced in data centers, there is an opportunity to explore their design space to build software frameworks that map diverse service accesses to suitable memories (along the vein of the Soft SKU approach detailed in Chapter IV). For example, RRAM, PCM, and CTT, have Multi-Level Cell (MLC) capabilities, allowing multiple bits to be packed into a single device to further increase density. However, MLCs have poor fault-tolerance. Whereas MLCs are not amenable to most applications due to their poor fault-tolerance, they might particularly benefit Deep Neural Networks (DNNs) since (1) DNNs have large models that must fit in memory, requiring a high storage density and (2) DNNs might be able to tolerate the errors that MLCs impose [400]. Hence, it is important to characterize the trade-off between different NVM device properties and use this characterization to build software frameworks that schedule diverse web service accesses to suitable device technologies.

In a similar vein as the *Accelerometer* model described in Chapter VI, I also envision the need to develop realistic analytical models that estimate the various efficiency implications of leveraging a particular device technology [401]. Going forward, I will develop generalized cross-stack infrastructures to efficiently incorporate emerging device technologies in hyperscale data centers.

7.1.6 Using Machine Learning to Self-Navigate the Hyperscale Design Space

As the hyperscale software/hardware design space continues to become more complex, I foresee empirical systems leveraging recent improvements in ML models to manage design complexity. I am interested in using ML techniques to self-navigate complex software/hardware design spaces such as resource allocation, request scheduling, and bottleneck identification.

7.1.7 Designing Energy-Efficient Data Centers

While this dissertation focuses on improving hyperscale efficiency, there is more work to be done to particularly improve data center *energy efficiency*. The end of Dennard scaling [217, 462] has severely impacted the power consumption of modern hardware systems. Although the hardware industry has continued to develop new power-centric process technologies, the fact still remains that power consumption no longer scales with feature size, resulting in data center systems with increasingly high power envelopes [271]. Today, a growing portion of the data center energy budget is spent on cooling the data center rather than on computations [334]. Hence, rethinking data center cooling technologies has become a deeply important problem.

I am interested in studying emerging data center cooling technologies such as 2-phase immersion cooling [271] to radically reimagine futuristic data centers. Since such systems can ignore thermal boundaries, there is an opportunity to design innovative systems based on newer notions of power density, thermal runways, or the form factor of boards. For example, such an energy-efficient system might particularly improve the nature of computations performed by edge data centers that are heavily power-constrained today.

7.1.8 Making Intersectionality, Equity, and Fairness as First-Order System Design Metrics

While efficiency and security metrics are certainly critical to gauge web systems, I find that as system designers, we must start to think about the societal implications of the web systems we build. I am interested in exploring intersectionality, equity, and fairness as first-order web system design metrics. For example, I observed that data center operators selectively supply responses based on the user's geographical location. If a user is in a remote location (e.g., a remote island) that has poor internet connection, a web system might exploit this fact to supply a slower response to the user, thereby improving the latency headroom in the data center.

This observation made me wonder about the implications of data center operators discriminating responses sent to users based on their age, gender, or occupation in a Wild And Crazy Ideas (WACI) session [443] held in association with the International Conference on Architectural Support for Programming Languages and Operating Systems. To improve the data center's latency headroom, such a discriminatory web system might exploit the notion that an older user might be more patient and willing to wait longer for a web service response, resulting in systems that actively discriminate against users.

I am interested in systematically characterizing the various web system properties and the kind of societal implications they might induce, to propose equity and fairness as first-order system design metrics that system designers must consider before deploying a hyperscale web system. I will also design abstraction frameworks that make system decisions based on such metrics to ensure “unbiased” computation (e.g., data center scheduling systems that screen requests based on laws around age or gender).

7.2 Dissertation Conclusions

The world is undergoing a technological revolution where modern web services such as social media, online messaging, web search, video streaming, and online banking must support billions of users, requiring data centers that scale to hundreds of thousands of servers, i.e., *hyperscale*. While at face value, hyperscale web services seem instantaneously available at the touch of a button, existing hyperscale systems barely meet performance requirements despite running on prohibitively expensive and power-hungry data centers. As hyperscale computation grows to drive increasingly sophisticated applications (e.g., virtual reality, self-driving cars, conversational AI, and the Internet of Things), existing hyperscale systems will face greater efficiency challenges due to these more complex tasks. This dissertation presented technologies that enable tomorrow’s hyperscale web services by designing efficient system stacks for hyperscale computation.

Over the past few years, there has been a radical shift in hyperscale computing due to an unprecedented growth in data, users, and web service functionality. However, modern hardware systems can no longer support this unprecedented growth in hyperscale trends. It is widely agreed that the hardware industry has been facing a steady decline in hardware performance scaling [480]. To enable hyperscale computation requirements despite the decline in hardware performance scaling, hardware architects must become more aware of hyperscale software needs and software researchers can no longer expect unlimited hardware performance scaling. Hence, systems researchers can no longer follow the traditional approach of building each layer of the systems stack separately. To enable hyperscale computation, it is extremely critical that systems researchers rethink the synergy between the software and hardware worlds from the ground up. Techniques presented in this dissertation establish the synergy between the software and hardware worlds to enable futuristic hyperscale web services. Specifically, this dissertation (1) designed software that is aware of new hardware constraints and (2) designed commodity and custom hardware to efficiently support new hyperscale software requirements.

Beyond the software and hardware paradigms considered in this dissertation, challenges faced by new application domains (e.g., the Internet of Things), service paradigms (e.g., serverless computation), and hardware technologies (e.g., Non-Volatile Memory technologies) could be addressed through an extension of techniques presented in this dissertation. As one example, NVM technologies (1) exhibit microsecond-scale access latencies [369], (2) face diversity in device properties [400], and (3) are being incorporated in modern hardware accelerators [129]. Techniques introduced in this dissertation (μ*Tune* and μ*Notify*) could be used to design cross-stack solutions that enable efficient communication with NVM devices that face microsecond-scale access latencies. Furthermore, this dissertation's contributions could be used to map diverse microservice accesses to suitable NVM devices based on device technology properties (*SoftSKU*), while analytically modeling the implications of such accesses to design more efficient NVM-based hyperscale systems (*Accelerometer*).

This dissertation makes a number of novel software and hardware contributions that bridge the software and hardware worlds to enable the hyperscale web services of tomorrow. Rather than following the traditional approach of building each layer of the systems stack separately, this dissertation uniquely brings new hardware insights when designing software stack layers and draws on fundamental software design principles to systematically architect the hardware layer.

First, this dissertation's software contributions in terms of a representative, open-source benchmark suite of modern web services facilitate future research on a prominent application design paradigm that will increasingly be employed in future hyperscale services. Using this benchmark suite, this dissertation built on decades of software threading model research, identifying gaps in existing software threading designs that arise as a consequence of the recent decline in hardware performance scaling. This dissertation presented new software threading model insights that enabled fundamentally redesigning software threading models for emerging hyperscale service paradigms.

Second, this dissertation's hardware contributions systematically architect data center hardware in a way that is aware of fundamental software design principles to support the unprecedented growth in hyperscale software trends. By comprehensively characterizing the commodity and custom hardware design space in light of emerging hyperscale trends, this dissertation facilitates a holistic approach to future hardware design. This characterization has influenced the design of commercial hardware architectures, enabled industry-academia joint benchmarking efforts, and improved the software development process. My characterization's insights enabled techniques that helped maintain the performance improvement rate for commodity processors, triggering a significant shift in the hardware industry, saving millions of dollars, and meaningfully reducing the global carbon footprint. Furthermore, driven by this characterization, this dissertation presented a rigorous, analytical alternative to ad hoc hardware customization approaches that enabled hyperscale enterprises to make well-informed hardware decisions.

More broadly, this dissertation's success in bridging the software and hardware worlds paves the way for fresh approaches to hyperscale computing throughout the hardware-software stack. Overall, this dissertation makes the following contributions:

- As a part of this dissertation's software contributions, Chapter II presented *μSuite*, the first open-source benchmark suite of end-to-end modern web services composed using the emerging microservice application paradigm. By demonstrating how *μSuite* can be used to study new hyperscale overheads that particularly arise from threading interactions with the underlying OS and network stacks, this dissertation facilitates future research, with *μSuite* being used by researchers in academia and industry (e.g., at MIT, UIUC, UT Austin, Georgia Tech, Cornell, ARM, and Intel) to analyze modern web services.

- Driven by Chapter II's observations on software threading-induced overheads in the microservice regime, Chapter III presented a systematic taxonomy of microservice software threading models, analyzing them to identify new insights in the age-old research area of software threading. This threading taxonomy and its systematic analysis enables expert and novice developers alike to guide their microservice threading designs. Based on this threading analysis, this chapter makes the important and non-obvious observation that that no single threading model is best across all hyperscale load conditions, paving the way for an automated approach and associated tool, *μTune*, that redesigns threading and concurrency paradigms for hyperscale microservices. *μTune* abstracts threading design from microservice code and automatically adapts to time-varying service load by intelligently tuning threading models and thread pool sizes during system runtime.

- As a part of this dissertation's hardware contributions, Chapter IV took a step towards identifying how we should build commodity data center hardware in the post-Moore era. Several architects today work on developing specialized hardware accelerators for key domains (e.g. Machine Learning tasks). Rather than following this prohibitively expensive approach, this chapter instead took a step back and systematically answered the following question: can we extract greater performance from cost-efficient commodity hardware to maintain server-class processors' performance improvement rate, despite the decline in hardware performance scaling? Chapter IV characterized the shortcomings in commodity hardware running hyperscale microservices, identifying hardware design opportunities that influenced commercial server-class processor architectures. Driven by this characterization, Chapter IV presented an automated design approach and tool, Soft SKU, that configures OS and hardware knobs to make an existing commodity processor more performance efficient for a given real-world, production microservice. Soft SKUs demonstrated how to extract greater performance from cheap commodity hardware, resulting in hyperscale enterprises prioritizing this approach's adoption over the modern-day trend of customizing hardware, and triggering a shift in the hardware industry. Furthermore, Chapter IV demonstrated how Soft SKUs achieve significantly greater performance efficiency than stock and expert-tuned server configurations when running production microservices that serve billions of users, saving millions of dollars and meaningfully reducing the global carbon footprint.

- In a similar vein to Chapter IV's goal of maintaining commodity processors' performance improvement rate, Chapter V demonstrated that to overcome new hyperscale overheads, existing hardware mechanisms can intelligently be used to redesign commodity server architectures with minimal hardware enhancements. In particular, to mitigate new hyperscale overheads that arise from I/O event notification, this chapter analyzed the limits of existing I/O notification paradigms and used the analysis's insights to present *µNotify*, the first I/O notification paradigm that achieves scalable, near-constant time I/O notification. *µNotify* uses commodity processors' cache coherence invalidation messages more intelligently and introduces a small enhancement to the cache coherence controller to reduce I/O notification overheads by extracting greater performance from commodity server-class hardware.

- While Chapters IV and V focused on extracting greater performance from commodity server-class hardware, Chapter VI focused on developing and deploying new custom hardware (particularly in the form of hardware accelerators) in a well-informed manner. Since designing custom hardware for every microservice is prohibitively expensive, two important questions arise: (1) Which microservice software operations consume the most CPU cycles and are worth accelerating? (2) How much can the accelerator realistically improve its targeted microservice overhead? To answer both questions, this chapter first undertook a comprehensive characterization of hardware acceleration opportunities at hyperscale that influenced commercial hardware design. My characterization tool has been integrated into Facebook's fleet-wide performance monitoring infrastructure, assimilating statistics from servers globally to help developers visualize the performance impact of their code changes at hyperscale [445]. Additionally, my characterization has resulted in a joint industry-academia benchmarking effort to develop scale-out applications that represent the hyperscale behaviors I identified [445]. Driven by this characterization's insights, this chapter then presented an analytical model, *Accelerometer*, that estimates realistic gains from hardware acceleration early in the hardware design phase. *Accelerometer*'s generality has resulted in multiple hyperscale enterprises and hardware vendors adopting it to make well-informed hardware decisions [445].

Overall, the combination of techniques introduced in this dissertation improve the performance, scalability, energy efficiency, and cost of operation of the next generation of hyperscale computing systems. This work bridges the software and hardware worlds, demonstrating the importance of that bridge in *enabling the hyperscale web services of tomorrow* via efficient solutions that span the systems stack. By realizing efficient web services from analytical models on paper to system deployment at hyperscale, this dissertation bridges the gap between hyperscale growth expectations and today's hardware reality.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A Brief History of Containers. https://d2iq.com/blog/brief-history-containers.

[2] A Brief History of Microservices. https://www.dataversity.net/a-brief-history-of-microservices/.

[3] About DPDK. https://www.dpdk.org/about/.

[4] Accelerometer. https://doi.org/10.5281/zenodo.3612796.

[5] Accelerometer & SoftSKU: Improving HW performance for diverse microservices. https://engineering.fb.com/data-center-engineering/accelerometer-and-softsku/.

[6] Add reco. to website. www.easyrec.org. [Accessed 4/27/2018].

[7] Adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[8] Advanced Encryption Standard (AES). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[9] Aerospike. https://www.aerospike.com/docs/client/java/usage/async/index.html.

[10] Allocation optimization with different block-sized allocation maps. https://patents.google.com/patent/US5481702A/en.

[11] Analytical model predicts exactly how much a piece of hardware will speed up data centers. https://news.engin.umich.edu/2020/04/analytical-model-predicts-exactly-how-much-a-piece-of-hardware-will-speed-up-data-centers/.

[12] Analytical model predicts how much a piece of hardware will speed up data centers. https://techxplore.com/news/2020-04-analytical-piece-hardware-centers.html.

[13] Apache http server project. https://httpd.apache.org/.

[14] Average number of search terms for online search queries in the United States as of August 2017. https://www.statista.com/statistics/269740/number-of-search-terms-in-internet-research-in-the-us/.

[15] Avx. www.wikipedia.org/wiki/Advanced_Vector_Extensions.

[16] Azure Synchronous I/O antipattern. https://docs.microsoft.com/en-us/azure/architecture/resiliency/high-availability-azure-applications.

[17] The biggest thing amazon got right: The platform. https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/.

[18] BLPOP key timeout. https://redis.io/commands/blpop.

[19] Bob Jenkins. SpookyHash: a 128-bit noncryptographic hash. http://burtleburtle.net/bob/hash/spooky.html.

[20] Bpftrace. https://github.com/iovisor/bpftrace.

[21] Building products at soundcloud: Dealing with the monolith. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith.

[22] Building Scalable and Resilient Web Applications on Google Cloud Platform. https://cloud.google.com/solutions/scalable-and-resilient-apps.

[23] Celery: Distributed Task Queue. http://www.celeryproject.org/.

[24] Chasing the bottleneck: True story about fighting thread contention in your code. https://blogs.mulesoft.com/biz/news/chasing-the-bottleneck-true-story-about-fighting-thread-contention-in-your-code/.

[25] Code and Data Prioritization - Introduction and Usage Models in the Intel Xeon Processor E5 v4 Family. https://software.intel.com/en-us/articles/introduction-to-code-and-data-prioritization-with-usage-models.

[26] Collaborative filtering via matrix decomposition in mlpack. www.ratml.org/pub/pdf/2015collaborative.pdf. [Accessed 4/27/2018].

[27] Containers. https://a16z.com/2015/01/22/containers/.

[28] Data Age 2025: the datasphere and data-readiness from edge to core. https://www.i-scoop.eu/big-data-action-value-context/data-age-2025-datasphere/.

[29] David Riddoch on Bypassing the Kernel and Hypervisor for Network I/O, Solarflare, OpenOnload. https://www.infoq.com/interviews/riddoch-kernel-bypass-solarflare/.

[30] Dawn of the data center operating system. https://www.infoworld.com/article/2906362/dawn-of-the-data-center-operating-system.html.

[31] Digital 2020: 3.8 Billion People Use Social Media. https://wearesocial.com/blog/2020/01/digital-2020-3-8-billion-people-use-social-media.

[32] Disclosure of H/W prefetcher control on some Intel processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[33] Emon user's guide. https://software.intel.com/en-us/download/emon-user-guide.

[34] Envoy. https://www.envoyproxy.io/.

[35] Explicit os support for hardware threads. http://www.barrelfish.org/publications/ma-apoenaru-hwthreads.pdf.

[36] Faban. http://faban.org. [Accessed 27-Apr-2018].

[37] Facebook Thrift. https://github.com/facebook/fbthrift.

[38] Fighting spam with haskell. https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/.

[39] Finagle. https://twitter.github.io/finagle/guide/index.html.

[40] Finding first set bit. https://en.wikipedia.org/wiki/Find_first_set.

[41] Flexible notification mechanism for user-level interrupts. https://patents.google.com/patent/US8285904.

[42] From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. https://www.infoq.com/presentations/linkedin-microservices-urn.

[43] Google Says the SoC is the New Motherboard. https://www.nextplatform.com/2021/03/22/google-says-the-soc-is-the-new-motherboard/.

[44] Google Search Statistics. http://www.internetlivestats.com/google-search-statistics/.

[45] gRPC. https://github.com/heathermiller/dist-prog-book/blob/master/chapter/1/gRPC.md.

[46] Handling 1 Million Requests per Minute with Go. http://marcio.io/2015/07/handling-1-million-requests-per-minute-with-golang/.

[47] Hapiger. https://github.com/grahamjenson/hapiger. [Accessed 4/27/2018].

[48] Hidden Costs of Memory Allocation. https://randomascii.wordpress.com/2014/12/10/hidden-costs-of-memory-allocation/.

[49] IBM Archives: IBM Mainframes. https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_intro2.html.

[50] Improve Application Performance With SwingWorker in Java SE 6. http://www.oracle.com/technetwork/articles/javase/swingworker-137249.html.

[51] Intel and Micron Produce Breakthrough Memory Technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[52] Intel delays its 10-nanometer 'Cannon Lake' CPUs yet again. https://www.engadget.com/2018-04-27-intel-delays-cannon-lake-chips-again.html.

[53] Intel dpt with aes-ni & secure key. www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard-aes/data-protection-aes-general-technology.html.

[54] Intel i/o at. www.intel.com/content/www/us/en/wireless-network/accel-technology.html.

[55] Intel knl. https://ark.intel.com/content/www/us/en/ark/products//knightslanding.html.

[56] Intel Memory Latency Checker v3.6. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[57] Intel optane technology, howpublished="http://www.intel.com/optane/".

[58] Intel resource director technology (rdt) in linux. https://01.org/intel-rdt-linux.

[59] Internet Users and Usage. https://psu.pb.unizin.org/ist110/chapter/1-4-internet-users-and-usage/.

[60] Introduction to Monolithic Architecture and MicroServices Architecture. https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63.

[61] Is Data Really the New Oil in the 21st Century? https://towardsdatascience.com/is-data-really-the-new-oil-in-the-21st-century-17d014811b88.

[62] John Hennessy and David Patterson Deliver Turing Lecture at ISCA 2018. https://www.acm.org/hennessy-patterson-turing-lecture.

[63] Key Components of a Software Defined Data Center. https://www.evolvingsol.com/2018/04/17/components-software-defined-data-center/.

[64] Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/blog/2009/7/25/latency-iseverywhere-and-it-costs-you-sales-how-to-crush-it.html.

[65] Let's look at Dispatch Timeout Handling in WebSphere Application Server for z/OS. www.ibm.com/developerworks/community/blogs/aimsupport/entry/dispatch_timeout_handling_in_websphere_application_server_for_zos.

[66] Linux bcc/BPF Run Queue (Scheduler) Latency. http://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html.

[67] LPOP key. https://redis.io/commands/lpop.

[68] Mahout. http://mahout.apache.org/. [Accessed 4/27/2018].

[69] Mcrouter. https://github.com/facebook/mcrouter.

[70] Memcached performance. https://github.com/memcached/memcached/wiki/Performance.

[71] Microsoft Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[72] Mips cache line locking. https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00904-2B-interAptiv-SUM-02.01.pdf.

[73] mongoDB. https://www.mongodb.com/.

[74] Monolithic application suites have dominated enterprise IT landscapes for the last 20 years. So why have they gone out of fashion? https://www.capgemini.com/2019/06/monolith-to-microservices-an-integration-journey/.

[75] Myrocks: A space- and write-optimized MySQL database. https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/.

[76] OpenImages: A public dataset for large-scale multi-label and multi-class image classification. https://github.com/openimages.

[77] OpenSSL & SSL/TLS Toolkit. https://www.openssl.org/.

[78] PerfKit Benchmarker. https://github.com/GoogleCloudPlatform/PerfKitBenchmarker.

[79] Pokemon go now the biggest mobile game in US history. http://www.cnbc.com/2016/07/13/pokemon-go-now-the-biggest-mobile-game-in-us-history.html.

[80] The poll system call. https://www.usenix.org/legacy/events/usenix99/full_papers/banga/banga_html/node4.html.

[81] The power of the proxy: Request routing memcached. https://dzone.com/articles/the-power-of-the-proxy-request-routing-memcached. [Accessed 4/27/2018].

[82] Pred.io. http://predictionio.apache.org/index.html. [Accessed 4/27/2018].

[83] Programmer's Guide, Release 2.0.0. https://www.intel.com/content/dam/www/public/us/en/documents/guides/dpdk-programmers-guide.pdf.

[84] Protocol Buffers. https://developers.google.com/protocol-buffers/.

[85] Raccoon. www.npmjs.com/package/raccoon. [Accessed 4/27/2018].

[86] Real-time AI: Microsoft announces preview of Project Brainwave. https://blogs.microsoft.com/ai/build-2018-project-brainwave/.

[87] Real World Technologies. https://www.realworldtech.com/forum/\?threadid=185536&curpostid=185536.

[88] Redis. https://redis.io/.

[89] Redis Replication. https://redis.io/topics/replication.

[90] Researchers from Facebook has designed a way to measure exactly how much a hardware accelerator would speed up a datacenter. https://debuglies.com/2020/04/08/researchers-from-facebook-has-designed-a-way-to-measure-exactly-how-much-a-hardware-accelerator-would-speed-up-a-datacenter/.

[91] Resque. https://github.com/defunkt/resque.

[92] RQ. http://python-rq.org/.

[93] Samsung Z-SSD. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[94] Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture. https://www.infoq.com/presentations/scale-gilt.

[95] Seldon. www.seldon.io/. [Accessed 4/27/2018].

[96] Serverless on AWS. https://aws.amazon.com/serverless/.

[97] Setting Up Internal Load Balancing. https://cloud.google.com/compute/docs/load-balancing/internal/.

[98] Similar Images graduates from Google Labs. https://googleblog.blogspot.com/2009/10/similar-images-graduates-from-google.html.

[99] Simple object access protocol (SOAP) 1.1. https://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[100] The evolution of data centers / data center development across the decades. https://www.yondrgroup.com/yondr-intel/the-evolution-of-cloud-data-centers-and-beyond/.

[101] The Evolution of Social Media: How Did It Begin, and Where Could It Go Next? https://online.maryville.edu/blog/evolution-social-media/.

[102] The Top 12 Future Web Development Trends in 2021. https://dev.to/adhyaswarnali/the-top-12-future-web-development-trends-in-2021-25k5.

[103] Unlock system performance in dynamic environments. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[104] Using tracing at facebook scale. https://tracingsummit.org/w/images/6/6f/TracingSummit2014-Tracing-at-Facebook-Scale.pdf.

[105] VMWare Timeline. https://www.vmware.com/timeline.html.

[106] What is microservices architecture? https://smartbear.com/learn/api-design/what-are-microservices/.

[107] What is SOA, or service-oriented architecture? https://www.ibm.com/cloud/learn/soa.

[108] What's causing the exponential growth of data? https://insights.nikkoam.com/articles/2019/12/whats_causing_the_exponential.

[109] What's causing the exponential growth of data? https://insights.nikkoam.com/articles/2019/12/whats_causing_the_exponential.

[110] Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Plagiarism&oldid=5139350.

[111] With long history of virtualization behind it, IBM looks to the future. https://www.networkworld.com/article/2254433/with-long-history-of-virtualization-behind-it--ibm-looks-to-the-future.html.

[112] Workers inside unit tests. http://python-rq.org/docs/testing/.

[113] x86/umwait: Enable user wait instructions. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[114] Zero-copy tcp receive. https://lwn.net/Articles/752188/.

[115] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *Computing Research Repository*, 2016.

[116] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. *Data Compression Accelerator on IBM POWER9 and Z15 Processors*. 2020.

[117] Tarek F Abdelzaher and Nina Bhatti. Web server QoS management by adaptive content delivery. In *International Workshop on Quality of Service*, 1999.

[118] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The hiphop virtual machine. In *Acm Sigplan Notices*, 2014.

[119] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[120] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference*, 2008.

[121] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. Remote memory in the age of fast networks. In *Symposium on Cloud Computing*, 2017.

[122] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *International Symposium on Computer Architecture*, 2015.

[123] Amitanand S Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. Storage infrastructure behind Facebook messages: Using HBase at scale. *IEEE Data Eng. Bull.*, 2012.

[124] Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal, and Daniel A Jiménez. Exploring predictive replacement policies for instruction cache and branch target buffer. In *International Symposium on Computer Architecture*, 2018.

[125] Hakan Akkan, Michael Lang, and Lorie M Liebrock. Stepping towards noiseless linux environment. In *International workshop on runtime and operating systems for supercomputers*, 2012.

[126] Alexa. Alexa, the web information company. `http://www.alexa.com/`. [Accessed 27-Apr-2018].

[127] Muhammad Shoaib Bin Altaf and David A. Wood. LogCA: A High-Level Performance Model for Hardware Accelerators. In *International Symposium on Computer Architecture*, 2017.

[128] Jose M Alvarez and Mathieu Salzmann. Compression-aware training of deep networks. In *Advances in Neural Information Processing Systems*, 2017.

[129] Stefano Ambrogio, Pritish Narayanan, Hsinyu Tsai, Robert M Shelby, Irem Boybat, Carmelo Di Nolfo, Severin Sidler, Massimo Giordano, Martina Bodini, Nathan CP Farinha, Benjamin Killeen, Christina Cheng, Yassine Jaoudi, and Geoffrey W. Burr. Equivalent-accuracy accelerated neural-network training using analogue memory. *Nature*, 558(7708):60–67, 2018.

[130] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Symposium on Operating Systems Principles*, 2009.

[131] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems*, 1990.

[132] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *IEEE Symposium on Foundations of Computer Science*, 2006.

[133] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and Optimal LSH for Angular Distance. In *Advances in Neural Information Processing Systems*. 2015.

[134] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, and Kaushik Roy. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[135] Ioannis Arapakis, Xiao Bai, and B. Barla Cambazoglu. Impact of Response Latency on User Behavior in Web Search. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2014.

[136] Andrea Arcangeli. Transparent hugepage support. *KVM forum*, 2010.

[137] Patroklos Argyroudis and Chariton Karamitas. Exploiting the jemalloc memory allocator: Owning Firefox's heap. *Blackhat USA*, 2012.

[138] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. *Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley*, 2006.

[139] Manu Awasthi. Rethinking Design Metrics for Datacenter DRAM. In *International Symposium on Memory Systems*, 2015.

[140] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. Memory Hierarchy for Web Search. In *International Symposium on High Performance Computer Architecture*, 2018.

[141] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *International Symposium on Computer Architecture*, 2019.

[142] Eytan Bakshy, Solomon Messing, and Lada A Adamic. Exposure to ideologically diverse news and opinion on Facebook. *Science*, 2015.

[143] Nikhil Bansal, Kedar Dhamdhere, Jochen Könemann, and Amitabh Sinha. Non-clairvoyant scheduling for minimizing mean slowdown. *Algorithmica*, 2004.

[144] Mahmoud Barhamgi, Djamal Benslimane, and Brahim Medjahed. A query rewriting approach for web service composition. *IEEE Transactions on Services Computing*, 2010.

[145] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 2017.

[146] Luiz Andre Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. In *IEEE Micro*, 2003.

[147] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *ACM SIGARCH Computer Architecture News*, 1998.

[148] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 2007.

[149] Luiz André Barroso and Urs Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[150] Matěj Bartík, Sven Ubik, and Pavel Kubalik. LZ4 compression algorithm on FPGA. In *Electronics, Circuits, and Systems*, 2015.

[151] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH forest: self-tuning indexes for similarity search. In *International conference on World Wide Web*, 2005.

[152] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[153] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[154] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX Conference on Operating Systems Design and Implementation*, 2014.

[155] S Berchtold, DA Keim, and HP Kriegel. An index structure for high-dimensional data. *Readings in multimedia computing and networking*, 2001.

[156] Stefan Berchtold, Christian Bohm, Hosagrahar V Jagadish, H-P Kriegel, and Jörg Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *International Conference on Data Engineering*, 2000.

[157] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. *Reconsidering custom memory allocation*. ACM, 2002.

[158] Tom Berson, Drew Dean, Matt Franklin, Diana Smetters, and Michael Spreitzer. Cryptography as a network service. In *Network and Distributed System Security Symposium*, 2001.

[159] Srikant Bharadwaj, Guilherme Cox, Tushar Krishna, and Abhishek Bhattacharjee. Scalable Distributed Shared Last-Level TLBs Using Low-Latency Interconnects. In *International Symposium on Microarchitecture*, 2018.

[160] L Bharathi, N Sangeetha Priya, BS Sathish, A Ranganayakulu, and S Jagan Mohan Rao. Burst rate based optimized io queue management for improved performance in optical burst switching networks. In *International Conference on Advanced Computing & Communication Systems*, 2019.

[161] Abhishek Bhattacharjee. Translation-Triggered Prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[162] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[163] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *arXiv preprint arXiv:1504.01048*, 2015.

[164] Filip Blagojevic, Dimitrios S Nikolopoulos, Alexandros Stamatakis, Christos D Antonopoulos, and Matthew Curtis-Maury. Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems. *Parallel Computing*, 2007.

[165] Christopher James Blythe, Gennaro A Cuomo, Erik A Daughtrey, and Matt R Hogstrom. Dynamic thread pool tuning techniques. *Google Patents*, 2007.

[166] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computer Survey*, 2001.

[167] Shekhar Borkar and Andrew A Chien. The future of microprocessors. *Communications of the ACM*, 2011.

[168] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, and Samuel Rash. Apache Hadoop goes realtime at Facebook. In *International Conference on Management of data*, 2011.

[169] Anna Bouch, Nina Bhatti, and Allan Kuchinsky. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *ACM Conference on Human Factors and Computing Systems*, 2000.

[170] Anne Bracy, Kshitij Doshi, and Quinn Jacobson. Disintermediated active communication. *IEEE Computer Architecture Letters*, 2006.

[171] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, 2013.

[172] Jianhua Cao, Mikael Andersson, Christian Nyberg, and Maria Kihl. Web server performance modeling using an m/g/1/k* ps queue. In *International Conference on Telecommunications*. IEEE.

[173] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *IEEE/ACM International Symposium on Microarchitecture*, 2016.

[174] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. *ACM SIGARCH Computer Architecture News*, 2006.

[175] Guang-Ho Cha and Chin-Wan Chung. The gc-tree: a high-dimensional index structure for similarity search in image databases. *IEEE transactions on multimedia*, 2002.

[176] Kevin Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *International Conference on Measurement and Modeling of Computer Science*, 2016.

[177] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *International Symposium on Code Generation & Optimization*, 2016.

[178] Doris Chen and Deshanand Singh. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In *Design Automation Conference*, 2013.

[179] Derek Chiou. The microsoft catapult project. In *International Symposium on Workload Characterization*, 2017.

[180] Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman, and Nima Honarmand. Taming the Killer Microsecond. In *International Symposium on Microarchitecture*, 2018.

[181] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Maleen Abeydeera, Logan Adams, Hari Angepat, Christian Boehn, Derek Chiou, Oren Firestein, Alessandro Forin, Kang Su Gatlin, Mahdi Ghandi, Stephen Heil, Kyle Holohan, Ahmad El Husseini, Tamas Juhasz, Kara Kagi, Ratna K. Kovvuri, Sitaram Lanka, Friedel van Megen, Dima Mukhortov, Prerak Patel, Brandon Perez, Amanda Rapsang, Steven Reinhardt, Bita Rouhani, Adam Sapek, Raja Seera, Sangeetha Shekar, Balaji Sridharan, Gabriel Weisz, Lisa Woods, Phillip Yi Xiao, Dan Zhang, Ritchie Zhao, and Doug Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018.

[182] Eric Chung, Peter Milder, James Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *International symposium on microarchitecture*, 2010.

[183] Jaewoong Chung and Karin Strauss. User-level interrupt mechanism for multi-core architectures, 2012.

[184] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ACM SIGARCH Computer Architecture News*, 2013.

[185] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing*, 2010.

[186] Guilherme Cox and Abhishek Bhattacharjee. Efficient Address Translation for Architectures with Multiple Page Sizes. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[187] Travis Craig. Building fifo and priority queuing spin locks from atomic swap. *Technical Report*, 1993.

[188] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. LogP: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*, 1993.

[189] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 2013.

[190] Matthew Curtis-Maury, James Dzierwa, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Annual International conference on Supercomputing*, 2006.

[191] Bruno Da Silva, An Braeken, Erik H D'Hollander, and Abdellah Touhafi. Performance modeling for FPGAs: extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing*, 2013.

[192] Mayank Daga, Ashwin M Aji, and Wu-chun Feng. On the efficacy of a fused CPU+ GPU processor (or APU) for parallel computing. In *Application Accelerators in High-Performance Computing*, 2011.

[193] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. RPCValet: NI-Driven Tail-Aware Balancing of us-Scale RPCs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[194] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. *ACM SIGARCH Computer Architecture News*, 2007.

[195] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, and James Alexander Docauer. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[196] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. CPU DB: Recording Microprocessor History. *Communications of the ACM*, 2012.

[197] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive Hashing Scheme Based on P-stable Distributions. In *Annual Symposium on Computational Geometry*, 2004.

[198] Jeffrey Dean and Luiz Andre Barroso. The Tail at Scale. *Communications of the ACM*, 2013.

[199] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[200] Carlo C Del Mundo, Vincent T Lee, Luis Ceze, and Mark Oskin. NCAM: Near-Data Processing for Nearest Neighbor Search. In *International Symposium on Memory Systems*, 2015.

[201] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[202] Christina Delimitrou and Christos Kozyrakis. Amdahl's law for tail latency. *Communications of the ACM*, 2018.

[203] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[204] Hu Ding, Yu Liu, Lingxiao Huang, and Jian Li. K-means clustering with distributed dimensions. In *Proceedings of The 33rd International Conference on Machine Learning*, 2016.

[205] Huping Ding, Yun Liang, and Tulika Mitra. Wcet-centric partial instruction cache locking. In *DAC Design Automation Conference 2012*, 2012.

[206] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+ abort: A timer-free high-precision l3 cache attack using intel TSX. In *USENIX Security Symposium*, 2017.

[207] Namiot Dmitry and Sneps-Sneppe Manfred. On micro-services architecture. *International Journal of Open Information Technologies*, 2014.

[208] James Donald and Margaret Martonosi. Techniques for Multicore Thermal Management: Classification and New Exploration. In *International Symposium on Computer Architecture*, 2006.

[209] Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling LSH for performance tuning. In *ACM conference on Information and knowledge management*, 2008.

[210] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*, 2012.

[211] Constantinos Dovrolis, Brad Thayer, and Parameswaran Ramanathan. Hip: hybrid interrupt-polling for the network interface. *SIGOPS Operating Systems Review*, 2001.

[212] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: new microarchitecture code-named skylake. *IEEE Micro*, 2017.

[213] Han-cong DUAN, Xian-liang LU, and Jie SONG. Analysis and design of communication server based on epoll and sped. *Computer Applications*, 2004.

[214] Jose Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Orti. Performance of CUDA virtualized remote GPUs in high performance clusters. In *Parallel Processing*, 2011.

[215] Paul Emmerich, Maximilian Pudelko, Simon Bauer, and Georg Carle. User Space Network Drivers. In *Proceedings of the Applied Networking Research Workshop*, 2018.

[216] Deniz Ersoz, Mazin S Yousif, and Chita R Das. Characterizing network traffic in a cluster-based, multi-tier data center. In *International Conference on Distributed Computing Systems*, 2007.

[217] Hadi Esmaeilzadeh, Emily Blem, Renee Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon & the End of Multicore Scaling. In *International Symposium on Computer Architecture*, 2011.

[218] Babak Falsafi and Thomas F Wenisch. A primer on hardware prefetching. *Synthesis Lectures on Computer Architecture*, 2014.

[219] Qi Fan and Qingyang Wang. Performance comparison of web servers with different architectures: a case study using high concurrency workload. In *IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2015.

[220] Dror G Feitelson. A survey of scheduling in multiprogrammed parallel systems. *IBM Research Division*, 1994.

[221] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[222] Xiaoli Zhang Fern and Carla E Brodley. Random projection for high dimensional data clustering: A cluster ensemble approach. In *ICML*, 2003.

[223] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004.

[224] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Field-Programmable Custom Computing Machines*, 2015.

[225] Eitan Frachtenberg. Reducing query latencies in web search using fine-grained parallelism. *World Wide Web*, 2009.

[226] Philip Werner Frey and Gustavo Alonso. Minimizing the Hidden Cost of RDMA. In *Distributed Computing Systems*, 2009.

[227] Borivoje Furht and Armando Escalante. *Handbook of cloud computing*. Springer, 2010.

[228] Aditya Sanjay Gadre, Kaustubh Kabra, Ashwin Vasani, and Keshav Darak. X-xen: huge page support in xen. In *Linux Symposium*, 2011.

[229] Yu Gan and Christina Delimitrou. The Architectural Implications of Cloud Microservices. *IEEE Computer Architecture Letters*, 2018.

[230] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 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.

[231] GR Gao, HHJ Hum, KB Theobald, Xin-Min Tian, and O Maquelin. Polling watchdog: Combining polling and interrupts for efficient message handling. In *International Symposium on Computer Architecture*, 1996.

[232] Bogdan Georgescu, Ilan Shimshoni, and Peter Meer. Mean shift based clustering in high dimensions: A texture classification example. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, 2003.

[233] Pawel Gepner and Michal Filip Kowalik. Multi-core processors: New way to achieve high system performance. In *International Symposium on Parallel Computing in Electrical Engineering*, 2006.

[234] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *International Conference on Very Large Data Bases*, 1999.

[235] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. Persistency for synchronization-free regions. In *Programming Language Design and Implementation*, 2018.

[236] Vaibhav Gogte, Aasheesh Kolli, Michael J Cafarella, Loris D'Antoni, and Thomas F Wenisch. HARE: Hardware accelerator for regular expressions. In *International Symposium on Microarchitecture*, 2016.

[237] Herman H Goldstine and Adele Goldstine. The Electronic Numerical Integrator and Computer (ENIAC). *Mathematical Tables and Other Aids to Computation*, 2(15):97–110, 1946.

[238] Hossein Golestani, Amirhossein Mirhosseini, and Thomas F. Wenisch. Software data planes: You can't always spin to win. In *Proceedings of the ACM Symposium on Cloud Computing*, 2019.

[239] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. Evolution of the samsung exynos CPU microarchitecture. In *International Symposium on Computer Architecture*, 2020.

[240] Rachid Guerraoui and Michał Kapałka. Principles of transactional memory. *Synthesis Lectures on Distributed Computing*, 2010.

[241] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015.

[242] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[243] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *International Symposium on Microarchitecture*, 2017.

[244] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *International Symposium on Computer Architecture*, 2009.

[245] F. Maxwell Harper and Joseph A. Konstan. The Movielens Datasets: History and Context. *ACM Tranactions on Interactive Intelligent Systems*, 2015.

[246] Johann Hauswald, Yiping Kang, Michael A. Laurenzano, Quan Chen, Cheng Li, Ronald Dreslinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Djinn and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers. In *International Symposium on Computer Architecture*, 2015.

[247] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Liv, Austin Rovinski, Arjun Khurana, Ron Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015.

[248] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, and Aditya Kalro. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *International Symposium on High Performance Computer Architecture*, 2018.

[249] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. Practical Lessons from Predicting Clicks on Ads at Facebook. In *International Workshop on Data Mining for Online Advertising*, 2014.

[250] Yuxiong He, Wen-Jing Hsu, and Charles E Leiserson. Provably efficient online nonclairvoyant adaptive scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[251] Mark Hempstead, Gu-Yeon Wei, and David Brooks. Navigo: An early-stage model to study power-contrained architectures & specialization. In *Workshop on Modeling, Benchmarking, and Simulations*, 2009.

[252] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[253] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comp. Arch. News*, 2006.

[254] Eric N Herness, Rob J High, and Jason R McGee. Websphere Application Server: A foundation for on demand computing. *IBM Systems Journal*, 2004.

[255] Mark Hill and Vijay Janapa Reddi. Gables: A Roofline Model for Mobile SoCs. In *High Performance Computer Architecture*, 2019.

[256] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[257] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Sleepy sloth: Threads as interrupts as threads. In *Real-Time Systems Symposium*, 2011.

[258] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: A Network Programming Interface for Non-Volatile Main Memory. In *USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[259] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *SIGARCH Computer Architecture News*, 2009.

[260] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. In *SIGARCH Computer Architecture News*, 2010.

[261] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Lingjia Tang, Jason Mars, and Ron Dreslinski. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *International Symposium on High Performance Computer Architecture*, 2015.

[262] James Hu, Irfan Pyarali, and Douglas C Schmidt. Applying the proactor pattern to high-performance web servers. In *International Conference on Parallel and Distributed Computing and Systems*, 1998.

[263] James C. Hu and Douglas C. Schmidt. JAWS: A Framework for High-performance Web Servers. In *In Domain-Specific Application Frameworks: Frameworks Experience by Industry*, 1999.

[264] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *USENIXAnnual Technical Conference*, 2015.

[265] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *ACM Workshop on Hot Topics in Networks*, 2019.

[266] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *ACM Symposium on Theory of Computing*, 1998.

[267] CAT Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *Intel Corporation, April*, 2015.

[268] Jürgen Jahns and Sing H Lee. *Optical Computing Hardware: Optical Computing*. Academic press, 2014.

[269] Akanksha Jain and Calvin Lin. Back to the future: leveraging Belady's algorithm for improved cache replacement. In *International Symposium on Computer Architecture*, 2016.

[270] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News*, 2010.

[271] Majid Jalili, Ioannis Manousakis, Íñigo Goiri, Pulkit A. Misra, Ashish Raniwala, Husam Alissa, Bharath Ramakrishnan, Phillip Tuma, Christian Belady, Marcus Fontoura, and Ricardo Bianchini. Cost-Efficient Overclocking in Immersion-Cooled Datacenters. In *International Symposium on Computer Architecture*, 2021.

[272] Vijay Janapa Reddi, Benjamin C Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *ACM SIGARCH Computer Architecture News*, 2010.

[273] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. Predictive Parallelization: Taming Tail Latencies in Web Search. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2014.

[274] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX Conference on Networked Systems Design and Implementation*, 2014.

[275] Haipeng Jia, Yunquan Zhang, Guoping Long, Jianliang Xu, Shengen Yan, and Yan Li. GPURoofline: a model for guiding performance optimizations on GPUs. In *European Conference on Parallel Processing*, 2012.

[276] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing data analysis workloads in data centers. In *International Symposium on Workload Characterization*, 2013.

[277] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling Contention Management from Scheduling. In *Architectural Support for Programming Languages and Operating Systems*, 2010.

[278] Teresa Johnson, Mehdi Amini, and Xinliang David Li. ThinLTO: scalable and incremental LTO. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2017.

[279] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture*, 2017.

[280] Changhee Jung, Daeseob Lim, Jaejin Lee, and SangYong Han. Adaptive execution techniques for SMT multiprocessor architectures. In *ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2005.

[281] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX Symposium on Networked Systems Design and Implementation*, 2019.

[282] Alain Kägi, Doug Burger, and James R Goodman. Efficient synchronization: Let them eat qolb. In *International symposium on Computer architecture*, 1997.

[283] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density-connected subspace clustering for high-dimensional data. In *SDM*, 2004.

[284] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A Kim. Measuring interference between live datacenter applications. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[285] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *ISCA*, 2015.

[286] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *IEEE International Symposium on Workload Characterization*, 2014.

[287] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating Memory Allocation. In *Architectural Support for Programming Languages and Operating Systems*, 2017.

[288] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure devops. In *International Conference on Cloud Engineering*, 2016.

[289] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *ACM Symposium on Cloud Computing*, 2012.

[290] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant Memory Mappings for Fast Access to Large Memories. In *International Symposium on Computer Architecture*, 2015.

[291] Martin Karsten and Saman Barghi. User-level threading: Have your cake and eat it too. *ACM Measurement and Analysis of Computing Systems*, 2020.

[292] Harshad Kasture, Davide B Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *International Symposium on Microarchitecture*, 2015.

[293] Harshad Kasture and Daniel Sanchez. Tailbench: A benchmark suite and evaluation methodology for latency-critical applications. In *IISWC*, 2016.

[294] Stefanos Kaxiras and Margaret Martonosi. Computer Architecture Techniques for Power-Efficiency. *Synthesis Lectures on Computer Architecture*, 2008.

[295] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: Unified Instruction Supply for Scale-out Servers. In *International Symposium on Microarchitecture*, 2015.

[296] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: unified instruction supply for scale-out servers. In *International Symposium on Microarchitecture*, 2015.

[297] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *IEEE/ACM International Symposium on Microarchitecture*, 2021.

[298] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *IEEE/ACM International Symposium on Microarchitecture*, 2020.

[299] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. In *International Symposium on Computer Architecture*, 2021.

[300] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *International Symposium on Microarchitecture*, 2012.

[301] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. Bit-plane compression: Transforming data for better compression in many-core architectures. In *International Symposium on Computer Architecture*, 2016.

[302] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *ACM International Conference on Web Search and Data Mining*, 2015.

[303] Taewhan Kim and Jungeun Kim. Integration of code scheduling, memory allocation, and array binding for memory-access optimization. *Computer-Aided Design of Integrated Circuits and Systems*, 2006.

[304] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash local flash. *ACM SIGARCH Computer Architecture News*, 2017.

[305] Kathleen Knobe, Joan D. Lukas, and Guy L. Stelle, Jr. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 1990.

[306] Walden Ko, Mark Yankelevsky, Dimitrios S Nikolopoulos, and Constantine D Polychronopoulos. Effective cross-platform, multilevel parallelism via dynamic adaptive execution. In *Parallel and Distributed Processing Symposium*, 2001.

[307] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers Not to the Hippo. In *International Conference on Knowledge Discovery and Data Mining*, 2007.

[308] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level Persistency. In *International Symposium on Computer Architecture*, 2017.

[309] Aasheesh Kolli, Steven Pelley, Ali G. Saidi, Peter M. Chen, and Thomas F. Wenisch. High-Performance Transactions for Persistent Memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[310] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009.

[311] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server engineering insights for large-scale online services. *IEEE micro*, 2010.

[312] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting Through the Front-End Bottleneck with Shotgun. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[313] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting through the front-end bottleneck with shotgun. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[314] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM Journal on Computing*, 2000.

[315] Monica S Lam. *A systolic array optimizing compiler*. 2012.

[316] Koen Langendoen, John Romein, Raoul Bhoedjang, and Henri Bal. Integrating polling, interrupts, and thread management. In *Symposium on the Frontiers of Massively Parallel Computing*, 1996.

[317] P-A Larson, Jonathan Goldstein, and Jingren Zhou. MTCache: Transparent mid-tier database caching in SQL server. In *International Conference on Data Engineering*, 2004.

[318] Maysam Lavasani, Hari Angepat, and Derek Chiou. An FPGA-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 2013.

[319] Doug Lea and Wolfram Gloger. *A memory allocator*. Unix/mail, 1996.

[320] Timothy R Learmont. Fine-grained consistency mechanism for optimistic concurrency control using lock groups. *Google Patents*, 2001.

[321] Janghaeng Lee, Haicheng Wu, Madhumitha Ravichandran, and Nathan Clark. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *International Symposium on Computer Architecture*, 2010.

[322] Sang-Yun Lee and Junil Park. Architecture of 3D memory cell array on 3D IC. In *IEEE International Memory Workshop*, 2012.

[323] Andriy Lesyuk. *Mastering Redmine*. 2013.

[324] Chen Li, Edward Chang, Hector Garcia-Molina, and Gio Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 2002.

[325] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Workshop on Experimental computer science*, 2007.

[326] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[327] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *ACM Symposium on Cloud Computing*, 2014.

[328] Yun Liang and Tulika Mitra. Instruction cache locking using temporal reuse profile. In *Proceedings of the 47th Design Automation Conference*, 2010.

[329] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *USENIX Conference on Networked Systems Design and Implementation*, 2014.

[330] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *ACM SIGARCH Computer Architecture News*, 2008.

[331] Ankur Limaye and Tosiron Adegbija. A Workload Characterization of the SPEC CPU2017 Benchmark Suite. In *International Symposium on Performance Analysis of Systems and Software*, 2018.

[332] Yibei Ling, Tracy Mullen, and Xiaola Lin. Analysis of Optimal Thread Pool Size. *SIGOPS Operating Systems Review*, 2000.

[333] Lisa Hsu. The Importance of End-to-End Thinking in System Design. www.sigarch.org/the-importance-of-end-to-end-thinking-in-system-design, 2020. [Online; accessed 10-August-2021].

[334] Lisa Hsu. The Future of Datacenter Cooling. https://www.sigarch.org/the-future-of-datacenter-cooling/, 2021. [Online; accessed 10-August-2021].

[335] Dong Liu and Ralph Deters. The Reverse C10K Problem for Server-Side Mashups. In *International Conference on Service-Oriented Computing Workshops*, 2008.

[336] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Architectural Support for Programming Languages and Operating Systems*, 2017.

[337] Tiantian Liu, Minming Li, and Chun Jason Xue. Minimizing wcet for real-time embedded systems via static instruction cache locking. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[338] Ting Liu, Andrew W Moore, Ke Yang, and Alexander G Gray. An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*, 2004.

[339] Patrick Michael LiVecchi. Performance enhancements for threaded servers, 2004. US Patent 6,823,515.

[340] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *International Symposium on Computer Architecture*, 2014.

[341] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *International Symposium on Computer Architecture*, 2015.

[342] YJ Lo, S Williams, BV Straalen, TJ Ligocki, MJ Cordery, NJ Wright, MW Hall, and L Oliker. Roofline: an insightful visual performance model for multicore architectures. *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, 2015.

[343] Paul N Loewenstein, Mark A Luttrell, and Paul J Jordan. Load-monitor mwait. *Google Patents*, 2014.

[344] Unai Lopez-Novoa, Alexander Mendiburu, and Jose Miguel-Alonso. A survey of performance modeling and simulation techniques for accelerator-based computing. *Parallel and Distributed Systems*, 2014.

[345] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. Scale-out Processors. In *International Symposium on Computer Architecture*, 2012.

[346] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *International symposium on microarchitecture*, 2012.

[347] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J Newburn, and Joseph Devietti. LASER: Light, Accurate Sharing dEtection and Repair. In *International Symposium on High Performance Computer Architecture*, 2016.

[348] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: Efficient Indexing for High-dimensional Similarity Search. In *International Conference on Very Large Data Bases*, 2007.

[349] Peter Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *International Parallel Processing Symposium*, 1994.

[350] Hosein Mohammadi Makrani and Houman Homayoun. MeNa: A memory navigator for modern hardware in a scale-out environment. In *International Symposium on Workload Characterization*, 2017.

[351] Howard Mao, Randy H Katz, and Krste Asanović. Hardware acceleration for memory to memory copies. *EECS Department, University of California Berkeley*, 2017.

[352] Jason Mars. *Rethinking the architecture of warehouse-scale computers*. Ph.D. Dissertation, 2012.

[353] Jason Mars and Lingjia Tang. Whare-map: heterogeneity in homogeneous warehouse-scale computers. In *International Symposium on Computer Architecture*, 2013.

[354] Jason Mars, Lingjia Tang, and Robert Hundt. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters*, 2011.

[355] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *International Symposium on Microarchitecture*, 2011.

[356] Jose F Martinez and Josep Torrellas. Speculative locks: Concurrent execution of critical sections in shared-memory multiprocessors. In *High Performance Memory Systems*. 2004.

[357] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *ACM Symposium on Operating Systems Principles*, 2019.

[358] Andrew McCallum, Kamal Nigam, and Lyle H Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000.

[359] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* 2010.

[360] Cathy McCann, Raj Vaswani, and John Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 1993.

[361] James McNames. A fast nearest-neighbor algorithm based on a principal axis search tree. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2001.

[362] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power Management of Online Data-intensive Services. In *International Symposium on Computer Architecture*, 2011.

[363] David Meisner, Junjie Wu, and Thomas F. Wenisch. BigHouse: A Simulation Infrastructure for Data Center Systems. In *International Symposium on Performance Analysis of Systems & Software*, 2012.

[364] John M Mellor-Crummey and Michael L Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices*, 1991.

[365] Mitesh R Meswani, Laura Carrington, Didem Unat, Allan Snavely, Scott Baden, and Stephen Poole. Modeling and predicting performance of high performance computing applications on hardware accelerators. *High Performance Computing Applications*, 2013.

[366] Maged M. Michael. Scalable Lock-free Dynamic Memory Allocation. In *Programming Language Design and Implementation*, 2004.

[367] Amirhossein Mirhosseini, Hossein Golestani, and Thomas F. Wenisch. Hyperplane: A notification accelerator for software data planes. In *International Symposium on Microarchitecture*, 2020.

[368] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. Enhancing Server Efficiency in the Face of Killer Microseconds. In *International Symposium on High Performance Computer Architecture*, 2019.

[369] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. Hiding the Microsecond-Scale Latency of Storage-Class Memories with Duplexity. In *Annual Non-Volative Memories Workshop*, 2019.

[370] Seyedamirhossein Mirhosseininiri. *Datacenter Architectures for the Microservices Era*. PhD thesis, University of Michigan, 2021.

[371] Nikita Mishra, John D Lafferty, and Henry Hoffmann. Esp: A machine learning approach to predicting application interference. In *International Conference on Autonomic Computing*, 2017.

[372] Sparsh Mittal. A survey of techniques for cache locking. *Transactions on Design Automation of Electronic Systems*, 2016.

[373] Gordon E Moore. Cramming more components onto integrated circuits. *McGraw-Hill New York*, 1965.

[374] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed event-based systems*. 2006.

[375] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2009.

[376] Marius Muja and David G. Lowe. Scalable Nearest Neighbor Algorithms for High Dimensional Data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[377] Shubhendu S Mukherjee, Babak Falsafi, Mark D Hill, and David A Wood. Coherent network interfaces for fine-grain communication. *ACM SIGARCH Computer Architecture News*, 1996.

[378] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. 2016.

[379] Vijay Nagarajan and Rajiv Gupta. Ecmon: exposing cache events for monitoring. *ACM SIGARCH Computer Architecture News*, 2009.

[380] Roger M. Needham. Denial of Service. In *ACM Conference on Computer and Communications Security*, 1993.

[381] Gilbert Neiger and Rajesh M Sankaran. Delivering interrupts to user-level applications. *Google Patents*, 2018.

[382] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding PCIe performance for end host networking. In *ACM Special Interest Group on Data Communication*, 2018.

[383] Jarek Nieplocha and Jialin Ju. *ARMCI: A portable aggregate remote memory copy interface*. Citeseer, 2000.

[384] Siddharth Nilakantan, Steven Battle, and Mark Hempstead. Metrics for early-stage modeling of many-accelerator architectures. *IEEE Computer Architecture Letters*, 2012.

[385] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, and Paul Saab. Scaling Memcache at Facebook. In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[386] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out numa. *ACM SIGPLAN Notices*, 2014.

[387] Cedric Nugteren and Henk Corporaal. The boat hull model: enabling performance prediction for parallel computing prior to code development. In *Conference on Computing Frontiers*, 2012.

[388] Guilherme Ottoni. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Conference on Programming Language Design and Implementation*, 2018.

[389] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX Symposium on Networked Systems Design and Implementation*, 2019.

[390] Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, 1999.

[391] Sankaralingam Panneerselvam and Michael Swift. Rinnegan: Efficient Resource Use in Heterogeneous Architectures. In *International Conference on Parallel Architectures and Compilation*, 2016.

[392] Gabriele Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf`.

[393] Ioannis Papadakis, Konstantinos Nikas, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. Improving QoS and Utilisation in modern multi-core servers with Dynamic Cache Partitioning. In *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017*, 2017.

[394] David Pariag, Tim Brecht, Ashif S. Harji, Peter A. Buhr, Amol Shukla, and David R. Cheriton. Comparing the performance of web server architectures. In *European Conference on Computer Systems*, 2007.

[395] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *IEEE/ACM International Conference on Computer-aided design*, 2006.

[396] Lance Parsons, Ehtesham Haque, and Huan Liu. Subspace clustering for high dimensional data: a review. *ACM SIGKDD Explorations Newsletter*, 2004.

[397] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 2015.

[398] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro*, 40(2):53–62, 2020.

[399] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. In *International Symposium on Computer Architecture*, 2014.

[400] Lillian Pentecost, Marco Donato, Brandon Reagen, Udit Gupta, Siming Ma, Gu-Yeon Wei, and David Brooks. MaxNVM: Maximizing DNN storage density and inference efficiency with sparse encoding and error mitigation. In *International Symposium on Microarchitecture*, 2019.

[401] Lillian Pentecost, Marco Donato, Akshitha Sriraman, Gu-Yeon Wei, and David Brooks. Analytically Modeling NVM Design Trade-Offs. In *Non-Volatile Memories Workshop (Poster)*.

[402] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems*, 2016.

[403] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H Loh. Increasing TLB reach by exploiting clustering in page translations. In *International Symposium on High Performance Computer Architecture*, 2014.

[404] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. Colt: Coalesced large-reach TLBs. In *International Symposium on Microarchitecture*, 2012.

[405] Martin F Porter. Snowball: A language for stemming algorithms. `http://snowball.tartarus.org/texts/introduction.html`.

[406] Martin F Porter. An algorithm for suffix stripping. *Program*, 1980.

[407] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus prime: Accelerating data transformation in servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[408] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Symposium on Operating Systems Principles*, 2017.

[409] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *ACM Symposium on Cloud Computing*, 2015.

[410] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 1990.

[411] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Towards scalable deep learning via i/o analysis and optimization. In *High Performance Computing and Communications*, 2017.

[412] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *International Symposium on Workload Characterization*, 2011.

[413] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *International Symposium on Computer Architecuture*, 2014.

[414] Emilee Rader and Rebecca Gray. Understanding user beliefs about algorithmic curation in the facebook news feed. In *ACM conference on human factors in computing systems*, 2015.

[415] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *ACM Conference on SIGCOMM*, 2014.

[416] David R Raymond and Scott F Midkiff. Denial-of-service in wireless sensor networks: Attacks and defenses. *IEEE Pervasive Computing*, 2008.

[417] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *International Symposium on Microarchitecture*, 1999.

[418] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, 2010.

[419] Max Roser, Hannah Ritchie, and Esteban Ortiz-Ospina. Internet. `https://ourworldindata.org/internet`.

[420] Efraim Rotem. Intel architecture, code name Skylake deep dive: A new architecture to manage power performance and energy efficiency. In *Intel Developer Forum*, 2015.

[421] J Rupley. Samsung Exynos M3 Processor. *IEEE Hot Chips*, 30, 2018.

[422] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, Haruhiko Kojima, et al. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *VLDB*, 2000.

[423] Yoshiei Sato, Ryuichi Nagaoka, Akihiro Musa, Ryusuke Egawa, Hiroyuki Takizawa, Koki Okabe, and Hiroaki Kobayashi. Performance tuning and analysis of future vector processors based on the roofline model. In *Workshop on MEmory performance: DEaling with Applications, systems and architecture*, 2009.

[424] Doug Schmidt and Paul Stephenson. Experience using design patterns to evolve communication software across diverse OS platforms. In *European Conference on Object-Oriented Programming*, 1995.

[425] Douglas C Schmidt and Chris Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine*, 1999.

[426] David Vincent Schuehler. *Techniques for processing TCP/IP flow content in network switches at gigabit line rates*. Semantic Scholar.

[427] Bart Selman and Carla P Gomes. Hill-climbing search. *Encyclopedia of Cognitive Science*, 2006.

[428] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *International Symposium on Microarchitecture*, 2013.

[429] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX Annual Technical Conference*, 2020.

[430] Gregory Shakhnarovich, Paul Viola, and Trevor Darrell. Fast pose estimation with parameter-sensitive hashing. In *IEEE International Conference on Computer Vision*, 2003.

[431] Ratnesh K Sharma, Cullen E Bash, Chandrakant D Patel, Richard J Friedrich, and Jeffrey S Chase. Balance of power: Dynamic thermal management for internet data centers. *IEEE Internet Computing*, 2005.

[432] Sourabh Sharma. *Mastering Microservices with Java 9: Build domain-driven microservice-based applications with Spring, Spring Cloud, and Angular*. Packt Publishing Ltd, 2017.

[433] Konstantin Shemyak and Kai Vehmanen. Scalability of tcp servers, handling persistent connections. In *International Conference on Networking*, 2007.

[434] Robert T Short, John M Parchem, and David N Cutler. Method and apparatus for reducing the rate of interrupts by generating a single interrupt for a group of events. *Google Patents*, 1998.

[435] Vaclav Simek and Ram Rakesh Asn. Gpu acceleration of 2d-dwt image compression in matlab with cuda. In *UKSIM European Symposium on Computer Modeling and Simulation*, 2008.

[436] Evangelia Sitaridi, Orestis Polychroniou, and Kenneth A Ross. SIMD-accelerated regular expression matching. In *International Workshop on Data Management on New Hardware*, 2016.

[437] Magnus Själander, Margaret Martonosi, and Stefanos Kaxiras. Power-Efficient Computer Architectures: Recent Advances. *Synthesis Lectures on Computer Architecture*, 2014.

[438] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal Processing Magazine*, 2008.

[439] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *ACM SIGCOMM Computer Comm. Review*, 2005.

[440] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In *International Symposium on Parallel and Distributed Processing*, 2013.

[441] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis lectures on computer architecture*, 2011.

[442] Stephen M Specht and Ruby B Lee. Distributed Denial of Service: Taxonomies of Attacks, Tools, and Countermeasures. In *ISCA International Conference on Parallel and Distributed Computing (and Communications) Systems*, 2004.

[443] Akshitha Sriraman. Unfair Data Centers for Fun and Profit. In *Wild and Crazy Ideas (ASPLOS)*, 2019.

[444] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[445] Akshitha Sriraman and Abhishek Dhanotia. Understanding Acceleration Opportunities at Hyperscale. *IEEE Micro*, 2021.

[446] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *The International Symposium on Computer Architecture*, 2019.

[447] Akshitha Sriraman, Sihang Liu, Sinan Gunbay, Shan Su, and Thomas F. Wenisch. Deconstructing the Tail at Scale Effect Across Network Protocols. *The Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2016.

[448] Akshitha Sriraman and Thomas F. Wenisch. μSuite: A Benchmark Suite for Microservices. In *IEEE International Symposium on Workload Characterization*, 2018.

[449] Akshitha Sriraman and Thomas F Wenisch. μTune: Auto-Tuned Threading for OLDI Microservices. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, 2018.

[450] Akshitha Sriraman and Thomas F. Wenisch. Performance-Efficient Notification Paradigms for Disaggregated OLDI Microservices. In *Workshop on Resource Disaggregation*, 2019.

[451] Alexei Starovoitov. BPF in LLVM and kernel. In *Linux Plumbers Conference*, 2015.

[452] W Richard Stevens and Gary R Wright. *TCP/IP illustrated (vol. 2) the implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[453] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. CAPI: A coherent accelerator processor interface. *IBM Journal R&D*, 2015.

[454] Daniel Stutzbach and Reza Rejaie. Improving lookup performance over a widely-deployed DHT. In *International Conference on Computer Communications*, 2006.

[455] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD “Zen 2” Processor. *IEEE Micro*, 40(2):45–52, 2020.

[456] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 2004.

[457] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[458] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[459] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications. In *Int. Symposium on Computer Architecture*, 2011.

[460] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In *ACM SIGMOD International Conference on Management of data*, 2009.

[461] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Transactions on Database Systems*, 2010.

[462] Michael B Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference*, 2012.

[463] Gil Tene. How not to measure latency. In *Low Latency Summit*, 2013.

[464] Kengo Terasawa and Yuzuru Tanaka. Spherical LSH for approximate nearest neighbor search on unit hypersphere. In *Workshop on Algorithms and Data Structures*, 2007.

[465] Xinmin Tian, Hideki Saito, Serguei V Preis, Eric N Garcia, Sergey S Kozhukhov, Matt Masten, Aleksei G Cherkasov, and Nikolay Panchenko. Practical simd vectorization techniques for intel® xeon phi coprocessors. In *Parallel & Distributed Processing*, 2013.

[466] Stefan Tilkov and Steve Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 2010.

[467] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *European Conference on Computer Systems*, 2020.

[468] Pedro Trancoso, J-L Larriba-Pey, Zheng Zhang, and Josep Torrellas. The memory performance of DSS commercial workloads in shared-memory multiprocessors. In *International Symposium High-Performance Computer Architecture*, 1997.

[469] Caroline June Trippel. *Concurrency and Security Verification in Heterogeneous Parallel Systems*. PhD thesis, Princeton University, 2019.

[470] Dan Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Workshop on Experimental computer science*, 2007.

[471] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2012.

[472] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. Timetrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *International Symposium on Microarchitecture*, 2015.

[473] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *International Symposium on High Performance Computer Architecture*, 2007.

[474] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, and Jeremy Hoon. Tao: how facebook serves the social graph. In *International Conference on Management of Data*, 2012.

[475] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems*, 2015.

[476] Jerome Vienne, Jitong Chen, Md Wasi-Ur-Rahman, Nusrat S Islam, Hari Subramoni, and Dhabaleswar K Panda. Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems. In *IEEE 20th Annual Symposium on High-Performance Interconnects*, 2012.

[477] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference*, 2015.

[478] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot. An effective dram cache architecture for scale-out servers. Technical Report MSR-TR-2016-20, 2016.

[479] J Robert Von Behren, Jeremy Condit, and Eric A Brewer. Why Events Are a Bad Idea (for High-Concurrency Servers). In *Hot Topics in Operating Systems*, 2003.

[480] M Mitchell Waldrop. The chips are down for moore's law. *Nature News*, 530(7589):144, 2016.

[481] Jin-Yi Wang, Yen-Shiang Shue, TN Vijaykumar, and Saurabh Bagchi. Pesticide: Using smt to improve performance of pointer-bug detection. In *International Conference on Computer Design*, 2006.

[482] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, and Shujie Zhang. Bigdatabench: A big data benchmark suite from internet services. In *HPCA*, 2014.

[483] Qingyang Wang, Chien-An Lai, Yasuhiko Kanemasa, Shungeng Zhang, and Calton Pu. A Study of Long-Tail Latency in n-Tier Systems: RPC vs. Asynchronous Invocations. In *International Conference on Distributed Computing Systems*, 2017.

[484] Yaohua Wang, Arash Tavakkol, Lois Orosa, Saugata Ghose, Nika Ghiasi, Minesh Patel, Jeremie S Kim, Hasan Hassan, Mohammad Sadrosadati, and Onur Mutlu. Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration. In *International Symposium on Microarchitecture*, 2018.

[485] Zheng Wang and Michael F.P. O'Boyle. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[486] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *ACM Symposium on Operating Systems Principles*, 2001.

[487] Thomas F Wenisch, Michael Ferdman, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal streams in commercial server applications. In *IEEE International Symposium on Workload Characterization*, 2008.

[488] Thomas F Wenisch, Stephen Somogyi, Nikolaos Hardavellas, Jangwoo Kim, Anastassia Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *International Symposium on Computer Architecture*, 2005.

[489] Wikipedia contributors. Alder lake (microprocessor) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Alder_Lake_(microprocessor)&oldid=990207738`, 2020. [Online; accessed 25-November-2020].

[490] Robert A Williams and Jerry C Kuo. Mechanism for minimizing overhead usage of a host system by polling for subsequent interrupts after service of a prior interrupt. *Google Patents*, 2000.

[491] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, and Sy et al. Choudhury. Machine learning at facebook: Understanding inference at the edge. In *High Performance Computer Architecture*, 2019.

[492] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, 2013.

[493] Owen Yamauchi. *Hack and HHVM: programming productivity without breaking things*. " O'Reilly Media, Inc.", 2015.

[494] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *USENIX Annual Technical Conference*, 2016.

[495] Ahmad Yasin, Yosi Ben-Asher, and Avi Mendelson. Deep-dive analysis of the data analytics workload in cloudsuite. In *International Symposium on Workload Characterization*, 2014.

[496] Joseph Yiu. *The definitive guide to the ARM Cortex-M3*. Newnes, 2009.

[497] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[498] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2019.

[499] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi 2: CPU performance isolation for shared compute clusters. In *European Conference on Computer Systems*, 2013.

[500] Yao Zhang and John D Owens. A quantitative perf. analysis model for GPU architectures. In *High Perf. Computer Architecture*, 2011.

[501] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the Source of Tail Latency Through Precise Load Testing and Statistical Inference. In *International Symposium on Computer Architecture*, 2016.

[502] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. Dynamic Memory Optimization Using Pool Allocation and Prefetching. *SIGARCH Comput. Archit. News*, 2005.

[503] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iwatcher: Efficient architectural support for software debugging. In *Annual International Symposium on Computer Architecture*, 2004.

[504] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News*, 2016.

[505] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. Microarchitectural Implications of Event-driven Server-side Web Applications. In *International Symposium on Microarchitecture*, 2015.

[506] Mark Zuckerberg, Ruchi Sanghvi, Andrew Bosworth, Chris Cox, Aaron Sittig, Chris Hughes, Katie Geminder, and Dan Corson. Dynamically providing a news feed about a user of a social network. `https://patents.google.com/patent/US7669123B2/en`.

[507] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *ICDE*, 2006.