

**DEMYSTIFYING TAIL LATENCY THROUGH BENCHMARK AND  
HARDWARE-SOFTWARE CODESIGN**

by

Heng Zhuo

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

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2023

Date of final oral examination: 06/13/23

The dissertation is approved by the following members of the Final Oral Committee:

Mikko Lipasti, Professor, Electrical and Computer Engineering, UW-Madison

Joshua San Miguel, Assistant Professor, Electrical and Computer Engineering, UW-Madison

Umit Ogras, Associate Professor, Electrical and Computer Engineering, UW-Madison

Michael Swift, Professor, Computer Science, UW-Madison

Matt Sinclair, Assistant Professor, Computer Science, UW-Madison



*To a better world.*

## ACKNOWLEDGMENTS

I still cannot believe what I have achieved so far. Some say they knew they always wanted a Ph.D. growing up, but that was not me. I came to the University of Wisconsin-Madison as a master's student. During that period, the more I learned, the more I wanted to stay and extend my study to a Ph.D. I did, and it took me seven years to finish in total. Looking back, there are many people I like to thank.

Firstly, I want to thank my advisor, Professor Mikko Lipasti. I only knew what a Ph.D. life was after I joined Predictive High-Performance Architecture Research Mavens (PHARM), the research group led by Mikko. Mikko first taught me how to read papers, which was rough initially. I can barely remember what the paper said two paragraphs before, nor understand what they did and why they did it. Of course, I am much better now. After deciding to stay and continue my Ph.D. study, I started exploring the unknown world. Mikko gives us the most freedom regarding what we want to research specifically as long as it fits the funding topics broadly. Mikko never pushes us on what he thinks is the best. Instead, he taught us how to evaluate current issues and situations, analyze potential solutions and tradeoffs, and finally decide ourselves. Like any other group member, I was like a trailblazer. I was interested in the specific path Mikko may have yet to consider. But knowing Mikko always has my back and support pushes me forward fearlessly. During our meetings, Mikko always says: "You know more than me in this.". It is true in how much detail we know about our research, but Mikko is the lighthouse, directing us through the fog and mist. That is precisely how I became confident in my research skills. There is a famous Chinese proverb, **"Even if someone is your teacher for only a day, you should regard him like your father for the rest of your life."** I am grateful to have had Mikko being my advisor for the last seven years, and will never forget what Mikko taught me.

Then, I want to thank my lab mates: Kyle, Ravi, Yanqi, Chien-fu, Bujji, Gokul, David, Soroosh, Carly, Rohit, Dibakar, and Michael. Yanqi is the one I met initially when I joined the group, and we were both master's degree-seeking students. Yanqi led me in programming and hacking simulators in the early days before he moved to Cornell to pursue his Ph.D.. Dibakar, Rohit, and Michael were senior Ph.D. students when I joined. They showed me discipline and hard work. Gokul and David are two great lab mates who were around two years into the program when I joined, with a little more experience but not distant. They give me countless guidance on becoming a better Ph.D. student in both my academic career and outside of the lab in life. Thanks to David for our collaboration in Tightly Coupled Accelerator (TCA) research that my thesis work is heavily based. Also, thanks to Kyle for our collaboration in Bitstream Computing workload and tools development. I want to thank Kyle and Ravi, especially, for being two great friends I

met. We started around the same time and were here the whole period together. It was fantastic to have each other when we achieved all the milestones: qualifier, prelim, and defense. We also shared a lot outside of work. We had great food together, had great drinks together, and been to incredible places together. Kyle and Ravi even traveled 16 hours of flight, all the way to China, to attend my wedding in Xiamen. To that, cheers.

Moreover, I must thank my parents, Chaofu Zhuo and Meiji Wu. I can only be here with their support. I came from Xiamen (also known as Amoy, from the Hokkien dialect), a beautiful island city in Fujian province on the southeast coast of China. I spend most of my education time in Xiamen before college. However, there was a long way before it. My family was from a much smaller place called Mingxi County, within Sanming City. It is around 400 km (250 miles) inland from Xiamen, within the same province. There was little to gain when my parents grew up during the 70s and 80s in China. Due to poverty, they did not even have a chance to finish high school. They moved to Guizhou, a landlocked province over 2000 km (1250 miles) from home, to seek better opportunities. They successfully started their business during the period of Chinese economic reform but wanted a better education than Guizhou could offer. They value education very much since they were not able to. So, at the age of 6, my parents decided to move to Xiamen, closer to their hometown but a wealthy and much more developed city. There is so much beyond this paragraph that can describe what they have done to provide everything for me and my younger sister, Xin Zhuo. Speaking of Xin, I used to fight a lot with Xin when we grew up, being perfectly normal siblings. However, we became much closer when both went to college and had a distance of a few thousand miles. We cheered each other no matter up and down and talked about our issues. I am fortunate to have the perfect sibling in my life.

Finally, I want to thank my beautiful wife, Jieyu Lin. We went to the same middle and high school but only started dating after high school, right before I left for London for my undergraduate study while she went to college in China. After that, I was in Wisconsin for my Ph.D. study for another seven years. We did have around two years of overlap in the US when she was pursuing her master's degree at Georgetown University in D.C. Long distance is a huge challenge, but here we are with smiles. She never doubts me about pursuing what I dream of, even if it means I am away in another country. She always cares about my work-life balance. She is always curious about what I am working on and patiently listen to my vent about roadblocks I have during research. She is always encouraging when I was in my low days. I am eternally grateful for having Jieyu being my wife supporting me over the past few years and for the upcoming journey together.

# CONTENTS

|                                                      |      |
|------------------------------------------------------|------|
| Abstract                                             | viii |
| 1 Introduction                                       | 1    |
| 1.1 Tail Latency                                     | 1    |
| 1.2 Tightly Coupled Accelerators                     | 5    |
| 1.3 Summary of Contributions and Thesis Organization | 7    |
| 2 Background and Related Work                        | 9    |
| 2.1 Server Performance Analysis and Benchmarks       | 9    |
| 2.2 Network and Interrupt Handling                   | 13   |
| 2.2.1 Runtime Configuration                          | 14   |
| 2.2.2 Kernel Bypassing                               | 15   |
| 2.2.3 Network Offloading                             | 17   |
| 2.2.4 Fast Interrupt Handling                        | 18   |
| 2.2.5 Power Management                               | 19   |
| 2.3 Synchronization Mechanism                        | 19   |
| 2.4 Hardware Accelerators                            | 21   |
| 2.5 Summary                                          | 23   |
| 3 Workload Analysis and Generation                   | 24   |
| 3.1 Workload Analysis                                | 28   |
| 3.1.1 Service Time Distribution                      | 29   |
| 3.1.2 Thread Scaling                                 | 31   |
| 3.1.3 NIC Interrupts                                 | 32   |
| 3.2 Workload Generation and Validation               | 33   |
| 3.3 Case Study                                       | 38   |
| 3.3.1 DNN evolution                                  | 38   |
| 3.3.2 NIC hardIRQ Accelerator                        | 39   |
| 3.3.3 Garbage Collection                             | 40   |
| 3.4 Summary                                          | 41   |
| 4 NAPI and threaded NAPI                             | 42   |
| 4.1 Terminology                                      | 43   |
| 4.1.1 CPU contexts                                   | 43   |
| 4.1.2 Other hardware interrupts                      | 44   |
| 4.2 NAPI                                             | 45   |
| 4.2.1 Network Interface Controller                   | 46   |
| 4.2.2 Hardware interrupt                             | 47   |
| 4.2.3 Polling in softirq                             | 47   |
| 4.2.4 Polling in ksoftirqd                           | 49   |

|       |                                                 |     |
|-------|-------------------------------------------------|-----|
| 4.3   | threaded NAPI . . . . .                         | 50  |
| 4.4   | Performance Analysis . . . . .                  | 52  |
| 4.5   | Summary . . . . .                               | 56  |
| 5     | Cascading-effect . . . . .                      | 57  |
| 5.1   | IRQ meets Synchronization . . . . .             | 57  |
| 5.1.1 | Base server activities . . . . .                | 58  |
| 5.1.2 | Threads race conditions . . . . .               | 59  |
| 5.1.3 | Two solutions: sleeping VS spinning . . . . .   | 60  |
| 5.1.4 | Hybrid lock as a solution . . . . .             | 62  |
| 5.1.5 | Timing Disturbance . . . . .                    | 63  |
| 5.2   | Quantitative Analysis . . . . .                 | 65  |
| 5.3   | Possible improvement . . . . .                  | 70  |
| 5.4   | Summary . . . . .                               | 72  |
| 6     | Interrupt Handling Accelerator . . . . .        | 73  |
| 6.1   | Opportunity for Hardware Acceleration . . . . . | 75  |
| 6.1.1 | Generic Interrupt . . . . .                     | 75  |
| 6.1.2 | NIC Devices Interrupt . . . . .                 | 79  |
| 6.1.3 | NAPI: switching from interrupt mode . . . . .   | 81  |
| 6.1.4 | NAPI: schedule polling thread . . . . .         | 83  |
| 6.2   | Accelerator design . . . . .                    | 85  |
| 6.3   | Accelerator Configurability . . . . .           | 87  |
| 6.3.1 | Obtaining Addresses . . . . .                   | 89  |
| 6.4   | Summary . . . . .                               | 90  |
| 7     | Adaptive Hybrid Lock Accelerator . . . . .      | 91  |
| 7.1   | Existing Synchronization Design . . . . .       | 93  |
| 7.1.1 | Glibc . . . . .                                 | 93  |
| 7.1.2 | futex . . . . .                                 | 95  |
| 7.2   | Adaptive Hybrid Lock Mechanism . . . . .        | 96  |
| 7.3   | LockAcc Design . . . . .                        | 98  |
| 7.4   | Summary . . . . .                               | 101 |
| 8     | Evaluation . . . . .                            | 102 |
| 8.1   | Methodology . . . . .                           | 102 |
| 8.2   | Performance Evaluation . . . . .                | 104 |
| 8.2.1 | Multithreaded Performance . . . . .             | 106 |
| 8.3   | Area and Energy Evaluation . . . . .            | 108 |
| 8.4   | Summary . . . . .                               | 110 |
| 9     | Conclusions and Future Directions . . . . .     | 111 |
| 9.1   | Summary of contributions . . . . .              | 111 |
| 9.2   | Limitations and Future Directions . . . . .     | 112 |
| 9.2.1 | TailWAG . . . . .                               | 112 |
| 9.2.2 | Tightly Coupled Accelerators . . . . .          | 114 |
| 10    | Appendix . . . . .                              | 116 |

|        |                                     |     |
|--------|-------------------------------------|-----|
| 10.1   | Kernel Source code . . . . .        | 116 |
| 10.1.1 | Interrupt Handling Layer . . . . .  | 116 |
| 10.1.2 | NIC Driver . . . . .                | 123 |
| 10.1.3 | NAPI . . . . .                      | 124 |
| 10.1.4 | Generic Scheduler . . . . .         | 126 |
| 10.1.5 | Round Robin Scheduler . . . . .     | 133 |
| 10.2   | How to calculate the time . . . . . | 135 |
| 10.2.1 | Resolution . . . . .                | 136 |

## LIST OF FIGURES

|            |                                                                                                                                                                                                                                                                                                                                                                                                         |    |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Figure 1.1 | Blue color region shows the cumulative distribution of a service time. The worse 1% of service time can be 5.4 slower than average latency. . . . .                                                                                                                                                                                                                                                     | 2  |
| Figure 1.2 | An application can break down into different tiers. Especially now, recent work and trends on microservices, and function as services on the cloud. . . . .                                                                                                                                                                                                                                             | 3  |
| Figure 1.3 | Latency vs Throughput. SEP: reserving core 1 for interrupt handling improves latency at low to moderate load, but lowers throughput; MIX: using all 4 cores for both interrupts and server threads yields higher throughput, with worse latency. . . . .                                                                                                                                                | 4  |
| Figure 2.1 | Latency: in server system, mean latency is important, same as tail latency, such as 95 percentile or 99 percentile. The application shown is silo from Tailbench, single threaded. . . . .                                                                                                                                                                                                              | 10 |
| Figure 2.2 | The service time itself usually does not change much, given enough computing resources. The application shown is silo from Tailbench, single threaded. . . . .                                                                                                                                                                                                                                          | 11 |
| Figure 3.1 | Linux real-time scheduling limit: The Linux kernel default limit for real-time tasks is 95% of total CPU time. When real-time tasks reach the limit, they are preempted for 50 ms, leading to the latency spikes shown above (left). Over-riding this setting to unlimited CPU utilization eliminates most of the latency spikes (right) but may cause starvation. Hardware setup in Table 3.1. . . . . | 25 |
| Figure 3.2 | TailWAG workflow. . . . .                                                                                                                                                                                                                                                                                                                                                                               | 27 |
| Figure 3.3 | Service Time Distribution: Blue line shows the measured service distribution; Red line shows distribution coverage used in TailWAG workload generation. Python Scipy is used for distribution fitting. . . . .                                                                                                                                                                                          | 29 |
| Figure 3.4 | Thread Scaling (95th tail) for 1/2/4 server threads. Dashed lines show TailWAG, and solid lines are measured. TailWAG matches behavior well in all cases except mores, where it reflects an upper bound without its memory bottleneck. TCP contention impacts scaling for silo. . . . .                                                                                                                 | 31 |
| Figure 3.5 | NIC disturbance (95th tail): Effect of IRQ Handling Noise on Tail Latency using TailWAG. TailWAG can capture effects from different applications, and also predict a memory bottleneck for mores. . . .                                                                                                                                                                                                 | 34 |
| Figure 3.6 | DNN evolution: Increased model complexity (orange), vectorization (green), hardware accelerator (red) affect mean latency. . . . .                                                                                                                                                                                                                                                                      | 39 |

|            |                                                                                                                                                                                                                                                                                                                                 |    |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Figure 3.7 | NIC IRQ: SEP/MIX with hardIRQ Accelerator affect 95th tail latency, with same application (silo). The optimal case needs an additional (5th) core. . . . .                                                                                                                                                                      | 40 |
| Figure 3.8 | Java GC: Parallel (plgc, throughput oriented) and Zero (zgc, latency oriented) affect mean and 99th tail latency. . . . .                                                                                                                                                                                                       | 41 |
| Figure 4.1 | Lower level conext (closer to hardware) has higher priority. Userspace and usercontext can preempt each other. . . . .                                                                                                                                                                                                          | 44 |
| Figure 4.2 | NIC interrupt interface NAPI is mixture of interrupt and polling. . .                                                                                                                                                                                                                                                           | 45 |
| Figure 4.3 | NAPI uses softirq for polling . . . . .                                                                                                                                                                                                                                                                                         | 50 |
| Figure 4.4 | threaded NAPI uses dedicated kernel thread for polling . . . . .                                                                                                                                                                                                                                                                | 51 |
| Figure 4.5 | Using threaded NAPI helps performance, and improves throughput and latency at higher loads. Even more, it can achieve better performance than SEP, which uses an additional core, shown on the left. On the right, it shows the interrupts number saturated and decreases when polling cannot handle packets fast enough. . .   | 54 |
| Figure 4.6 | Scheduling policy matters: NAPI thread has higher priority and is equally bad polling as softirq since it preempts server threads. Using Round-Robin and enforcing the same or lower priority is preferred. On the right, it shows the interrupts number does not saturate for high-priority threaded NAPI. . . . .             | 55 |
| Figure 4.7 | 4 threads workload running. tNAPI performs better than NAPI without surprise. SEP performs better than TNAPI in multiple threads cases. On the right, it shows the interrupts number does not saturate for NAPI, since multiple threads on multiple cores push server application to be run cores 1-3, leave core 0 for NAPI. . | 55 |
| Figure 4.8 | Scheduling policy matters: NAPI thread with higher priority is equally bad as using CFS. Using Round-Robin same or lower priority is preferred. Interrupt numbers on the right shows it saturates sooner when NAPI thread has lower priority. . . . .                                                                           | 56 |
| Figure 5.1 | Ideal case with single thread. . . . .                                                                                                                                                                                                                                                                                          | 58 |
| Figure 5.2 | Race condition in multiple threads application . . . . .                                                                                                                                                                                                                                                                        | 59 |
| Figure 5.3 | Sleeping Locks: additional cost on waking up . . . . .                                                                                                                                                                                                                                                                          | 61 |
| Figure 5.4 | Spinning locks: CPU cannot serve other task . . . . .                                                                                                                                                                                                                                                                           | 61 |
| Figure 5.5 | Hybrid Lock: spin count is important . . . . .                                                                                                                                                                                                                                                                                  | 63 |
| Figure 5.6 | Asynchronous Timing Disturbance can cause trouble: may need longer spin count.; . . . . .                                                                                                                                                                                                                                       | 64 |
| Figure 5.7 | CPU can be “stolen” by other task. . . . .                                                                                                                                                                                                                                                                                      | 64 |
| Figure 5.8 | Even with threaded NAPI, asynchronous interrupt exits and thread2 is delayed. . . . .                                                                                                                                                                                                                                           | 65 |
| Figure 5.9 | Spinning lock improves the threads scaling, saturation points are closer with spinning lock. . . . .                                                                                                                                                                                                                            | 66 |

|             |                                                                                                                                                                                                                                                                                                 |    |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Figure 5.10 | Spinning lock improvement, the server can spin forever if no other task can use the CPU. A longer spin limit increases the lock acquire rate by spinning (on the right), leading to better latency (on the left). Different color lines represent different QPS per thread. . . . .             | 66 |
| Figure 5.11 | Server is more sensitive to spin limit when there is another task that shares the CPU. Here is NIC IRQ. Different color lines represent different QPS per thread. Similarly, the spin limit helps increase the lock acquire rate, but there is an upper bound before it spins too long. . . . . | 67 |
| Figure 5.12 | Threaded NAPI: reduce the asynchronous region allows a wide range of spin limits for acceptable performance. Smaller timing disturbance allows higher lock acquire to start. . . . .                                                                                                            | 67 |
| Figure 5.13 | Left: Mean spin usage is not increased much after, but it helps for catching longer critical instances. Right: when the spin limit is higher, the one that fails to acquire the lock are longer latency ones, until all are acquired by spinning. . . . .                                       | 68 |
| Figure 5.14 | 99th tail on the time obtaining the lock: having others can use CPU leads to disaster. . . . .                                                                                                                                                                                                  | 68 |
| Figure 5.15 | 8 threads Threaded NAPI: the spin limit is similar comparing to 4 threads example. . . . .                                                                                                                                                                                                      | 69 |
| Figure 5.16 | Cascading effects: increasing the spin limit for application reduces the penalties for shorter tails (blue line on the right). However, if the spin limit is too high, it increased even longer tail, which is unwanted. . . . .                                                                | 69 |
| Figure 5.17 | Break down of CPU time spend in different region. . . . .                                                                                                                                                                                                                                       | 70 |
| Figure 5.18 | Higher IRQPS leads to higher percentage of request affected by IRQ. . . . .                                                                                                                                                                                                                     | 71 |
| Figure 6.1  | Chip Level overview: interrupt signal sent to CPU through irq chip and bus. The proposed accelerator is resident within the CPU, it does not alter the interface between devices. . . . .                                                                                                       | 74 |
| Figure 6.2  | Interrupt table: it begins with predefined interrupts, end with user defined interrupts. When an interrupt happens, the CPU uses address stored in the interrupt table to local entry point, defined in the linux kernel . . . . .                                                              | 77 |
| Figure 6.3  | Accelerator overview: within the CPU, the IRQAcc behaves as Finite State Machine. . . . .                                                                                                                                                                                                       | 85 |
| Figure 6.4  | Accelerator FSM: whenever IRQ arrives, TCA checks condition, take actions if meet and processed; else, return pass IRQ to default path or early return success. . . . .                                                                                                                         | 86 |
| Figure 7.1  | 2d heatmap of 99 percentile latency for two different loads. The range of ideal spin limit is different on two different locks, for two different loads . . . . .                                                                                                                               | 92 |

|            |                                                                                                                                                                                                       |     |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| Figure 7.2 | QPS=18000: The spin limit quickly move to the desired region. When the system under higher load, the recv spin limit is higher than the transmitting spin limit. . . . .                              | 97  |
| Figure 7.3 | QPS=16000: The spin limit quickly move to the desired region. For lower system load, the spin limit for these locks are closer. . . . .                                                               | 98  |
| Figure 7.4 | Using Accelerator maintained lock information . . . . .                                                                                                                                               | 100 |
| Figure 8.1 | Single Thread Performance: Similar to native hardware, TNAPI performances better than MIX and SEP. . . . .                                                                                            | 105 |
| Figure 8.2 | Normalized versions shows how the difference between different configuration more closely, and TCA is always better than TNAPI. . . . .                                                               | 105 |
| Figure 8.3 | Normalized versions shows how the difference between different configuration more closely, and TCA is always better than TNAPI. . . . .                                                               | 106 |
| Figure 8.4 | Similar to native hardware, the application is more sensitive to the RX spin limit than the TX spin limit. Darker color means better performance. . . . .                                             | 107 |
| Figure 8.5 | Using IRQAcc improves tail latency when spin limit is not optimized. . . . .                                                                                                                          | 107 |
| Figure 8.6 | TCA helps reduce long latency on failure to acquire the lock. . . . .                                                                                                                                 | 108 |
| Figure 8.7 | Energy Savings on using TCA: during medium to low load, frequent interrupts consume more CPU cycles. For a four-core, 8 threads setup, using both TCAs can save up to 12% CPU cycles on CPUo. . . . . | 109 |

## LIST OF TABLES

|           |                                                                                                                                                             |    |
|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Table 3.1 | Server System Configuration. . . . .                                                                                                                        | 29 |
| Table 3.2 | Parameters used for workload generation. . . . .                                                                                                            | 36 |
| Table 4.1 | NAPI has a smaller hardirq region but a longer softirq and is also<br>unpredictable. On the other hand, tNAPI has no softirq but longer<br>hardirq. . . . . | 52 |
| Table 4.2 | Server System Configuration. . . . .                                                                                                                        | 53 |

## MY PUBLICATIONS

(Under Submission) Zhuo, Heng and Mikko H Lipasti (2023).

“Accelerator for Tail Latency”. In: *MICRO56*.

Daruwalla, Kyle, Heng Zhuo, and Mikko Lipasti (Nov. 2019a).

“BitSAD v2: Compiler Optimization and Analysis for Bitstream Computing”.

In: *Transactions on Architecture and Code Optimization* 16.4. DOI: [10.1145/3364999](https://doi.org/10.1145/3364999).

— (June 2019b). “BitSAD: A Domain-Specific Language for Bitstream Processing”.

In: *First ISCA Workshop on Unary Computing - June 2019*.

ISCA Workshop on Unary Computing (WUC). Phoenix, AZ, USA.

Daruwalla, Kyle, Heng Zhuo, Carly Schulz, et al. (2019).

“BitBench: A Benchmark for Bitstream Computing”.

In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems - LCTES 2019*.

The 20th ACM SIGPLAN/SIGBED International Conference.

Phoenix, AZ, USA: ACM Press, pp. 177–187. ISBN: 978-1-4503-6724-0.

DOI: [10.1145/3316482.3326355](https://doi.org/10.1145/3316482.3326355). (Visited on 01/25/2020).

Schlais, David J., Heng Zhuo, and Mikko H. Lipasti (Aug. 2020).

“Modeling Architectural Support for Tightly-Coupled Accelerators”. In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Boston, MA, USA: IEEE, pp. 253–262. ISBN: 978-1-72814-798-7.

DOI: [10.1109/ISPASS48437.2020.00045](https://doi.org/10.1109/ISPASS48437.2020.00045). (Visited on 08/15/2022).

— (Oct. 2022). “Work-in-Progress: NoRF: A Case Against Register File Operands in Tightly-Coupled Accelerators”. In: *2022 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*.

2022 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES). Shanghai, China: IEEE, pp. 33–34.

ISBN: 978-1-66547-296-8. DOI: [10.1109/CASES55004.2022.00028](https://doi.org/10.1109/CASES55004.2022.00028).

(Visited on 06/02/2023).

Zhuo, Heng and Mikko Lipasti (Feb. 25, 2023).

“TailWAG: Tail Latency Workload Analysis and Generation”.

In: *Proceedings of the 5th International Workshop on Benchmarking in the Data Center*.

BID ’23: 5th International Workshop on Benchmarking in the Data Center.

Montreal QC Canada: ACM, pp. 1–9. ISBN: 9798400701177.

DOI: [10.1145/3583060.3583170](https://doi.org/10.1145/3583060.3583170). (Visited on 05/30/2023).

Zhuo, Heng and Mikko H Lipasti (Apr. 2022). “Accelerator for Interrupt Handling”.

In: *ASPLOS 2022 WACI (WILD AND CRAZY IDEAS) SESSION*.

## ABSTRACT

Server performance has always been an active field of research, spanning topics including databases, network applications, and cloud computing. Tail latency is one of the most important performance metrics for server performance, next to mean latency and throughput. More recently, energy and power consumption also attracts attention for better efficiency. There are many factors that can have an impact on system performance, including software service design and hardware environments.

In the first part of this work, we study how different timing components affect server workload performance, including thread contention and interrupt handling. We introduce a workload analysis and generation tool: TailWAG. TailWAG relies on measured timing components from applications to generate a synthetic workload that mimics the original behavior. TailWAG can help server designers and application developers explore performance bottlenecks and optimization opportunities at early stages.

Then in the next part, we described how interrupt handling and its interaction with synchronization is one of the root causes of tail latency. In addition, we evaluate a recent feature addition to Linux kernel 5.12: threaded NAPI, which improves classic network interrupt handling with controllable scheduling, reducing the asynchronous penalty from the bottom half of network interrupt handling.

Based on the implication from the interaction, we take a further step by proposing a pair of tightly coupled accelerators: one for targeting the top half of the interrupt, and one for managing spin lock timeouts. In combination, these accelerators can prevent cascading timing disturbances for multi-threaded server workloads and save CPU energy consumption.

# 1 INTRODUCTION

With the current trend toward cloud computing, more and more focus has been attracted to server performance. Companies and researchers pay increasing attention to every aspect of cluster computing. Today's data centers contain large clusters of computers that exploit extreme levels of parallelism to execute complex workloads. Software and hardware have different paces of evolution. Server system hardware innovations and applications are evolving but at different rates. Hardware innovation usually goes through a detailed simulation and validation stage before implementation, which requires significant effort and a longer turnaround time. Conversely, applications change more rapidly and can evolve quickly via software patches. But share the same goal: lower latency, higher throughput, and better energy efficiency.

## 1.1 TAIL LATENCY

Simply having a cluster with thousands of powerful cores does not guarantee optimal performance (Delimitrou and Kozyrakis 2018); careful design and configuration, depending on the workload, are required (Kanev, Darago, et al. 2015; Kanev, Hazelwood, et al. 2014).

The performance metrics are consistent throughout all cases: throughput and latency. Key performance targets for throughput and latency can be improved by utilizing the CPU more effectively. In addition to mean latency, server applications also emphasize tail latency (95th or 99th percentile latency, worst 5% or 1% respectively of all requests), which is often used to specify the quality of service requirements for Service Level Agreements

(SLA) (Dean and Barroso 2013). Figure [ref{fig:introduction-tail-latency}](#) shows an example of latency distribution.

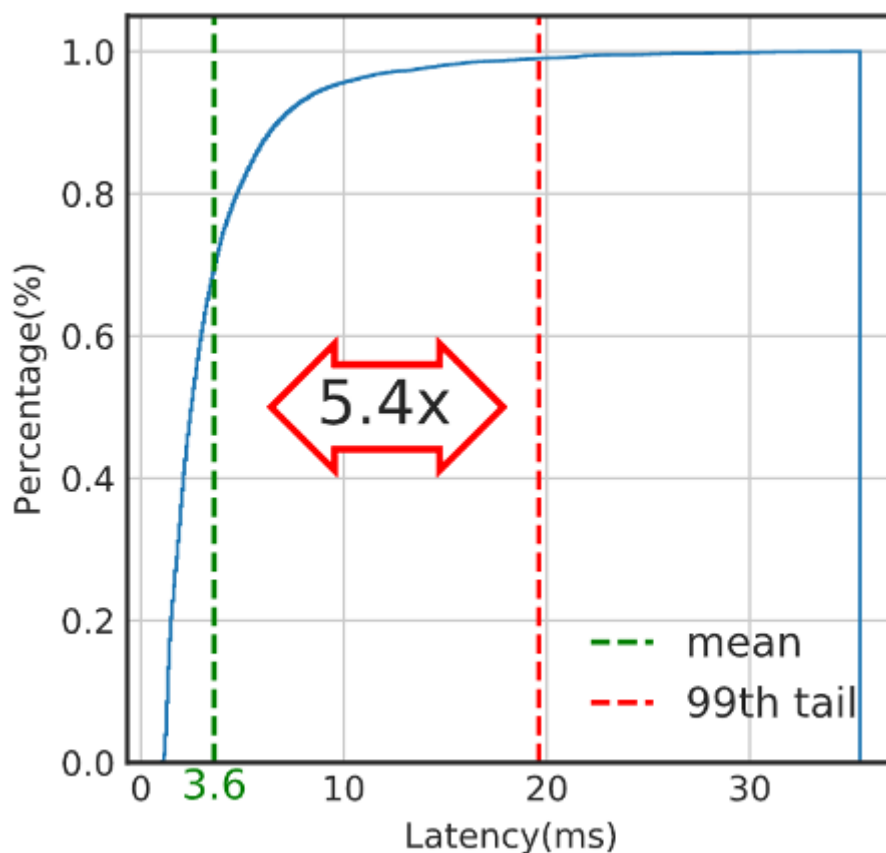


Figure 1.1: Blue color region shows the cumulative distribution of a service time. The worse 1% of service time can be 5.4 slower than average latency.

The tail latency may be acceptable for a single node system, where the local CPU calculates, and local storage (cache/memory/disk) keeps all the data. However, things can be complicated for a server system that is at scale, with multiple nodes, or even remote access for a distribution system.

As shown in Figure [1.2](#), one user request for a Google search often ends up accessing hundreds if not thousands of leaf functions. Furthermore, since many server workloads (e.g., map-reduce) rely on extreme levels of thread-level parallelism, the overall request latency is often determined by the slowest thread. With a long tail latency distribution, these late threads can severely impact system usability and throughput.

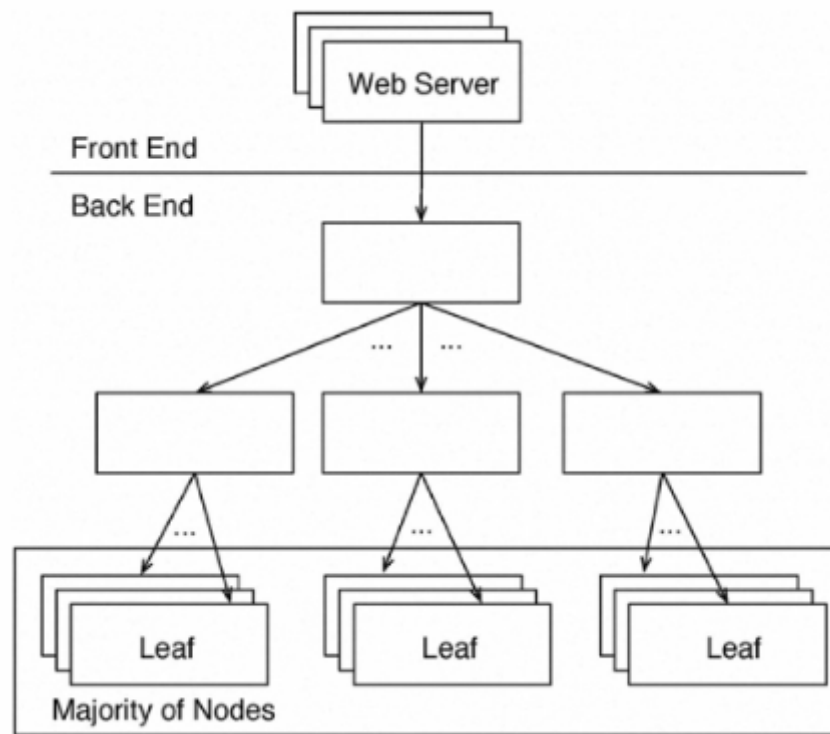


Figure 1.2: An application can break down into different tiers. Especially now, recent work and trends on microservices, and function as services on the cloud.

Server systems are challenging to evaluate due to complex and fragile workload configurations. Tuning the system to run for the best performance is very difficult and time-consuming. Changes to one configuration parameter can lead to very different performances. As one example, prior work (Belay, Prekas, Primorac, et al. 2017; Kasture and Sanchez 2016; Li et al. 2014) showed that using a dedicated core for interrupt handling can isolate server threads from interrupts and reduce tail latency.

We verified this with a sample application (silo) from Tailbench (Kasture and Sanchez 2016) under various scenarios. As shown in Figure 1.3, assigning interrupt handling and server threads to different cores produces lower latency at low to moderate load, but higher loads (at the expense of latency) are possible if all cores are used for both tasks.

However, the follow-up questions are non-trivial: How many cores are required for dedicated interrupt handling? Are they dynamically allocated or statically determined?

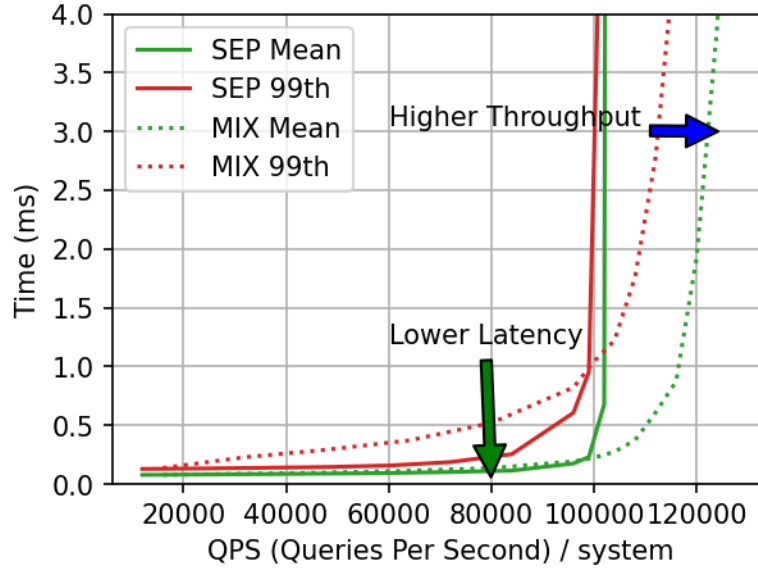


Figure 1.3: Latency vs Throughput. SEP: reserving core 1 for interrupt handling improves latency at low to moderate load, but lowers throughput; MIX: using all 4 cores for both interrupts and server threads yields higher throughput, with worse latency.

Also, more cores for interrupt handling means fewer cores for server threads. Using dedicated core (s) is an ad hoc solution that is only ideal for some scenarios.

Also, for any hardware features or configurations that are not available to the native machine, simulation becomes the only choice. Simulators provide flexibility and detailed exploration of architecture changes, but with the cost of extremely slow speed, such as gem5 (Nathan Binkert et al. 2011; Lowe-Power et al. 2020), and are not ideal for early-stage design exploration. ZSim (Sanchez and Kozyrakis 2013), on the other hand, trades off detailed microarchitectural implementation for faster speed. There are also other tools targeting RISC-V ISA, such as firesim (Karandikar, H. Mao, et al. 2018; Karandikar, Ou, et al. 2020; Pemberton and Amid 2021).

Thus, a microbenchmark that keeps the timing characteristics, but abstracts architectural details away, can be beneficial in identifying potentials and allows early-stage exploration of future hardware for server systems easily.

## 1.2 TIGHTLY COUPLED ACCELERATORS

Continuing trying to answer the question: How many cores are required for dedicated interrupt handling? More barriers are hiding in the current NIC interrupt designs making it more obscure.

Below we show an example of the call sequence invoked by the Linux kernel when an interrupt arrives; the code shown corresponds to roughly 2,000 dynamic instructions, and interrupting a server thread to execute this path can perturb that thread in multiple ways, leading to a cascading effect that ultimately increases service time by 2-3 orders of magnitude for the affected thread. More details are in [Chapter 5](#).

---

```

1 //generic irq
2 irq_enter_rcu();
3 __common_interrupt();
4 handle_edge_irq();
5 _raw_spin_lock();
6 irq_chip_ack_parent();
7 apic_ack_irq();
8 handle_irq_event();
9 __handle_irq_event_percpu();
10 // NIC irq
11 igb_msix_ring();
12 // NAPI
13 napi_schedule_prep();
14 __napi_schedule();
15 // scheduling
16 wake_up_process();
17 try_to_wake_up();
18 _raw_spin_lock_irqsave();
19 kthread_is_per_cpu();
20 ttwu_queue_wakelist();
21 raw_spin_rq_lock_nested();
22 _raw_spin_lock();
23 update_rq_clock();
24 ttwu_do_activate();
25 psi_task_change();
26 psi_flags_change();
27 psi_group_change();

```

```

28         enqueue_task_rt();
29         dequeue_rt_stack();
30         dequeue_top_rt_rq();
31         cpupri_set();
32         update_rt_migration();
33         enqueue_top_rt_rq();
34         ttwu_do_wakeup();
35         check_preempt_curr();
36         resched_curr();
37         task_woken_rt();
38         raw_spin_rq_unlock();
39         _raw_spin_unlock_irqrestore();
40 //generic irq
41     add_interrupt_randomness();
42     fast_mix();
43     note_interrupt();
44     _raw_spin_lock();
45 irq_exit_rcu();

```

---

Assigning interrupt handling to a dedicated core can avoid these effects and improve tail latency at low to moderate loads. Still, higher loads are only achievable if all cores are used for both tasks. We want to avoid these perturbations while still utilizing all cores for server threads and providing higher throughput.

General purpose CPU has the goal of handling all different workloads. It has a pipeline to hide access latency and mispredictions. Increasing clock frequency is a solution with a deep pipeline to push better performance until it hits Dark Silicon. A hardware accelerator is usually proposed to give better efficiency for specific tasks. Initially, floating point operation is an emulation function purely in software. When workloads started to use more floating-point operations, it made sense to have floating-point hardware support. It pushes the Floating Point Unit (FPU) to become a part of the Functional Units (FUs). More recently, more workloads like machine learning algorithms use matrix-matrix multiplication on GPUs.

For our specific target, NIC interrupt handling, since they occur frequently and are relatively lightweight, we advocate moving them to tightly-coupled accelerators (TCAs),

which are defined as accelerators that incur little overhead to invoke, can directly access memory via the CPU’s coherent memory interface, and obey standard in-order commit semantics just like any other CPU instruction.

Prior work has shown that TCAs are effective for workloads that frequently perform fine-grained, repetitive tasks easily mapped to hardware (Gope, Schlais, and M. H. Lipasti 2017).

Also, synchronization is required for many tasks while studying server workload, but the existing harness did not address this. Energy efficiency is still an issue even with a software solution of a hybrid lock of spinning and sleeping. Thus, we propose to offload software maintained synchronization also to a TCA.

### 1.3 SUMMARY OF CONTRIBUTIONS AND THESIS ORGANIZATION

Server system performance improvement needs pieces from all layers of the computing stacks. This thesis will address some of them by deeply analyzing existing workloads, locating bottlenecks, and proposing using hardware accelerators as solutions.

Below, we describe the organization of this thesis and list our main contribution:

1. Chapter 2 provides essential backgrounds and related work to our thesis. We showed how to evaluate server system performance with benchmarks. We discussed bottlenecks coming from Ethernet interrupt handling and synchronization mechanisms. We introduce tightly coupled hardware accelerators to solve short and repetitive tasks for better efficiency.
2. Chapter 3 performs a detailed analysis of Tailbench(Kasture and Sanchez 2016), an existing benchmark suite, breaking down timing components and abstract details from the individual application but providing high-level behavior analysis and contributing insights on latency and throughput. We introduce TailWAG, a single

configurable synthetic workload that enables swift parameter sensitivity analysis, and validate it against real hardware.

3. Chapter 4 dives into details of NIC interrupt handling API of NAPI. It provides the first performance evaluation of the threaded new API (TNAPI) network interrupt handling implementation introduced in Linux kernel version 5.12.
4. Chapter 5 explains how  $\mu$ s timing disturbance leads to ms scale latency in a server system. We showed the synchronization contentions from multi threads of accessing networking stacks. This effect was worse when combined with NIC interrupt handling, leading to cascading effects.
5. Chapter 6 proposes a TCA (Tightly-Coupled Accelerator) for NIC interrupt handling, motivated by the threaded NAPI and cascading effect from the interaction between it and the server application.
6. Chapter 7 targets another factor contributing to the cascading effect and proposes a TCA (Tightly-Coupled Accelerator) for the adaptive hybrid lock.
7. Chapter 8 describes our experimental setup for evaluating these two TCAs and shows prominent performance and energy efficiency results.
8. Chapter 9 summarizes this thesis and gives insight into potential future directions.

## 2 BACKGROUND AND RELATED WORK

Science and technology evolve and develop based on the history of the learning process. One of the ways to interpret research is to examine its existence, find its limitations and propose improvements, or even create a new solution. It is the same for this thesis. Before we present new work, essential backgrounds and related work must be introduced. This chapter will cover basic concepts around our thesis work, shaping the scale of our focus smaller. Also, we will cover related work in the corresponding area of study. This chapter is organized as below:

1. Chapter [2.1](#) introduces methods for performance analysis of a computer system, including tail latency concept and benchmarking.
2. Chapter [2.2](#) introduces our focus on hardware architecture, the network environment, and the system with an Ethernet connection.
3. Chapter [2.3](#) explains locking as a synchronization mechanism for modern-day software with multiple threads.
4. Chapter [2.4](#) introduces the hardware accelerator as a popular solution for better system performance.

### 2.1 SERVER PERFORMANCE ANALYSIS AND BENCHMARKS

Modern day systems are more complex than ever. There are different categories of computer systems, such as personal computers (PC), mobile devices, and servers. Sometimes, there is no distinctive line between them. For example, any device can become a server

if it hosts a software program and provides any “service” for other devices, such as doing calculations or storage. Server systems nowadays are clusters of machines, such as data centers and cloud computing providers. It is a rapidly changing area that draws much attention in academia and industry. As with any computer system, latency and throughput are the top performance metrics for server systems. Also, power and energy become more significant factors.

For classic metrics, latency usually means average latency only. For server systems, In addition to mean latency, server applications also emphasize tail latency (95th or 99th percentile latency, worst 5% or 1% respectively of all requests), which is often used to specify the quality of service requirements for Service Level Agreements (SLA) (Dean and Barroso 2013).

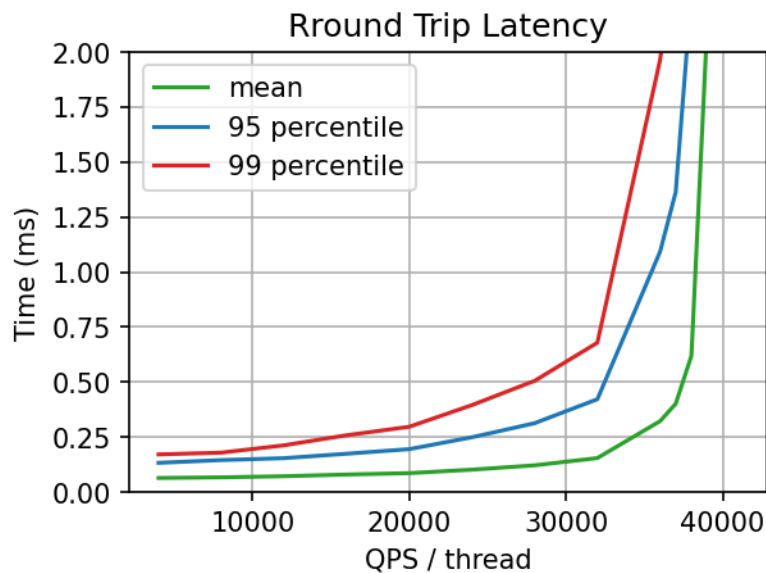


Figure 2.1: Latency: in server system, mean latency is important, same as tail latency, such as 95 percentile or 99 percentile. The application shown is silo from Tailbench, single threaded.

Example latency behavior is shown in Figure 2.1. During the lower load region, mean, and tail latency are closer. The difference is more significant when the system has a medium to higher load. Round trip latency is when clients send out the request until the client receives the request back. One of the main components of round-trip latency is

service time, where the server services the request, such as a data fetch for a database application. Moreover, there is communication latency between the server and clients through the Ethernet and queueing delays when the system becomes busier.

There are many reasons why the gap becomes bigger; time spent within the application itself is usually different. For example, in Figure 2.2, we show the sample experiment but the service time only, and it does not increase when the system becomes busier.

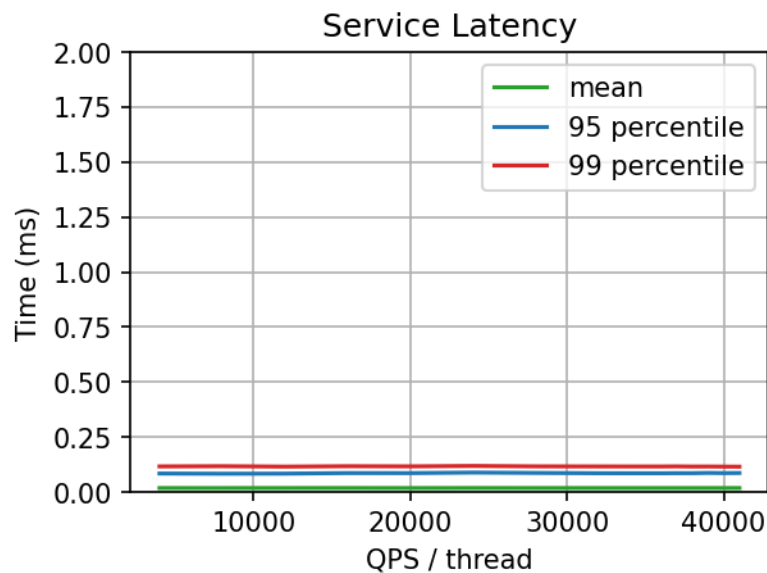


Figure 2.2: The service time itself usually does not change much, given enough computing resources. The application shown is silo from Tailbench, single threaded.

Here, we only showed one of the applications above as an example. Different application has different characteristic and behaves differently. Often, a monolith application that spans out into leaf functions has even more complicated system behavior. To ensure the SLA, more hardware resources are required (Barroso and Hölzle 2007; Barroso, Urs, and Parthasarathy 2019), and CPU utilization is always low (20% to 70%).

Furthermore, we only show one of the applications above. Applications have different characteristics and behave drastically differently under different environments, such as compiler versions, core frequency, and scheduling policies. Therefore, evaluating a system somewhat is a crucial motivation for the benchmark suite. Identical to the classic

single-core computer system paradigm, benchmark suites are used for server system performance evaluation. Therefore, a reliable and comprehensive suite that covers and tests different aspects of a system is essential and can be as little as cache and branch prediction.

Big companies with their server system can use internal workload-generated traces and data for research, such as Microsoft Azure, Google Cloud, and Amazon Web Services. Classic benchmark suites in academia can also be used for server systems like SPEC and PARSEC. But, they are designed for single core or single node systems with limited implications, intended for micro-architectural research. Motivated by this, there is more recent work on pushing server scale benchmark suites within the academia.

Some domain-specific suite includes Tonic Suite(Hauswald, Y. Kang, et al. 2015) for machine learning, Big Data Benchmark(Pavlo et al. 2009) for MapReduce, Sirius(Hauswald, Laurenzano, et al. 2015) for Apple Siri.

Also, there are benchmark suites that consist of different types of applications for wider scopes, such as CloudSuite(Ferdman et al. n.d.), BigDataBench (L. Wang et al. 2014), and Tailbench(Kasture and Sanchez 2016). All these three examples provide a set of applications but with some limitations. CloudSuite uses YCSB(Cooper et al. 2010) load testers that model a closed-loop system that has bounded individual requests from clients to specific server threads deterministically. Both CloudSuite and BigDataBench focus more on the microarchitectural analysis of the system. Tailbench, on the other hand, includes a broader range of latency, from microseconds to seconds. In addition, Tailbench provides three different sets of harnesses that can mimic both close-loop and open-looped systems. After comparison, Tailbench was picked and used as the benchmark suite for our research. Details breakdown will be in Chapter 3.

## 2.2 NETWORK AND INTERRUPT HANDLING

Advanced Research Projects Agency Network (ARPANET), invented in 1969 by DARPA (was called ARPA back then), was the first to connect computers. The first local connection map graph was very complicated, based on the famous TCP/IP protocol. TCP/IP specifies how data should be put into packets, how machines should be addressed, and how data should be transmitted, routed, and received between devices.

In TCP/IP definition, there are four layers:

1. Application layer, interface for most software programmer usage: web browsing (HTTP), File Transfer Protocol (FTP), Email (SMTP), etc.
2. Transport layer: Transmission Control Protocol (TCP), User Datagram Protocol (UDP).
3. Internet layer: Internet Protocol (IPv4, IPv6), Internet Control Message Protocol (ICMP).
4. Link layer: device drivers, firmware, and physical components, such as Ethernet devices.

Another more comprehensive model is the Open Systems Interconnection model (OSI). OSI has 7 layers instead of 4. OSI breaks the Application layer into Application, Presentation, and Session.

In addition to the application already in TCP/IP, it has: 1. Presentation layer: specifies translation (ASIC->machine code), Compression (lossy, lossless) in video/audio streaming, Encryption/Decryption (SSL, secure sockets layer). 2. Session layer: specifies APIs (Application Programming Interfaces), Session Management, authentication, and authorization. Example: mail user getting emails from the account hosting server.

Also, the Link layer is separated into Data Link and Physical, and they are:

1. Data Link layer: frame level, where Media Access Control (MAC) and Logical Link Control (LLC) Physical Addressing layer, meanwhile Network use logical addressing. Data link uses MAC address, access the media (framing), and controls how data is placed and received from the media (between router to satellite to server host router).
2. Physical layer: lowest level, such as wired cable, WIFI, Bluetooth. It is only called physical because wireless technology was not created back when OSI was created. Thus, any component that sends and receives bits are physical layers.

Both OSI and TCP/IP are well-known in the Networking society. No matter which model is used, much search work is going on at every level. Numerous components contribute to a server system's tail latency, including resource sharing from other applications or background daemons to garbage collections [Cai et al. (2021);].

Our research focused on the network stack and especially the Ethernet interrupt handling. Interrupt handling (Gleixner and Molnar 2010) is an important core-API layer and is commonly defined as synchronous (exception, software) and asynchronous (timer, driver hardware) (Bovet and Cesati 2006). The Linux kernel is equipped with a mixture of interrupt and polling design, named NAPI (LinuxFoundation 2016). Details and the new improvement of NAPI are in Chapter 3.1.3. Next, we describe some example runtime configurations and related work in network offloading and power management.

### 2.2.1 Runtime Configuration

Many things can affect system performance during kernel or OS configuration runtime, such as memory allocations and interrupt mapping. For instance, in the listing below, we show how the runtime tcp memory limit can be altered:

---

```

1 sysctl -w net.ipv4.tcp_timestamps=0
2 sysctl -w net.ipv4.tcp_sack=0
3 sysctl -w net.ipv4.tcp_rmem='16777216 16777216 16777216'
4 sysctl -w net.ipv4.tcp_wmem='16777216 16777216 16777216'
5 sysctl -w net.ipv4.tcp_mem='16777216 16777216 16777216'
6 sysctl -w net.core.rmem_max=16777216
7 sysctl -w net.core.wmem_max=16777216
8 sysctl -w net.core.rmem_default=16777216
9 sysctl -w net.core.wmem_default=16777216
10 sysctl -w net.core.optmem_max=16777216
11 sysctl -w net.core.netdev_max_backlog=10000

```

---

Modern Linux kernel provides mechanisms directing traffic, deciding where packets go when NIC receive them, such as Receive Side Scaling (RSS) and Receive Packet Steering (RPS). RSS can be used for spreading traffic across different cores. This feature is provided in modest server NIC, with multiple receive queues. There is an indirect table with assigned weights. Whenever a new packet comes in, it chooses a queue based on a 4-tuple (client address, client port, server address, server port), which means one connection will always send traffic to a specific queue.

### 2.2.2 *Kernel Bypassing*

After many years of developing hardware architecture and software applications, people realized some of the classic TCP/IP model methods need rework. Several fields of study offload the pressure from the networking stack, both on the software and hardware levels. Traditional network stack has both data and control combined into one piece, and the first order of approach is to separate them into two: the data plane and control plane,

Kernel bypassing is a commonly used concept in software methods for the Linux kernel. The key idea is to have data directly consumed by the user in user space instead of going through layers of the stack within the kernel. Many NIC vendor provides customized solutions for user space kernel bypassing for packet handling through soft-

ware libraries. However, a community force is driven to seek a unified solution. DPDK (Data-plane Processing Development Kit) is an open-source tool led by Intel for kernel bypassing. DPDK map NIC to user space that can move packet around. It does not provide TCP but can develop TCP on top as a backend base.

Academia also has progressed based on the DPDK approach. IX (Belay, Prekas, Kozyrakis, et al. [n.d.](#); Belay, Prekas, Primorac, et al. [2017](#)) is a data-plane architecture that uses hardware virtualization for network processing in user space. Similarly, Snap (Marty et al. [2019](#)) is a microkernel that provides fine-grained dynamic scaling and uses a different communication stack called Pony Express. Netmap (Rizzo [n.d.](#)) is halfway in between, user space control but packets resident in kernel space for minimum movement. The application does not need to copy data to user space before processing, essentially reducing data movement, which is the primary goal.

The benefits are specialized acceleration for the network process. However, this comes with trade-offs. These approaches require dedicated computer resources for polling packets. It does give lower latency but also lower throughput since some CPUs are not available for server applications.

Also, customization reduces compatibility, and one solution will not be viable when the system changes with a different NIC card. Even within a community, there needs to be a unified standard of APIs on portability, and it has a high barrier of knowledge required for low-level optimizations. User space network management such as DPDK lost the NIC part for kernel visibility. Even simple ssh or more complicated NGNIX need a port bind, which requires backend POSIX library support, and the kernel cannot see it anymore. Furthermore, the additional threads created can affect kernel scheduling and threads interactions.

In our work, we propose a solution that uses configurable TCA logic that can serve a wide range of applications without much burden on software programmers.

### 2.2.3 *Network Offloading*

In a distributed system, storage and computation are not performed in the same node. Instead, different nodes are connected using switches and routers. For storage virtualization, data in other storage nodes are presented as centralized storage, similar to virtual memory in the kernel. Like kernel bypassing in software, one can have a hardware solution for the decoupled control and data planes.

With a classic CPU interface, NIC can handle up to 10Gbps of traffic without losing the usability of the CPU. Then, it makes sense to do network offloading to specialized hardware for acceleration. These should all be under the umbrella: SmartNIC, DPU, etc. Smart NIC uses RDMA (Remote Direct Memory Access) and RoCE (RDMA-over Converged Ethernet).

RDMA (Remote Direct Memory Access) is a remote version of DMA. DMA allows data movement between devices and memory without the CPU touching it. RDMA is within NIC, as a similar engine, using a pointer and counter to move data. The network stack does not handle them. The ethernet of RDMA is RoCE (RDMA over Converged Ethernet). These interfaces make sense when we have a distributed system where multiple servers can access data stored in multiple storage devices.

SDN (Software Defined Network) defines an application-level networking mechanism (Sezer et al. 2013). When the system scaled up, it became difficult to maintain different applications and platforms supported by a single cluster. SDN was proposed as a solution to unifying the software-level control flow. SDK stack has the applications and control plane, including network topology, forwarding, and routing, link management. Networks are centralized and controlled, and the infrastructure is abstracted from applications. This survey (Yan et al. 2016) shows good features from SDN on defending against potential DDOS attacks.

SDS (Software Defined Storage) (Thereska et al. 2013) also defines the storage mechanism on the application level. SDS acts based on application need, using a user-level policy process for storage provision. SDS and SDN share the same philosophy: separate hardware and software and control them in different places. The downside of SDS and SDN is that they are not optimized for specific hardware and can get complex when the system component adds up.

For virtualized environment specifically, SR-IOV (Dong et al. 2010; *Overview of Single Root I/O Virtualization (SR-IOV) - Windows Drivers | Microsoft Learn* 2023), Single-Root I/O Virtualization standards extend PCIe (PCI Express) and allows a NIC device to be accessed by multiple VM (Virtual Machines) directly.

To more extreme, VmWare (*Announcing Project Monterey - Redefining Hybrid Cloud Architecture - VMware vSphere Blog* 2023) started project Monterey, which uses NVIDIA Bluefield Data Processing Unit (DPU) in 2020. The DPU is an advanced version with 8 Arm cores of 2.3GHz frequency, and 2x100G smart NICs can even run its own OS, such as Ubuntu.

In our work, our proposed accelerator does not handle as much work as SmartNIC. However, we share the same goal: let the CPU do what it does the best, general workload, and offload frequent work to hardware, but only network interrupt handling.

#### 2.2.4 Fast Interrupt Handling

There are software interfaces within interrupt handling logic in the Linux kernel for fast interrupt handling. For a system such as ARM, it embeds the feature of Fast Interrupt (FIQ), which is a higher priority class of interrupts. It has been used for a system that has a distinct highest priority task, such as real-time applications for industrial scenarios (Souza et al. 2020). The FIQ can only be used for one interrupt routine at a time. FIQ has dedicated registers for its routine, so it can prevent context switches latency since

no pushing and popping are required, and it also ensures value persistence between invocations.

The disadvantage of using FIQ is mainly the effort for customization. This requires the kernel designer to write specific assembly code for specific interrupts and cannot be changed afterward.

Our work is similar in register usage, such that both TCA and FIQ have reserved registers for specific interrupts. However, our work provides an instructions interface that can reconfigure the accelerator logic afterward. Furthermore, our target interrupt does not require to be the highest priority.

### 2.2.5 *Power Management*

Furthermore, the server system is power-hungry as the system becomes massive. NMAP (K.-D. Kang et al. 2021) studied latency impact from Ethernet Interrupt handling (NAPI) when a system uses different governors, such as on-demand and performance. It proposed a power management tool adjusting voltage and frequency based on NAPI interrupt or polling mode. In our study, we carefully set our system on fixed frequency using the user space governor, disabling deep sleep state and hyperthreading.

Power and energy are other reasons for using dedicated hardware accelerators, which will be covered more in Chapter 2.4.

## 2.3 SYNCHRONIZATION MECHANISM

Parallel applications require a synchronization mechanism between threads. Locking is the most commonly used approach. Locking is an abstraction that provides mutual exclusion in that only one access unit can enter the critical section. The access units can

be a thread for a multi-thread program; or a thread from a different application that tries to access system resources, such as memory management.

In addition, different lock algorithms can be implemented for locking. For instance, the lock transfer ownership can be a competitive or direct handoff. Competitive succession makes all waiting threads make atomic access, and one non-deterministic winner stands out; direct handoff has a deterministic successor and implicitly maintains a queue structure.

Furthermore, there can be different waiting policies when contention is detected. For example, the waiting thread can choose to give up hardware resources and put itself to sleep immediately, or it can occupy the CPU and spin, waiting for the lock to be acquired.

While locks have been used to synchronize critical sections for decades, the question of which flavor of lock to use still needs to be answered (Michael [n.d.](#)). Guerraoui et al. surveyed and evaluated a wide array of locking strategies and proposed a set of six guidelines for programmers to consider when deciding which lock to use in a particular use case (Guerraoui, Guiroux, et al. [2018](#)). However, from its observation, even with the best lock design, it is only best for 53% of all the applications they study.

From a pragmatic perspective, most programmers are not prepared to undertake this level of analysis and fall back on default options like `pthread_mutex` since they are widely supported and easy to understand. However, even given this default choice, further questions arise, such as whether to sleep on a failed lock acquire, spin, and wait and if so, for how long.

A hybrid approach of spinning before sleeping was first suggested by Ousterhout (Ousterhout [n.d.](#)), and subsequent work proposed a complex analytical solution for determining the optimal wait time (Boguslavsky et al. [1994](#)). This analytical solution, while effective, is difficult to deploy in a practical design, so more recent approaches have relied on heuristics. For example, the MUTEXEE design spins for 8,000 cycles before

sleeping, setting the threshold based on an estimate of the round-trip latency for entering and exiting supervisor mode (Falsafi et al. 2016).

Spinning is inherently inefficient, and prior work has examined various approaches for reducing the cost by slowing the spin loop with pause or barrier instructions, as discussed in (Falsafi et al. 2016). Here, we use an approach similar to the one deployed to reduce idle loop power consumption: the Linux kernel uses the `monitor/mwait` instructions to place the CPU into a low-power state until a specified memory location is updated (“Intel® 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4” 2023). The memory update is detected by monitoring coherence activity for that address. Dutu et al. have recently proposed extending `monitor/mwait` instructions to coordinate and synchronize workgroups in GPUs for better energy efficiency (Dutu et al. 2020).

In Chapter 7, we use `MUTEXEE` as one of our baselines of comparison and propose an accelerator to determine the spinning time dynamically. We then use the `monitor/mwait` approach to inexpensively wait for a change in the lock variable but encapsulate that operation into the lock acquire operation.

While likely safe in kernel mode, user-mode versions of `monitor/mwait` have been shown to enable timerless attacks similar to the Spectre family (R. Zhang et al. 2023). Our approach exposes a limited interface to the user, making it arguably more difficult to deploy for security attacks. Detailed analysis of that potential vulnerability is left to future work.

## 2.4 HARDWARE ACCELERATORS

Hardware accelerator is a field that consists of solutions that help general purpose CPUs on handling specific tasks. The accelerator is a board term, and it is relative to general-purpose hardware. For instance, a Graphics Programming Unit (GPU) was once

an accelerator when explicitly designed to accelerate image processing in a frame buffer for display devices. Then, some other parallel workloads can be executed on GPU. This makes General-purpose computing on GPU (GPGPU).

Many prior works have identified potential in different domains throughout the history of hardware accelerator development. Some examples are DaDianNao(T. Chen, Du, and Sun [n.d.](#)), PuDianNao(Liu et al. [2015](#)), TPU(Jouppi, Hyun Yoon, et al. [2021](#)) For machine learning; Widx(Kocberber et al. [2013](#)) for database hash index lookup; Huang et, al. (Yu-Wen Huang et al. [2005](#)) proposed accelerator for H.264/AVC video encoding. These accelerators can have orders of magnitude in performance improvement.

As the system becomes more complicated, more accelerators are proposed and embedded into the system, and it becomes a sea of accelerators (Y.-T. Chen et al. [2013](#)). This trend is both in large-scale systems and smaller-scale edge devices. The big-scale example is the IBM Power Edge of Network processor (Brown et al. [2011](#)), which includes individual Crypto, RegEx, Compress, and XML engines. Small-scale includes mobile phones used nowadays with system-on-chip (SOC). This style of loosely coupled accelerator makes sense for a system that provides robust functionality.

Nowatzki et al. (Nowatzki et al. [2016](#)) observed Domain Specific Accelerator (DSA) design targeting these fields: concurrency; computation; communication; data reuse; and coordination. Most accelerator designs explore opportunities and propose hardware solutions based on any one (or more than one) aspect.

There are tradeoffs between using general-purpose CPUs and domain-specific hardware accelerators, including design cost, area, and energy. Furthermore, invoking the accelerator requires data offloading and communication between components. Thus, many hardware accelerators are targeting the coarse granularity of work. However, routines on a smaller scale, ranging from 10-1000 instructions, are being executed a lot. This motivates tightly-coupled accelerators (TCA) closer to the CPU pipeline, reducing innovation costs. Prior work has shown that TCAs are effective for various workloads that

perform fine-grained, repetitive tasks easily mapped to hardware. For example, Kanev et al. (Kanev, Xi, et al. 2017) argue that user-space heap management is a good candidate, while Gope et al. (Gope, Schlais, and M. H. Lipasti 2017) showed that PHP applications benefit from accelerators for string handling and hash maps.

We also contributed with a validated analytical model that argued that high-performance cores need to integrate TCAs to minimize invocation overheads (Schlais, Zhuo, and M. H. Lipasti 2020) closely.

## 2.5 SUMMARY

This chapter introduced the background and related work to this thesis. First, we started with performance metrics on evaluating server systems and introduced the concept of tail latency. There are many factors contributing to the performance. Then, we introduced the main focus of our thesis, which is networking stack and interrupt handling. Furthermore, we described hardware accelerators as being one of the solutions for performance and efficiency. Finally, a similar philosophy can be applied to thread synchronization.

### 3 WORKLOAD ANALYSIS AND GENERATION

Server performance has always been an active field of research, spanning topics including databases, network applications, and cloud computing. Cloud computing relies on utilizing many compute nodes across multiple layers of abstraction. Often, single requests are broken down into multiple sub-requests, and overall performance can be determined by the most poorly-performing sub-requests, i.e.,  $n$ -th percentile tail latency. As a result, tail latency is becoming a more important factor as a measure of the quality of service (Dean and Barroso 2013; Sriraman and Wenisch 2018; Yanqi Zhang, Gan, and Delimitrou 2019).

Sriraman and Wenisch studied how different OS operations affect server latency using the eBPF tool (*BPF Documentation — The Linux Kernel Documentation* 2022; Sriraman, Dhanotia, and Wenisch 2019). These operations include hardware, network software interrupts (both receive and transmit), and scheduling actions. Not surprisingly, Active-Exe (active thread servicing the requests) and Net (net mid-tier latency) contribute the most to service time. Tiny components contribute minimally to average latency but can cause a severe cascade tail latency problem (Belay, Prekas, Primorac, et al. 2017). Other work such as Multilate (Leverich and Kozyrakis 2014) and Treadmill (Yunqi Zhang et al. 2016) are recent studies on load testers for evaluating server systems and looked into pitfalls, including co-location applications.

Server systems are challenging to evaluate due to complex and fragile workload configurations. Tuning the system to run for the best performance is very difficult and time-consuming. Changes to one configuration parameter can lead to very different performances. Furthermore, many system parameters can dramatically affect performance (Dean and Barroso 2013; Kasture and Sanchez 2016). Features including sleep states and

voltage/frequency scaling are helpful for energy management but can induce unexpected latency to requests, distorting the tail latency distribution. Also, Treadmill (Yunqi Zhang et al. 2016) pointed out client-side queuing can generate biased results and should be avoided by using multiple clients. In Chapter 1.1, we showed the latency versus throughput based on interrupt handling.

In addition, the runtime scheduler can also have an impact. Setting a real-time priority for server threads avoids competition with lower-priority tasks and can be used to ensure lower latency. But, for safety reasons, the kernel prevents real-time priority tasks from consuming 100% of the CPU, leading to the latency hiccups shown in Figure 3.1. Server maintainers can choose to set -1, which means there is no limit. However, this may lead to potential starvation as interrupt handling can be delayed indefinitely under heavy load.

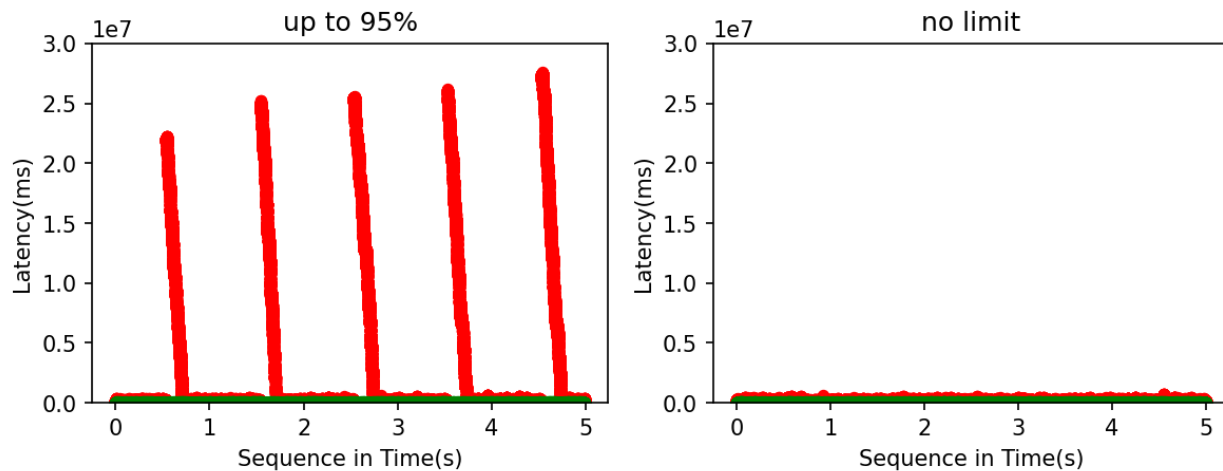


Figure 3.1: Linux real-time scheduling limit: The Linux kernel default limit for real-time tasks is 95% of total CPU time. When real-time tasks reach the limit, they are preempted for 50 ms, leading to the latency spikes shown above (left). Over-riding this setting to unlimited CPU utilization eliminates most of the latency spikes (right) but may cause starvation. Hardware setup in Table 3.1.

Server system hardware innovations and applications are evolving but at different paces. For example, hardware innovation usually goes through a detailed simulation and validation stage before implementation, which requires significant effort and a longer

turnaround time. Conversely, applications change more rapidly and can evolve quickly via software patches.

For research purposes, benchmarks can be executed in a simulated environment to examine system behavior under different combinations of a user-specified disk image, kernel, and modification to the hardware system, and any improvement or innovation can be observed and evaluated. However, a noticeable gap exists between performance on real hardware and simulation. One of the reasons is that real machines have more unexpected background noise than the simulated environment. As a result, it is almost impossible to replicate the same performance over different setups, core frequencies, cache sizes, and Ethernet hardware.

Moreover, hardware simulation is time-consuming. For simulation data points, one second of hardware machine time costs hours to days of simulation time on gem5 full system mode (Lowe-Power et al. 2020). ZSim can provide improved speed but with less microarchitectural detail, and it still requires environmental build and setup. Before proposing techniques to accelerate interrupt handling, we need to demonstrate the effects of these timing disturbances in a controlled fashion.

A synthetic workload can be representative, easy to tune, and quick to converge, making it ideal for early-stage design exploration. Similar work in other domains includes SynFull (Badr and Jerger 2014) for evaluating NoC traffic and STMBench7 (Guerraoui, Kapalka, and Vitek 2007) for software transactional memory.

In this chapter, we introduce TailWAG, a model for analyzing existing applications and generating synthetic workloads. The high-level workflow is shown in Figure 3.2. The key advantages of this synthetic workload are:

- TailWAG is simple, hardware independent, and easy to run natively or in simulation.
- TailWAG has repeatable behavior and measurements.
- TailWAG enables exploration of future/emerging/evolving workloads and hardware platforms before they exist.

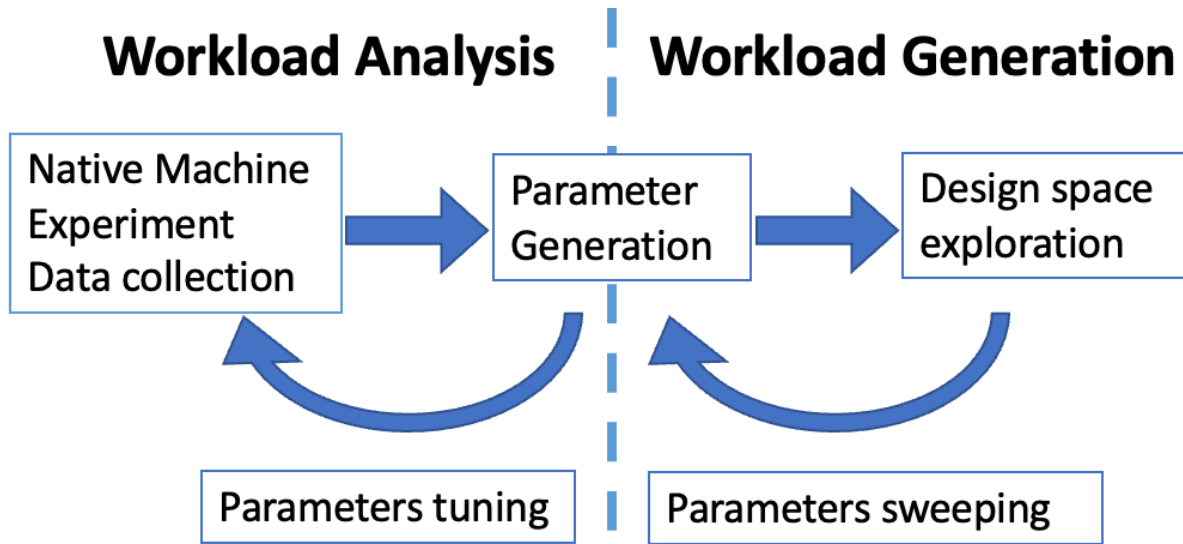


Figure 3.2: TailWAG workflow.

TailWAG abstracts workload performance into timing components, and the timing parameters can represent latency components from different abstraction layers. Thus, TailWAG can be used for different target users, and examples of ways to explore:

- Software developer: changes of optimizations on software applications, usage of different system libraries, such as Huge Pages and specific heap manager.
- Server maintainer: bottlenecks from runtime configurations, such as threads scaling issues, core binding for servers, and interrupt handling that leads to timing disturbance.
- Kernel developer: evaluate module interactions between components, proposing new kernel features such as threaded NAPI.
- Hardware designer: explore future hardware design that offloads current software routine, and impacts on specific timing parameters.

The rest of this chapter is organized as follows:

1. Chapter 3.1 performs a detailed analysis of Tailbench applications, breaks down timing components, abstract details from the individual application but provide high-level behavior analysis, and contribute insights on latency and throughput.

2. Chapter 3.2 goes over workload generation and validation against real hardware.
3. Chapter 3.3 showcases the ability of TailWAG to make performance predictions and implications of improvement.

### 3.1 WORKLOAD ANALYSIS

From an abstraction layer of view, there are two orthogonal paths for improving the performance of a server workload: application developers can work to reduce service time, and service providers can reduce run time environmental overheads and noise. This section addresses the application service itself, thread scaling, and timing disturbance from systems, such as NIC interrupts.

First, we describe the applications and the setup from Tailbench (Kasture and Sanchez 2016). There are eight applications with varying attributes: **silob** (Tu et al. 2013), which is an in-memory transaction database; **specjbb** (spec 2015), which is a Java middleware warehouse server benchmark; **masstree** (Y. Mao, Kohler, and Morris 2012), which is a key-value store database with mycsb-a dataset (50% get and 50% set); **img-dnn**, image recognition with MNIST dataset (Li Deng 2012); **xapian** (Xapian 2022), an online search engine with Wikipedia English version; **shore** (Johnson et al. 2009), on disk transaction database with TPC-C; **sphinx** is speech recognition with CMU AN4 alphanumeric database (Walker et al. 2004); **moses** (Koehn et al. 2007) is real-time translation, fed with opensubtitles.org English-Spanish corpus snippets.

The Tailbench harness provides three configurations: integrated (single process handles both server and client, communicating via memory mapped buffers); loopback (separate server and client process on same node communicating via TCP using localhost); and networked (server and client are on different nodes, communicating via Ethernet). All analysis experiments are conducted on a networked setup, and TailWAG

uses the integrated setup for simplicity. Server system hardware setup is listed in Table 3.1.

The round trip latency and service time of every query are collected from the server system for all experiments. Afterward, analysis and generation can be performed on any machine. Python Scipy (Virtanen et al. 2020) library was used for analysis.

Table 3.1: Server System Configuration.

| Model            | Supermicro SYS-2029GP-TR            |
|------------------|-------------------------------------|
| Cores            | Cores Intel Xeon Gold 6128, 3.5 GHz |
| Caches           | 32KB L1, 6MB L2, 19.25MB L3         |
| Main Memory      | 96GB, 1333 MHz                      |
| Operating System | Ubuntu 22.04, Linux kernel 5.15.0   |

### 3.1.1 Service Time Distribution

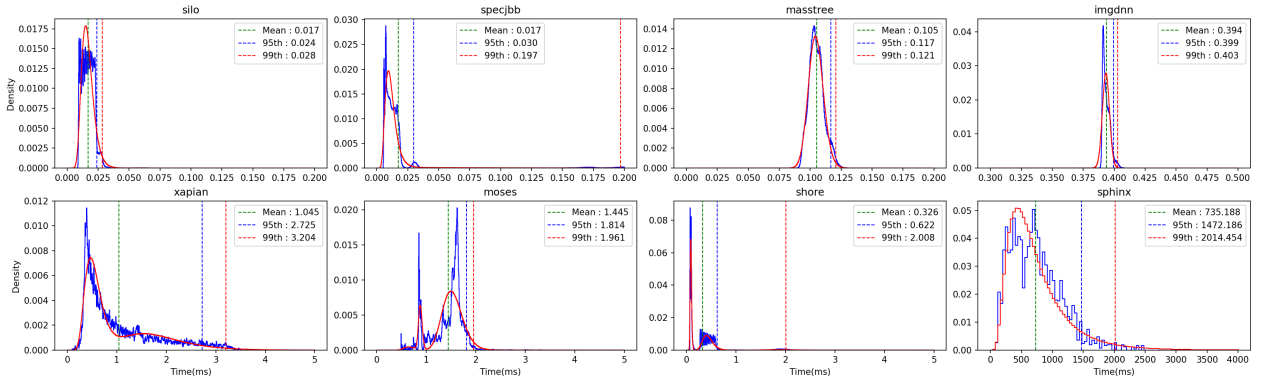


Figure 3.3: Service Time Distribution: Blue line shows the measured service distribution; Red line shows distribution coverage used in TailWAG workload generation. Python Scipy is used for distribution fitting.

First, we break down application service time, which consists of active CPU time spent on computation, memory access such as caches, and I/O. CDF ( Cumulative Distribution Function) was reported in the Tailbench paper (Kasture and Sanchez 2016).

To better understand and analyze the difference statically, we use PDF (Probability Density Function). This also helps us abstract characteristics into TailWAG parameters. As shown in Figure 3.3, we can build PDF from:

- One centralized (low variance) continuous sample distribution: silo, over 99% are under 40  $\mu$ s; masstree, over 99.9% are within 75 – 150  $\mu$ s; img-dnn, all service times are within 0.35 – 0.45 ms.
- One wide continuous sample distribution: xapian, spanning 0 – 4 ms; mores, over 99% ranging from around 0.6 – 3 ms; sphinx, ranging up to 3 s.
- Two distributions: specjbb, around 94% are under 25  $\mu$ s, 3% are between 25 – 40  $\mu$ s; shore, in which around 50% are less than 200  $\mu$ s, 45% are within 300 – 800  $\mu$ s.
- To give 100% coverage, for instance, additional distributions for specjbb: around 3% of requests are within 150 – 250  $\mu$ s.

$1/(\text{Service time})$  indicates the theoretical upper bound for how many requests the server can handle, i.e., shorter service time leads to a higher achievable load QPS (queries per second).

In addition to service time, round trip latency in networked configurations includes TCP stack and NIC (Network Interface controller) latency. The round-trip latency starts when the query is generated on the client side and ends when the client receives the response from the server.

Networking activities affect the server as load increases and can lead to cascading effects.

### 3.1.2 Thread Scaling

Server applications are written as multithreaded for higher throughput, utilizing multiple cores or multi-node server systems. Even with well-designed multi-threaded applications,

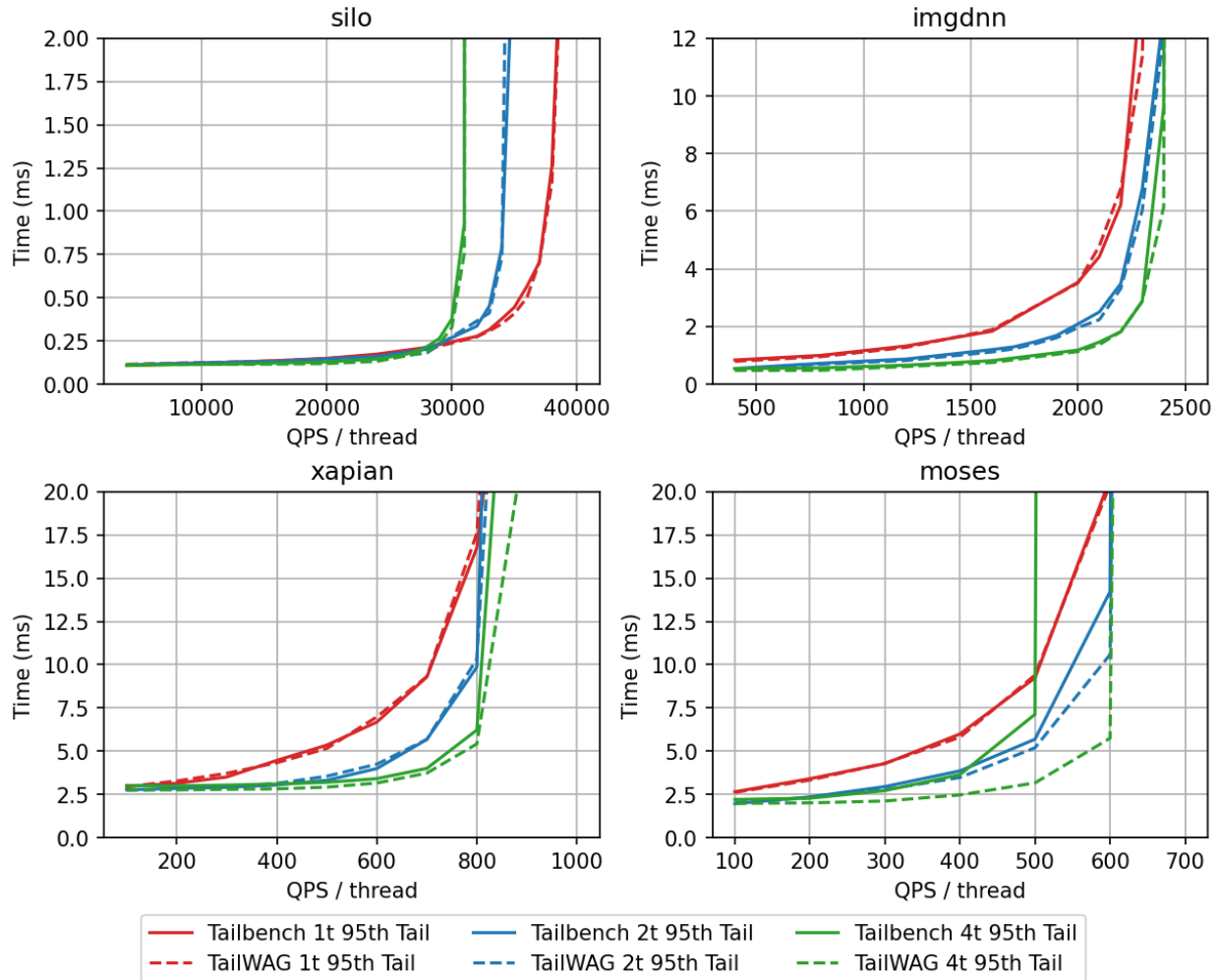


Figure 3.4: Thread Scaling (95th tail) for 1/2/4 server threads. Dashed lines show TailWAG, and solid lines are measured. TailWAG matches behavior well in all cases except moses, where it reflects an upper bound without its memory bottleneck. TCP contention impacts scaling for silo.

achieving linear scaling on performance is challenging due to resource contention. There are unavoidable minimum penalties that can be as small as microseconds.

Not all Tailbench applications scale up well for different reasons. Half of the applications have close to linear scaling: masstree, img-dnn, xapian, and sphinx since they have longer service and less critical section sharing within the application. Due to open loop configuration (multiple clients to multiple server threads), using more threads can hide some longer service requests. Figure 3.4 shows selected examples.

Applications not scaling well include silo, specjbb, shore, and mores. Moses was reported to have a memory bottleneck in Tailbench (Kasture and Sanchez 2016). Shore was only provided a single warehouse database, which cannot scale as intended. Silo and specjbb share the same root cause as *TCP send()/recv()* behave as a high-contention critical section due to very short service times.

Executing TCP stack code as part of the harness can be easily overlooked when deployed. Multiple threads share a server port in the Tailbench harness, requiring serialization. Measurements show that networking system calls, including *recv()* and *send()*, cost from 3  $\mu$ s to over 10  $\mu$ s, depending on the request size and response. This latency is harmless for applications with mean latency at a millisecond scale. Still, it can be crucial for small service time applications, such as specjbb and silo, where mean service latency is under 100  $\mu$ s.

### 3.1.3 NIC Interrupts

Interrupt handling (Gleixner and Molnar 2010) is an important core-API layer and is commonly defined as synchronous (exception, software) and asynchronous (timer, driver hardware) (Bovet and Cesati 2006). The Linux kernel is equipped with a mixture of interrupt and polling design named NAPI (**NetworkingNapiWiki2016**). When a packet arrives (receiving process), the NIC hardware interface raises an asynchronous hardware interrupt request (IRQ); the system is notified, the CPU saves the current program state, switches context, and enters the interrupt handling routine (hereafter referred to as *hardIRQ*).

During the *hardIRQ*, the CPU will disable preemption and interrupts, so only a minimal amount of work should be performed: (*napi\_schedule*) function raises *NET\_RX\_SOFTIRQ*, schedules the software interrupt routine (hereafter *softIRQ*) and clears the interrupt line. On the other hand, *softIRQ* is preemptible and serves the pri-

mary function of polling network packets into specified user buffers depending on the protocol and applications.

Also, as part of the generic interrupt handling layer, before the hardIRQ exits with the function call `irq_exit`, it always tries to `invoke_softirq` if there is a softIRQ waiting and the current time quantum has not elapsed.

These two mechanisms are designed to serve different goals for the Linux kernel system and received little attention for immediate improvement when NAPI was introduced. With either design, the asynchronous hardIRQ can lead to unpredictable performance hiccups. Chapter 5 describes more details on NAPI. Also, Chapter 5 further illustrates a microsecond-scale hardIRQ can lead to millisecond-scale latency.

Here, we expand the analysis to include three more applications running on four cores with different allocation strategies: MIX, where interrupt handling is bound to the same core as server threads, and SEP, where interrupt handling is bound to a dedicated core. Results are shown in Figure 3.5: applications like `silo` and `img-dnn`, with low variance service time, are more sensitive to interrupts; on the other hand, `xapian` and `moses`, with higher variance service time, are less sensitive to interrupts.

## 3.2 WORKLOAD GENERATION AND VALIDATION

TailWAG does not try to capture all the details of each application from Tailbench, including data movement, cache accesses, loop constructs, or other control flow. Instead, we use the timing distributions that are learned as discussed. Each target application run with the networked setup is treated as a black box, and only service time and total time are captured as metrics.

The generated workload is written as an individual application that occupies the CPU and exhibits the same characteristics as the real workload in terms of CDF (cumulative distribution function) of round trip service time, as shown in Listing 3.1. Each query

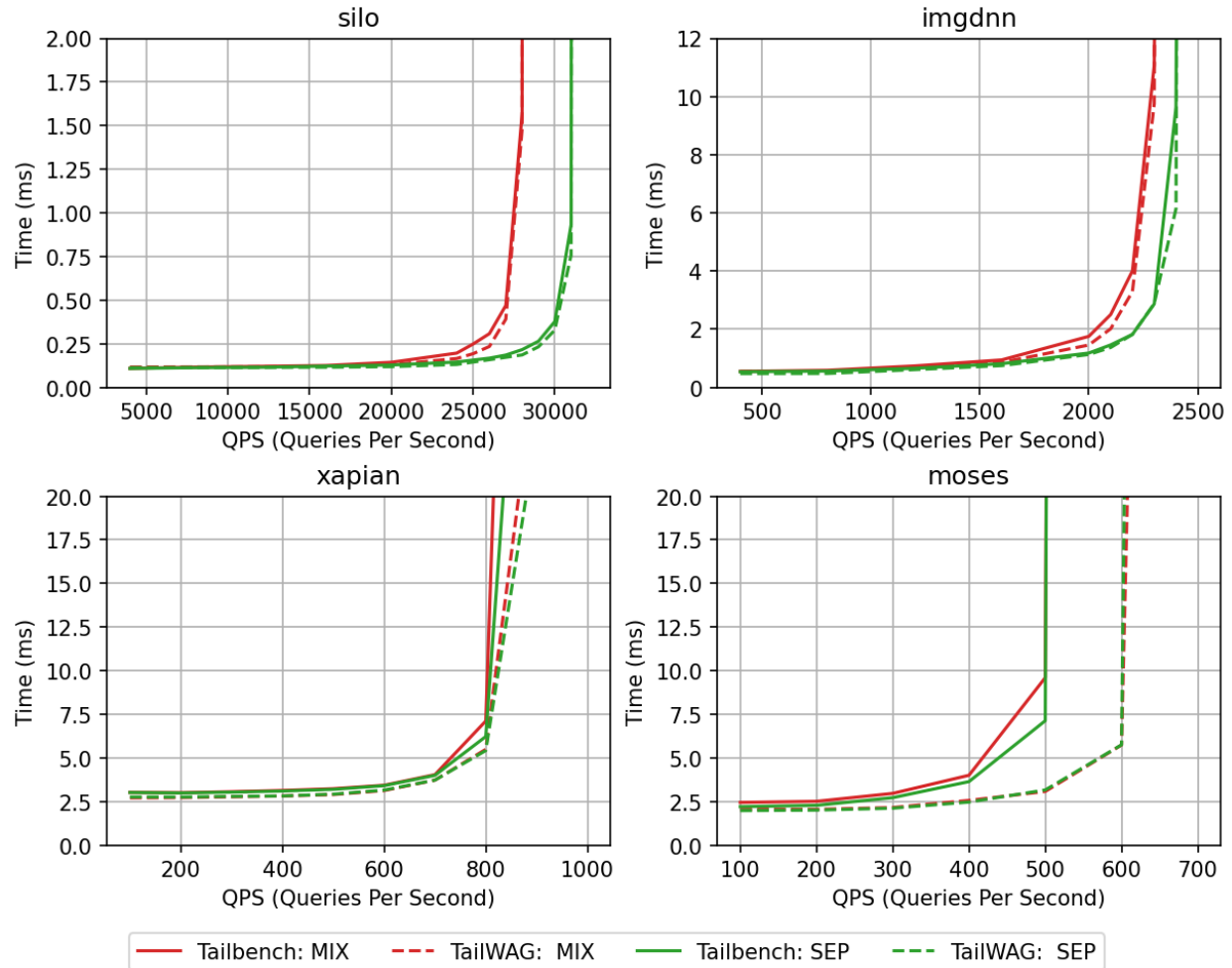


Figure 3.5: NIC disturbance (95th tail): Effect of IRQ Handling Noise on Tail Latency using TailWAG. TailWAG can capture effects from different applications, and also predict a memory bottleneck for mooses.

spins for cycles specified by different components, representing occupying CPU. It also acquires and releases a lock to represent a critical section.

Key parameters are:

1. Main execution distributions: These are the main parameters that determine the active service time for a request. We are using three distribution components derived from characterizing the applications during the study. These are represented as a vector of distributions with corresponding means, variances, and probabilities (summing to one). We found three are enough for these applications, but this can

be expanded to more if necessary. In addition, statistical functions from the Python Scipy package provide continuous probability function fitting (Virtanen et al. 2020), giving a data set's best estimation of mean and variance.

2. Threads: using multiple worker threads helps with system utilization, keeping fewer resources idle and increasing overall application throughput in the ideal case. However, more threads also increase the possibility of lock contention.
3. Critical execution time: There are shared resources between threads, corresponding to critical execution time, as we discovered (tcp send and recv time). After the non-critical section, every request will try to acquire the lock. A hybrid lock was added that will spin for a fixed time, then go to sleep. After successfully grabbing the lock, the thread will remain busy until it expends the critical section execution time.
4. Timing disturbance: a timing disturbance happens within the service section. There can be multiple timing disturbances. Small timing disturbance range from tens to hundreds of microseconds, emulating hardware interrupts. It can also be over milliseconds to seconds, emulating rare events such as I/O, runtime policies, and garbage collection.

Table 3.2: Parameters used for workload generation.

| distribution |               |                      | Critical        | critical           |                    |
|--------------|---------------|----------------------|-----------------|--------------------|--------------------|
| Application  |               | distribution 2       | distribution 3  | section 1          | section 2          |
| silo         | 16.14 $\mu$ s | 0.1%, 59.6 $\mu$ s   |                 | 3.5 $\mu$ s (0.15) | 3.5 $\mu$ s (0.15) |
|              | (0.288)       | (0.085)              |                 |                    |                    |
| specjbb      | 11.1 $\mu$ s  | 3%, 30.73 $\mu$ s    | 3%, 200 $\mu$ s | 3.9 $\mu$ s (0.1)  | 3.9 $\mu$ s (0.1)  |
|              | (0.39)        | (0.066)              | (0.049)         |                    |                    |
| masstree     | 105 $\mu$ s   | 0.001%, 1400 $\mu$ s |                 | 3.5 $\mu$ s (0.2)  | 3.5 $\mu$ s (0.2)  |
|              | (0.06)        | (0.05)               |                 |                    |                    |

| distribution |                         |                               | Critical                       | critical          |
|--------------|-------------------------|-------------------------------|--------------------------------|-------------------|
| Application  | distribution 2          | distribution 3                | section 1                      | section 2         |
| imgdnn       | 395 $\mu$ s<br>(0.007)  |                               | 6.4 $\mu$ s (0.5)              | 6.4 $\mu$ s (0.5) |
| xapian       | 532 $\mu$ s (0.3)       | 37.8%, 1800 $\mu$ s<br>(0.35) | 3.9 $\mu$ s (0.1)              | 3.9 $\mu$ s (0.1) |
| shore        | 92.8 $\mu$ s<br>(0.163) | 45%, 432 $\mu$ s<br>(0.24)    | 1900 $\mu$ s (0.124)           | 4.2 $\mu$ s (0.1) |
| Moses        | 840 $\mu$ s<br>(0.133)  | 85.7%, 1525 $\mu$ s<br>(0.14) | 0.05%, 11450 $\mu$ s<br>(0.54) | 4.9 $\mu$ s (0.1) |
| Sphinx       | 624.28 ms<br>(0.510)    |                               | 50 $\mu$ s (0.6)               | 50 $\mu$ s (0.6)  |

Listing 3.1: Generated Workload Pseudocode. Each loop represents a request sequential timing behavior.

```

1  void doRun() {
2  //setup parameters
3  ...
4  //each iteration
5  //is one query
6  while (true) {
7      recvRequest();
8
9      lock(recvLock);
10     spin(recvLockCycles);
11     unlock(recvLock);
12
13     p=uniform_random(0.001%,100%);
14     if p < prob1
15         distCycles=log_random(mean1,var1);
16     else if p < prob1+prob2
17         distCycles=log_random(mean2,var2);
18     else
19         distCycles=log_random(mean3,var3);

```

```

20         spin(distCycles);
21
22         //can be multiple noise injection
23         spin(noiseCycles);
24
25         lock(sentLock);
26         spin(sentLockCycles);
27         unlock(sentLock);
28
29         sendResponse();
30     }
31 }

```

---

With proper selection and tuning of parameters, TailWAG can mimic the latency distribution from real applications. Parameter values are listed in Table 3.2, and behavior for single-thread configurations is plotted against measured results in Figure with dash lines. Overall, TailWAG closely matches measured results for both latency (mean, 95th, 99th) and throughput, except specjbb 99th tail latency, caused by background threads that TailWAG does not model. Further validation results are shown in Figure 3.4 and Figure 3.5 (dashed vs. solid lines).

### 3.3 CASE STUDY

TailWAG can now represent a variety of applications using different sets of parameters, and the workflow is shown in Figure 3.2. Moreover, these parameters can be dependent or independent of user specifications. Workload generation by changing one of the parameters allows designers and programmers to explore potential changes without fully implementing them. Below, we showcase how to use TailWAG to explore: changes in service time due to algorithm complexity or hardware innovation for img-dnn, NIC IRQ handler acceleration for silo, and infrequent timing hiccups (garbage collection) for specjbb.

### 3.3.1 DNN evolution

Machine learning evolves rapidly, with algorithm, model, and hardware changes appearing regularly. **img-dnn** within this suite would be affected by such modifications. The baseline **img-dnn** runs on a Xeon CPU. TailWAG abstracts hardware details, replacing them with timing components (main distribution of mean 395  $\mu$ s) reflecting performance. Here we evaluate two changes to the model: Increasing complexity of the ML model for better accuracy and robustness, with more burden on computation; Google (Jouppi, Hyun Yoon, et al. 2021) reported there is an annual increase of 1.46X on FLOPs in CNN-based image recognition within Google. Second, hardware innovations that speed up the ML workload; Google (Jouppi, Young, et al. 2017) measured there is a 1.6x to 70x speedup (compared to CPU) when GPU or TPU are used for running CNN models. TailWAG can capture these trends: GPU/TPU innovation reduces the mean service time by 1.7x to 320  $\mu$ s or 70x to 7.75  $\mu$ s. Here, TailWAG only predicts the impact of this particular change, providing insight into other important behaviors (contention, tail latency distribution, etc.). We can use our model to predict performance benefits as in Figure 3.6. As performance increases, **img-dnn** moves into a realm where microsecond-scale timing disturbances are important, such as thread contention, lock implementation, and interrupt handling, similar to **silo** and **specjbb**.

### 3.3.2 NIC hardIRQ Accelerator

As previously discussed and shown in Chapter 1.1, reserving a core for interrupt handling (SEP vs. MIX) can improve tail latency at the cost of lowered throughput. Here we use TailWAG to examine the SEP and MIX cases and compare them to a hypothetical IDEAL case where interrupts do not perturb the server threads at all, as well as a hypothetical hardIRQ accelerator that services the interrupt with minimal overhead (around 0.5  $\mu$ s)

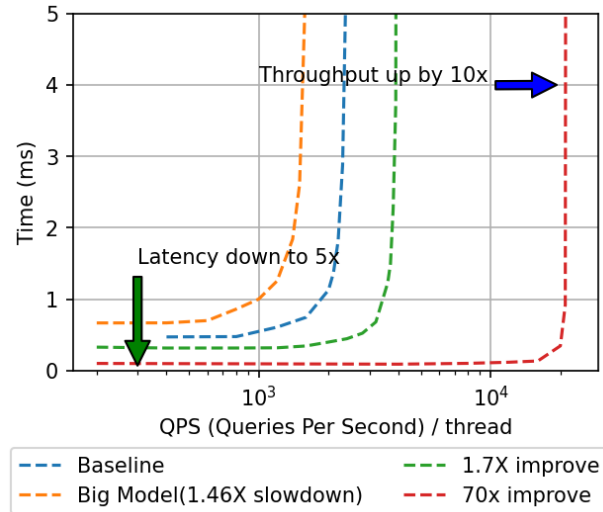


Figure 3.6: DNN evolution: Increased model complexity (orange), vectorization (green), hardware accelerator (red) affect mean latency.

using a tightly-coupled accelerator (TCA) attached to the CPU core. The TailWAG results in Figure 3.7 show that such an accelerator can bridge much of the performance gap to the optimal case while likely requiring much less hardware than a dedicated core. Detailed exploration of a hardIRQ accelerator is in Chapter 6.

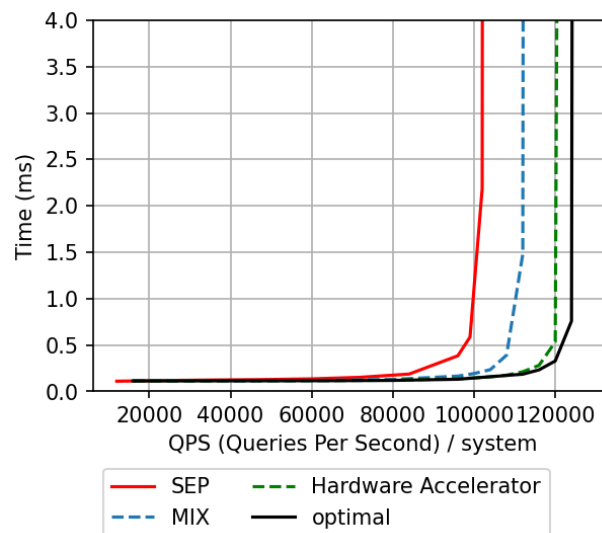


Figure 3.7: NIC IRQ: SEP/MIX with hardIRQ Accelerator affect 95th tail latency, with same application (silo). The optimal case needs an additional (5th) core.

### 3.3.3 Garbage Collection

Many applications are written in high-level programming languages that provide garbage collection, such as specjbb in our test suite. Garbage collection itself is a non-trivial problem: no single garbage collector is best for every application, and it should be chosen with trade-offs in mind (Yang and Wrigstad 2022). Several garbage collectors are used in Java; here, we picked plgc (parallel, throughput-oriented) and zgc (latency-oriented) to showcase TailWAG. Here, single thread specjbb is modeled in Figure 3.8: plgc case has better service latency but worse “stop the world” events, around 50 ms every 10 seconds; zgc has worse mean service time (around 10%), since garbage collection runs concurrently, but has better “stop the world” events, less than 1ms each, leading to better tail latency distribution.

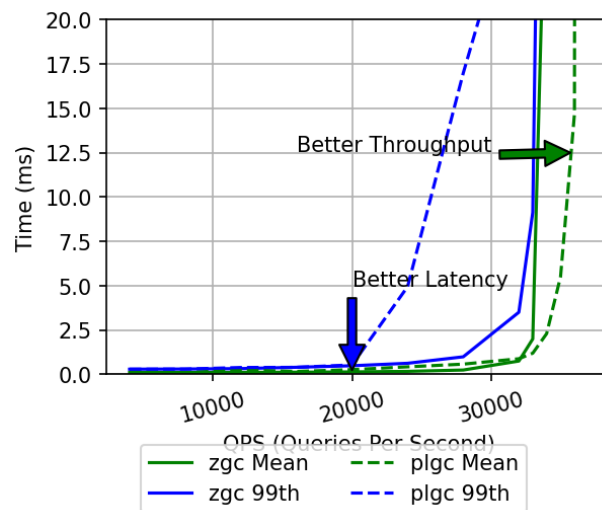


Figure 3.8: Java GC: Parallel (plgc, throughput oriented) and Zero (zgc, latency oriented) affect mean and 99th tail latency.

### 3.4 SUMMARY

Tail latency is a crucial performance metric for server workloads. Benchmarks exist but suffer from limitations and require significant effort to run as they suffer from run-to-run variation and experimental noise. In this chapter, we propose TailWAG, a workflow that analyzes tail latency-sensitive workloads, extracts parameters that describe timing behavior; then generates a synthetic workload that closely mimics real application performance, both single load cumulative distribution and trends under different loads. TailWAG performs well overall, with an average of 6.2% errors on mean, 1.9% on the 95 percentile tail, and 7.3% on the 99 percentile tail latency, respectively. Moreover, using the generated workload and parameter sweeping, we can explore the design space of hardware/software innovations and assess runtime configuration impacts with minimal implementation effort. Furthermore, the TailWAG generation version of the silo is used for our study in a native machine in Chapter 4 and Chapter 5. Then, it is also used in the evaluation of the proposed accelerator in Chapter 8.

## 4 NAPI AND THREADED NAPI

In the previous chapter, we described the TailWAG workflow that analyzes Tailbench(Kasture and Sanchez 2016) and then generate microbenchmarks allowing early-stage design exploration. One of the case studies we applied is investigating the possibility of removing timing disturbance of handling interrupts from NIC and implied potential hardware accelerator solution. Therefore, this chapter further explores interrupt handling.

Interrupt handling (Gleixner and Molnar 2010; *Unreliable Guide To Hacking The Linux Kernel — The Linux Kernel Documentation* 2023) is a vital core-API layer and is commonly defined as synchronous (exception, software) and asynchronous (timer, driver hardware)(Bovet and Cesati 2006). NIC (Network-in-Chip) interrupt interface is called NAPI(LinuxFoundation 2016; Salim, Olsson, and Kuznetsov n.d.). NAPI was introduced into the Linux kernel 2.6 in 2003. NAPI is a mixture of interrupts and polling, which the CPU alternates between. First, the CPU uses interrupts to signal to receive and acknowledge a packet, then uses polling to move additional packets from NIC to memory. This polling happens inside of the software interrupt (softirq).

There have been incremental improvements in NAPI throughout the years, but the mixture model always stays the same. Recently in 2020, the Linux kernel version 5.12 (Kernelnewbies 2023) included the newest implementation that changes this. It moves polling from softirq to a dedicated kernel thread (*Linux: Starting and Stopping a Kernel Thread* 2023).

To the best of our knowledge, this feature had only documented performance analysis within the mail exchange when changes were proposed to push into mainline (Corbet 2023; W. Wang 2023). The results included in the mail are tcp\_rr tests, including latency

and throughput on sets of 3 different message sizes. It showed moderate improvement with limited implications.

Thus, in this Chapter, we will dive into the details of the NAPI model, conduct experiments explore the advantages of threaded NAPI. This chapter structures as follows:

1. Chapter 4.1 covers different contexts within the Linux kernel.
2. Chapter 4.2 breaks down NAPI in details.
3. Chapter 4.3 introduces the new threaded NAPI.
4. Chapter 4.4 does performance analysis of threaded NAPI and implications.

## 4.1 TERMINOLOGY

### 4.1.1 CPU contexts

At any given time, the CPU is in one of the four states: user space, user context, softirq, and hardirq.

Any user program running process is maintained in user space or associated library, such as malloc and memcpy in the C library.

A user program switches to kernel space when it makes a system call. The process has higher privileges in kernel mode and can perform tasks such as POSIX socket message access using recv and send. Still, it has a user process corresponding to it, so it is labeled user context.

When hardware devices need CPU attention, it uses interrupts. When the system has multiple hardware devices and the service routine becomes longer and more cumbersome, it is broken into a top half and a bottom half. Top-half performs as a hand-shake mechanism. It must be minimal since the CPU is now in the highest privilege level, and no other task can interrupt it, even from other devices. Bottom-half performs the heavy

work, such as moving data, checking, and processing. Now, the more common names for them are hardirq and softirq.

The two lower-level (closer to hardware) states have higher priority. Hardware interrupts can always preempt software, user space applications, and system calls in kernel space for user context. Software interrupts can always preempt user space applications and user context.

The two high-level states are loosely in one preemption level. In other words, the process within the user space application and user context can preempt each other.



Figure 4.1: Lower level context (closer to hardware) has higher priority. Userspace and usercontext can preempt each other.

#### 4.1.2 Other hardware interrupts

A few interrupts can happen simultaneously, including but not limited to Timer, RES (Rescheduling Interrupts), and CAL (Call Function Interrupts). Timer interrupts are straightforward and well-known, unlike RES and CAL, needing more documentation and usage guidance. Both RES and CAL are delivered using IPI (inter-process interrupts).

RES happens when one needs to schedule another task on a remote core with a higher priority than the current task.

CAL happens when a thread can wake up, but the current core is busy, and a remote core is available. In this case, it will try to check and schedule itself on the remote core through CAL. This happens when having more threads running than the cores. There will be constant changes in running cores for the individual threads.

## 4.2 NAPI

The CPU moves between these different states to accomplish tasks running user applications and communicating with the hardware.

NAPI was invented for better networking packet handling for scaling up systems, including SMP (Symmetrical MultiProcessing) and Gigabit Ethernet.

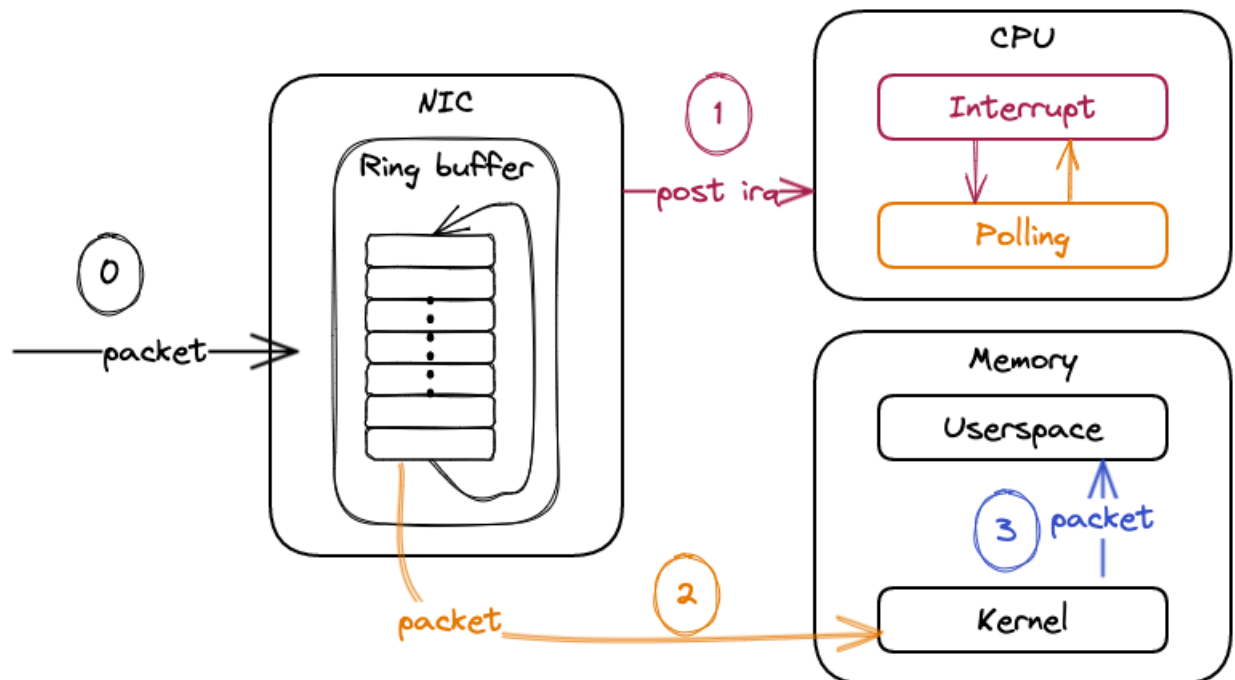


Figure 4.2: NIC interrupt interface NAPI is mixture of interrupt and polling.

When a packet receives through Ethernet, there are multiple steps until it is available for the user. The steps illustrated in 4.2, and as below:

- o. Packet arrives at NIC
- 1. NIC post interrupt to CPU, CPU acknowledge and switch to polling mode.
- 2. Polling function moves packets from NIC to the main memory.
- 3. user uses syscall (POSIX recv) to move packets from the kernel to the user space.

Step 3 by itself is complicated, and there are many different ways for users to consume the packet, such as TCP stack, UDP stack, etc. This part is beyond our scope of focus. Instead, we focus on interrupts and polling packets to kernel space.

#### 4.2.1 *Network Interface Controller*

NIC(Network Interface Controller) is the primary device receiving and sending packets to the outside world through an Ethernet port. NIC itself has its processing unit and memory. The main memory structures for packets are ring buffers. Some modern NICs can provide multiple queues with separate ring buffers for transmission and reception. These ring buffers have a head pointer and a tail pointer. These pointers indicate the range for buffers that contain packets that need to be transmitted through Ethernet or packets received from Ethernet and waiting for the CPU to do the polling.

Two interrupts correspond to Ethernet devices, RX(receiving) and TX(transmission). On the RX side, whenever any packet arrives, the Ethernet device will signal an interrupt through PCI communication to the CPU (similarly for the TX side). With TCP-based applications, a single request can generate 2 pairs of interrupts(2 RX and 2 TX), one for receiving requests and sending the acknowledgment and another for sending the response and receiving an acknowledgment. This number can increase if the message cannot fit within MTU(maximum transmission unit) and must be segmented.

#### 4.2.2 *Hardware interrupt*

The target CPU must stop the current task and respond when an interrupt signal is posted onto a specific CPU through the interrupt controller. This routine is hardware interrupt, noted as hardirq.

The generic routine for interrupt handling is as follows:

1. Jump to a specific interrupt handling routine entry point based on the current context and vector table.
2. Check and save the current program stack.
3. Switch to hardirq context.
4. Check the interrupt vector table and use the interrupt number to find the corresponding interrupt handler (irq\_handler).
5. Acknowledge interrupts from NIC and mask future interrupts on the specific queue.
6. Switch to polling mode, and raise softirq pending signal NET\_RX\_SOFTIRQ.
7. Return from this hardware context.

#### 4.2.3 *Polling in softirq*

As part of the generic interrupt handling layer, before the hardirq context exits the current routine, it calls irq\_exit. As it shows in the listing, the CPU always tries to invoke\_softirq if a softirq routine awaits. In this case, it is NET\_RX\_SOFTIRQ, and the CPU switches to the softirq context.

---

```

1 // /linux/kernel/softirq.c
2 void irq_exit(void) {
3 ...
4     if (!in_interrupt() && local_softirq_pending())
5         invoke_softirq();

```

---

```
6 }
```

Similar to `hardirq`, there are several steps for the software interrupt routine:

---

```
1 // Pseudocode of polling in softirq
2 void softirq(void) {
3 ...
4     if (ksoftirqd.running) {
5         // let ksoftirqd process
6         return;
7     }
8     acknowledge(softirq);
9     // get net_rx_action
10    action = vector_table.find(softirq);
11
12    while (time < limit) {
13        action.run();
14    }
15    if !work_finished() {
16        wake_up(ksoftirq);
17    }
18 }
```

---

The **limit** was a mechanism balancing latency and fairness for polling the packets. The limit sets how long polling runs within `softirq` and is specified in different places in different ways.

In Generic `softirq`:

---

```
1 // within __do_softirq
2 #define MAX_SOFTIRQ_TIME msecs_to_jiffies(2)
3 #define MAX_SOFTIRQ_RESTART 10
4 if (pending) {
5     end = jiffies + MAX_SOFTIRQ_TIME;
6     ....
7 restart
8     net_rx_action();
9     if (softirq_pending) {
10         if (time_before(jiffies, end) && !need_resched() && --max_restart)
```

```

11         goto restart;
12         // limit reached
13         wakeup_softirqd();
14     }
15 }

```

---

In specific NIC polling function:

---

```

1 // within net_rx_action
2 /* Must be at least 2 jiffies to guarantee 1 jiffy timeout */
3 void net_rx_action(){
4     unsigned int netdev_budget_usecs = 2 * USEC_PER_SEC / HZ;
5     int netdev_budget = 300;
6     int weight = 64;
7     unsigned long time_limit = jiffies +
8         usecs_to_jiffies(netdev_budget_usecs);
9
10    for (;;) {
11        struct napi_struct *n;
12        budget -= napi_poll(napi_struct, &repoll);
13        /// If softirq window is exhausted, then punt.
14        if (unlikely(budget <= 0 ||
15            time_after_eq(jiffies, time_limit))) {
16            sd->time_squeeze++;
17            break;
18        }
19    }
20 }

```

---

As one can imagine, the combination of these makes polling in softirq context behavior challenging to control and sometimes unpredictable.

#### 4.2.4 Polling in ksoftirqd

Finally, the ksoftirqd is woken up. Ksoftirqd runs until all packets are polled into the kernel stack. Ultimately, the CPU clears an interrupt mask on NIC, allowing future interrupts to be posted. CPU now switches back to interrupt mode.

As shown in Figure 4.3, we summarize the steps in time.

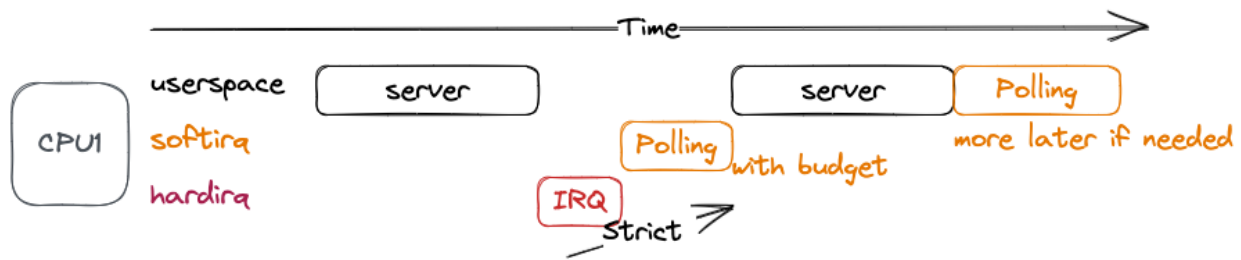


Figure 4.3: NAPI uses softirq for polling

NAPI continues the philosophy with generic interrupt handling using the hardirq/-softirq. The CPU spends a minimum amount of time in hardirq (hardware interrupts), eliminating heavy work in the highest priority and disabling further interrupts from the same NIC; on the return path, the kernel picks up softirq and polls the NIC since it is likely the packet will be needed in user space. But, a budget is given, and it ensures the user space application is not waiting too long.

A mixture of interrupt and polling provides overall efficient throughput and acceptable response latency. The polling in the softirq with limits was fair 20 years ago. However, with high-speed Ethernet NIC and 4GHz core with modern-day architecture, this time spent in kernel asynchronously leads to unstable performance. It causes effort to push this away from softirq within the kernel community. The asynchronous hardirq can now lead to unpredictable performance hiccups, especially for tail latency.

### 4.3 THREADED NAPI

One of the earliest ideas of Threaded NAPI was in 2009 (*Threaded Network Device Interrupts / Linux Plumbers Conf: Developing the Kernel, Libraries and Utilities / September 23-25, 09 / Portland, OR 2023*), but not introduced in the Linux kernel until version 5.12 (Kernel-newbies 2023) on April 2021. Some noticeable distributions include this feature: Ubuntu (*Ubuntu Kernel Lifecycle and Enablement Stack 2023*) 20.04.4 on February 2022 or newer;

Fedora 35 on November 2021 or newer; Red Hat Enterprise Linux 9.0 on May 2022 or newer.

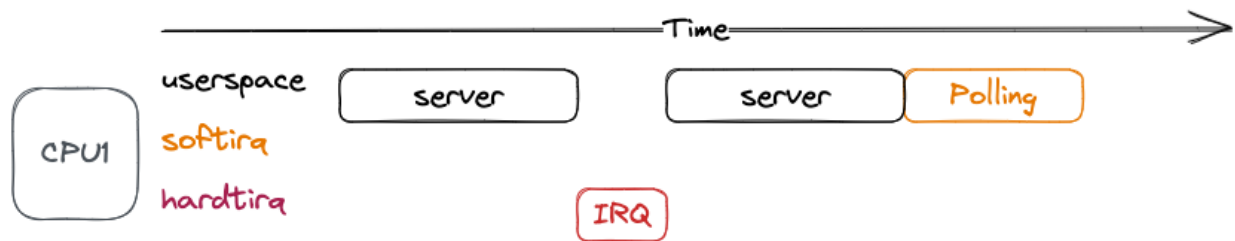


Figure 4.4: threaded NAPI uses dedicated kernel thread for polling

As shown in Figure 4.4, the threaded NAPI still uses the same polling function for moving packets from NIC to the main memory. However, threaded NAPI changes where the polling is running, from softirq + ksoftirqd to a NAPI thread.

Kernel threads are similar to user threads but can only be created for kernel tasks and are maintained in kernel space (*Linux: Starting and Stopping a Kernel Thread 2023*). Ksoftirqd is also a kernel thread per CPU and cannot be moved to a different core. Meanwhile, the NAPI thread is per-queue and visible to the scheduler.

---

```

1 // Linux kernel v5.14
2 // linux/net/core/dev.c
3 static inline void ____napi_schedule(struct softnet_data *sd, struct napi_struct
4
5     // new, threaded NAPI
6     struct task_struct *thread;
7     if (test_bit(NAPI_STATE_THREADED, &napi->state)) {
8         thread = READ_ONCE(napi->thread);
9         if (thread) {
10             if (READ_ONCE(thread->__state) != TASK_INTERRUPTIBLE)
11                 set_bit(NAPI_STATE_SCHED_THREADED, &napi->state);
12                 wake_up_process(thread);
13                 return;
14             }
15         }
16
17     // classic, softirq NAPI
18     list_add_tail(&napi->poll_list, &sd->poll_list);
19     __raise_softirq_irqoff(NET_RX_SOFTIRQ);

```

Above in the listing, we show how the path diverges between threaded NAPI and NAPI. As we can see, threaded NAPI does not raise softirq and allows the CPU to return directly from the hardware interrupt context (hardirq) to user space.

However, scheduling a task in user space by calling `wake_up_process` is not free.

The Linux kernel provides different schedulers. Common ones are CFS (Completely Fair Scheduler), RR (Round-Robin), and FIFO (First come, first served). The penalty is related to a current system running status. This latency also exists in the classic NAPI model when `ksoftirqd` is scheduled. We will dive into more details about scheduling in Chapter 6.2.

Table 4.1: NAPI has a smaller hardirq region but a longer softirq and is also unpredictable. On the other hand, tNAPI has no softirq but longer hardirq.

| Mode  | hardirq                   | softirq              |
|-------|---------------------------|----------------------|
| NAPI  |                           | +polling (1 $\mu$ s) |
| tNAPI | +scheduling (0.5 $\mu$ s) |                      |

Table 4.1 compares the trade-off between time spent within the asynchronous region (hardirq + softirq). Traces show that scheduling is around 0.5  $\mu$ s on average; polling is 1  $\mu$ s on average but can go as high as 10  $\mu$ s. Even though tNAPI spends more time within the hardirq, it exits to user space without touching softirq.

Note the threaded NAPI is not a replacement for NAPI, at least not in the current stage of Linux development. Instead, the threaded NAPI is an additional feature that users can choose to use, and it is not enabled by default.

#### 4.4 PERFORMANCE ANALYSIS

Using tNAPI increases the region where interrupts are disabled since the logic now needs to access the scheduler, covers the wakeup of a separate thread, and puts them on the run queue. However, the whole asynchronous part was reduced. Note the time in softirq is not fixed. Here, kernel configuration is set to be a maximum of 2  $\mu$ s. When there is little work to do, such as one single ping request, it will be less. Thus, when the system is under a very low load, using tNAPI can have a negative impact since it delays the packet process time from softirq to a later kernel thread.

On the other hand, when the system is at medium to high load, tNAPI helps the CPU focus on the user application, reducing the time spent in interrupts. Most importantly, NAPI polling performance is unpredictable. It can be performed during exiting hardware context or on ksoftirqd, the thread shared by all potential interrupt routines on a core.

As a dedicated thread for NIC interrupts, Threaded NAPI can have altered priority, be pinned to any core, and is easy to monitor from the kernel maintainer's view, giving predictable interaction with user space applications and performance. In the following subsections, we examine the performance on:

1. Single thread on a single core, in Figure 4.5.
2. Scheduling Priority difference, in Figure 4.6.
3. Multiple threads on multiple cores, in Figure 4.7.

Table 4.2: Server System Configuration.

| Model       | Supermicro SYS-2029GP-TR            |
|-------------|-------------------------------------|
| Cores       | Cores Intel Xeon Gold 6128, 3.5 GHz |
| Caches      | 32KB L1, 6MB L2, 19.25MB L3         |
| Main Memory | 96GB, 1333 MHz                      |

|                  |                                   |
|------------------|-----------------------------------|
| Model            | Supermicro SYS-2029GP-TR          |
| Operating System | Ubuntu 22.04, Linux kernel 5.15.0 |

This experiment is performed on a native server-scale machine, configurations are listed in Table 4.2. Hyper-threading is disabled in our experiment, and only one node (6 cores) is used. For all comparisons, both SEP and NAPI use classic NAPI. In addition, SEP uses an additional core for interrupt handling, representing a standard solution for interrupt isolation.

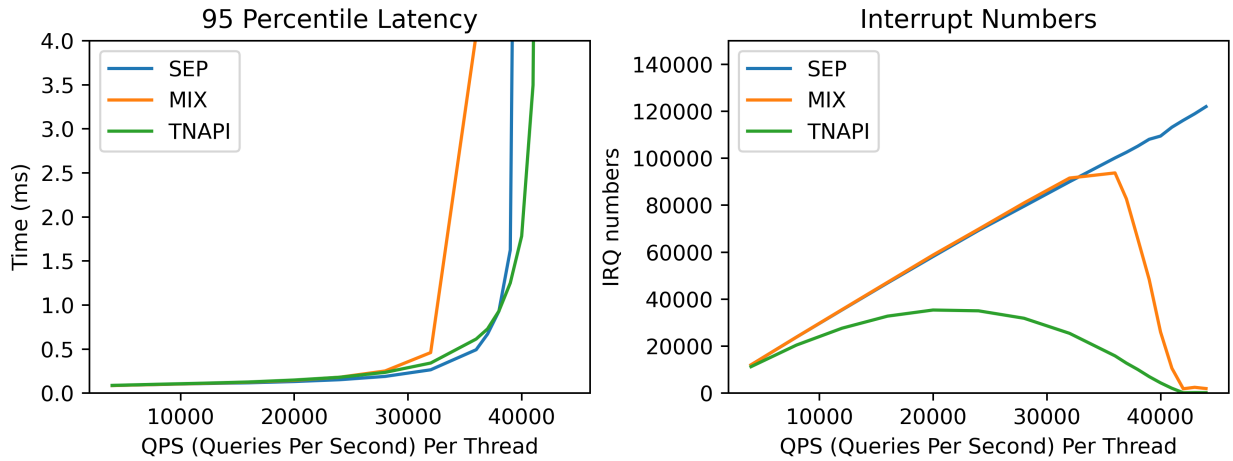


Figure 4.5: Using threaded NAPI helps performance, and improves throughput and latency at higher loads. Even more, it can achieve better performance than SEP, which uses an additional core, shown on the left. On the right, it shows the interrupts number saturated and decreases when polling cannot handle packets fast enough.

First, Figure 4.5 shows the baseline comparison with a single thread. The interrupt numbers go down after a certain point since the system is saturated with the packets, and polling is not responding fast enough to switch back to interrupts. One of the possible causes for threaded NAPI performing better than SEP is data movement between cores.

The whole purpose of using threaded NAPI is for better scheduling. As shown in Figure 4.6, tNAPI only works if it has no higher priority than the server application. Using CFS intrinsically makes it a higher priority since the NAPI thread does not consume much time. Whenever the NAPI thread is ready to run, CFS checks how much it has

consumed the CPU time. NAPI thread has lower CPU time consumption than server threads over time. As a result, the NAPI thread can always take over the CPU.

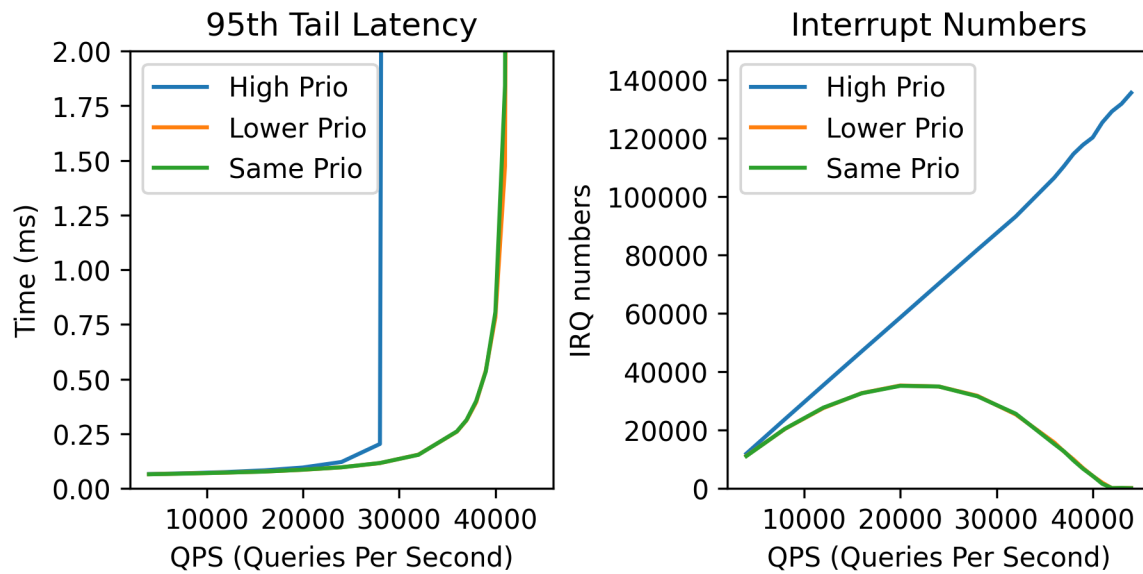


Figure 4.6: Scheduling policy matters: NAPI thread has higher priority and is equally bad polling as softirq since it preempts server threads. Using Round-Robin and enforcing the same or lower priority is preferred. On the right, it shows the interrupts number does not saturate for high-priority threaded NAPI.

Furthermore, now we scale up from a single thread to 4 threads. We have the server threads running on cores 0-3, and SEP uses an additional 5th core for interrupt handling only. For NAPI and tNAPI, we use a single queue and bind interrupt and the NAPI thread to core 0.

More exciting cases exist for multiple server threads that can move between cores, shown in Figure 4.8. Lower priority guarantees the server workload runs before NAPI, so it responds to polling packets slower. On the other hand, higher priority makes it respond faster. Thus, lower priority has a lower saturation point on interrupt numbers on the right.

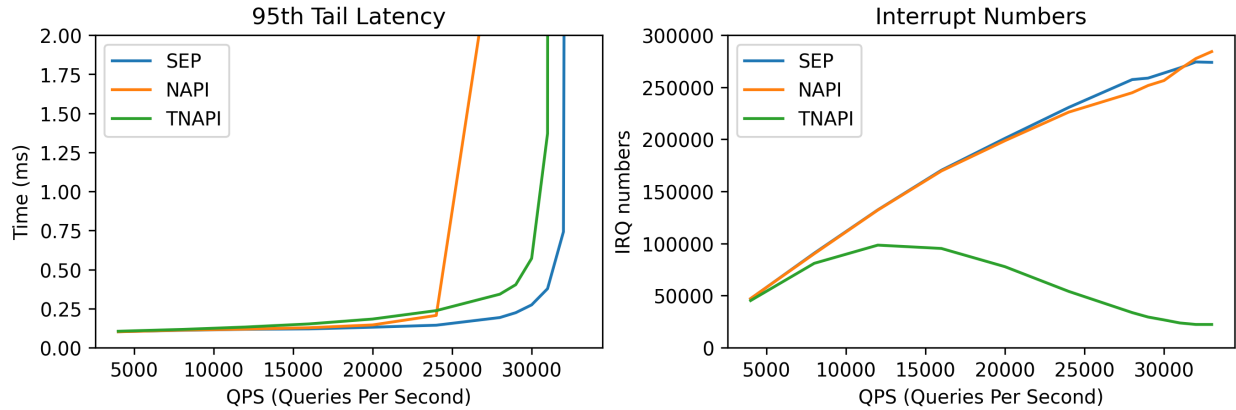


Figure 4.7: 4 threads workload running. tNAPI performs better than NAPI without surprise. SEP performs better than TNAPI in multiple threads cases. On the right, it shows the interrupts number does not saturate for NAPI, since multiple threads on multiple cores push server application to be run cores 1-3, leave core 0 for NAPI.

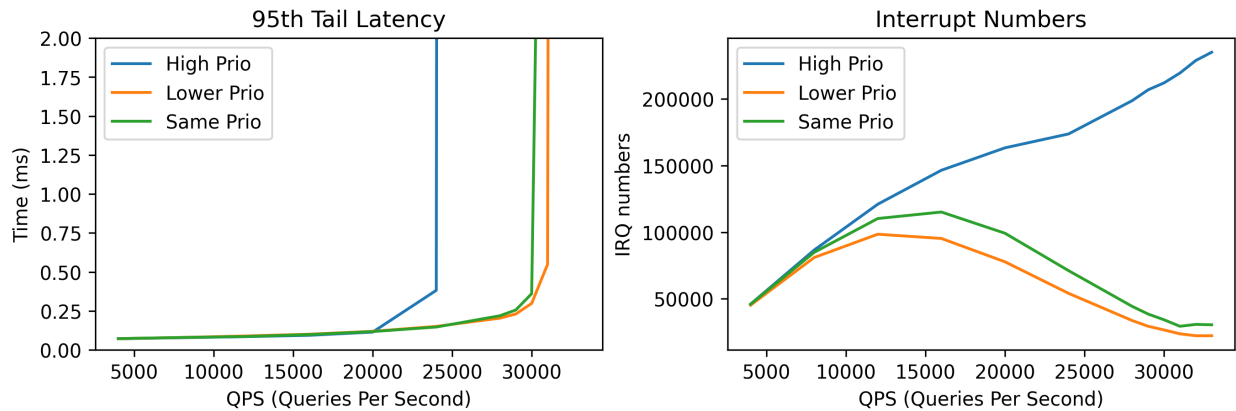


Figure 4.8: Scheduling policy matters: NAPI thread with higher priority is equally bad as using CFS. Using Round-Robin same or lower priority is preferred. Interrupt numbers on the right shows it saturates sooner when NAPI thread has lower priority.

#### 4.5 SUMMARY

NAPI is a mixture model of interrupt and polling. The interrupt part of NAPI turns off other interrupts completely and does a minimum amount of work on switching to polling. The polling happens within softirq with a limited budget. If more polling is required, the per CPU kernel thread (`ksoftirqd`) is woken up.

The threaded NAPI does not use `softirq`. Instead, it wakes up per queue kernel thread immediately. The polling is all performed in the kernel thread and can be controlled by the kernel scheduler. A significant amount reduces the asynchronous interrupt time, and polling can be more controllable.

We propose to reduce it further by offloading interrupts to an accelerator and removing the asynchronous disturbance from the CPU, from  $1\ \mu\text{s}$  to  $0\ \mu\text{s}$ . Also, why does the interaction with the kernel scheduler double the latency within the interrupt? We will look into it in Chapter [6.2](#).

## 5 CASCADING-EFFECT

In the last Chapter, we described the NIC interrupt handling routine in the modern computer system, NAPI, and the more recent development on the threaded NAPI. This timing disturbance from handling asynchronous hardIRQ is the  $\mu\text{s}$  scale. This chapter will explain how that  $\mu\text{s}$  timing disturbance leads to ms scale latency. This Chapter follows the organization as below:

1. Chapter 5.1 shows a step-by-step walkthrough on how the  $\mu\text{s}$  timing disturbance can lead to unexpected ms latency for a server workload.
2. Chapter 5.2 provides a quantitative analysis of how much of an impact the  $\mu\text{s}$  timing disturbance can have.
3. Chapter 5.3 made improvement implications from the quantitative analysis.

### 5.1 IRQ MEETS SYNCHRONIZATION

In this section, we started with a simple single-thread application with analogies used. Next, we show the lock contention origin with the race condition. Then, we described both spinning and sleeping for synchronization. Furthermore, we show that a mixture of hybrid spin locks makes sense for real applications. Finally, we show how  $\mu\text{s}$  IRQ and the critical section interact.

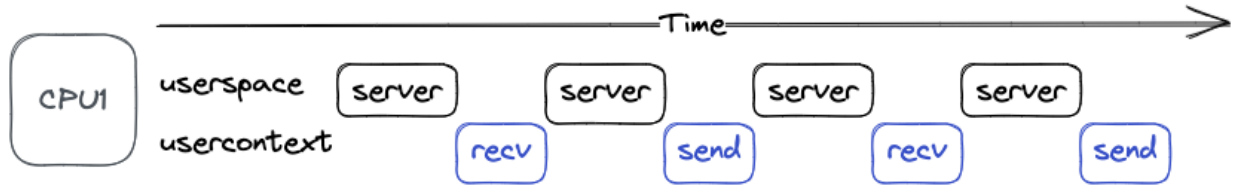


Figure 5.1: Ideal case with single thread.

### 5.1.1 Base server activities

Figure 5.1 shows an ideal simplified CPU utilization when running a server application at 100% load. The server is running in userspace, and each request will take a certain amount of time, but in between services, the server needs to **recv** a new request and **send** a response back. POSIX ( Portable Operating System Interface) is the most widely used standard for compatibility between components, such as socket and multithreading programming. Here, the server uses POSIX **recv** and **send** as below:

---

```

1 ssize_t recv(int socket, void *buffer, size_t length, int flags);
2 ssize_t send(int socket, const void *buffer, size_t length, int flags);

```

---

In a UNIX system, every socket is treated as a file, and server and client communication through socket programming is read and written from the same file. The detailed implementation of syscalls and their implications are beyond the scope of this thesis. In our study, the data field in the message(request and response) between the server and client has a minimum size of 4 bytes. Note that the message contains other information, such as id and time spent on different parts of the stages defined on the user application. After each layer of abstraction on the TCP/IP stack, the **packet** contains more information on IP, ports, checksum, etc. After all, we measured the average send time for the minimum response at around 3  $\mu$ s.

In our study, we focus on the send for isolation. The same analysis can be applied to recv lock as well. However, failure on recv lock can be caused by no available packet (system under low load), and the time spent waiting for receiving lock does not count towards the round trip time from client to server and back to the client. In other words, only the send is on the critical path.

### 5.1.2 Threads race conditions

The server needs multiple threads serving multiple clients in an open-loop design for scalable throughput. Therefore, n-M pairing is expected, with N being the number of server threads and M being the number of client connections. Each connection is still a socket connection. From the server side, an array of file descriptors must be checked round-robin for incoming requests. Different threads access the same file descriptors for a server with multiple threads, which can easily cause race conditions, as shown in Figure 5.2.

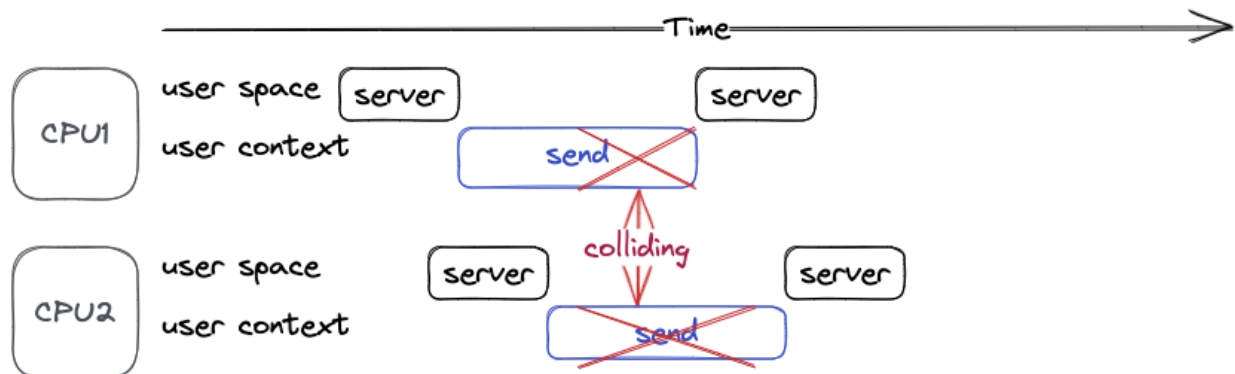


Figure 5.2: Race condition in multiple threads application

### 5.1.3 Two solutions: sleeping VS spinning

It is common to have barriers guarding shared resources for synchronization in multi-threaded programming. Two major synchronization mechanisms in the Linux kernel can achieve this: sleeping and spinning locks (*Lock Types and Their Rules — The Linux Kernel Documentation 2023*).

Spinning lock: as suggested by the name, when the process fails to acquire the lock, it will occupy the CPU and spin until it succeeds. It could be dangerous to use spinning locks, such that functions such as `raw_spinlock_t` and `spinlock_t` are mainly used inside the Linux kernel implementation.

On the contrary, sleeping locks are less dangerous. If the process fails to acquire the sleeping lock, it will preempt itself and goes to sleep. The process will be awakened when another process releases the lock. Mutex is a widely used sleeping lock, and the Linux kernel has a Generic Mutex Subsystem (*Generic Mutex Subsystem — The Linux Kernel Documentation 2023*).

Modern programming practice relies on the standard library and built-in function since they are more likely to be optimized and maintained for a specific purpose. It is common to use POSIX `pthread_mutex`, which uses a generic mutex subsystem underneath, and the syntax is shown below:

---

```
1 int pthread_mutex_lock(pthread_mutex_t *mutex);  
2 int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

---

In addition to the sleeping lock style function call, it provides the `trylock` function call. The `trylock` function call checks if the lock can be acquired. If so, it will return the lock obtained; otherwise, it returns `fail`. Usually, the `trylock` will be used when the user can take alternative actions on the failure to acquire the lock. One usage is to keep on calling

trylock until success. This achieves a spinning lock in user space easily. But, it should be used cautiously.

---

```
1 int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

---

Let us see the implications of using these two different mechanisms in Figure 5.3 and Figure 5.4. Figure 5.3 shows the interaction when a sleeping lock is used. Thread in CPU2 preempts itself immediately when acquiring the lock is failed. On the other hand, Figure 5.4, utilizing a spinning lock, keeps the server thread on CPU2, and provides a fast response when the server thread on CPU1 releases the lock.

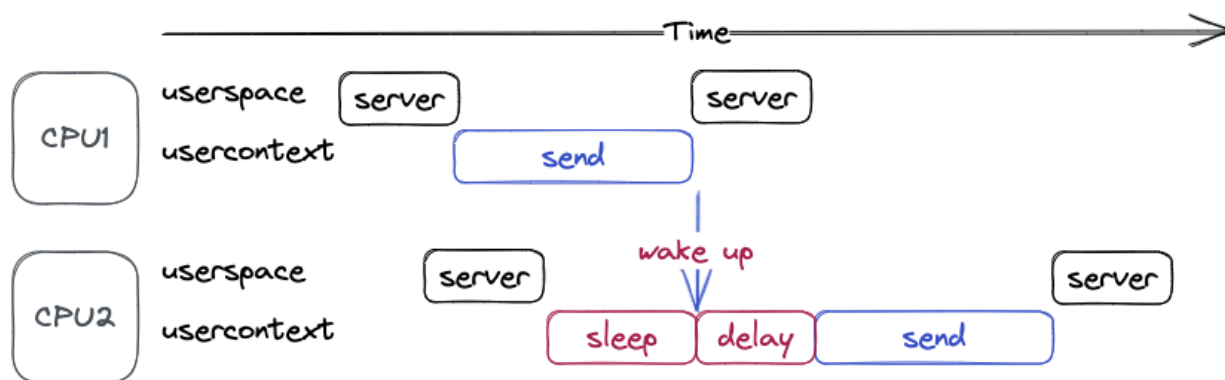


Figure 5.3: Sleeping Locks: additional cost on waking up

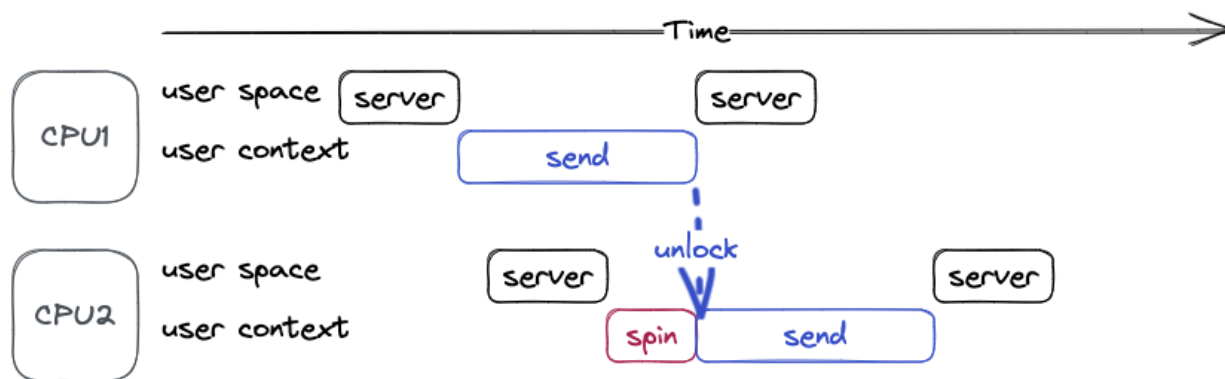


Figure 5.4: Spinning locks: CPU cannot serve other task

The spinning lock will be responsive since the sleeping lock must pay for the delay in waking up. From the comparison, why would one not want to spin at all? No one wants to pay for the delay penalty on scheduling waking up. This is true in an ideal world, with more resources than needed and no CPU sharing required.

- When does one not want to spin forever? When there is other valuable work to do.
  1. other user threads, there are more server threads than cores.
  2. background task: networking, such as tNAPI and threaded NAPI, that do polling for packet transactions.

#### 5.1.4 Hybrid lock as a solution

The hybrid lock will spin for a specific time, then give up if the spin limit is hit and sleeping. It is a middle ground between the pure spinning and pure sleeping locks. Within the newer Linux kernel mutex implementation, there is an addition of `PTHREAD_MUTEX_ADAPTIVE_NP`, which spins for a period before sleep, in case hardware can access the lock fast enough. Still, it is beyond the user's reach and is not used explicitly. At the user application level, `pthread_mutex_trylock` will be the candidate for hybrid lock usage as below:

---

```

1  static uint8_t hybridLock(pthread_mutex_t *mutex, uint64_t spinMax) {
2      uint64_t spin = 0;
3      // Try spinning until we hit the spinMax
4      while (spin != spinMax) {
5          if (pthread_mutex_trylock(theLock) == 0) {
6              return (uint8_t) 1;
7          }
8          spin++;
9      }
10     // give up, we should use the sleeping lock
11     pthread_mutex_lock(mutex);
12     return (uint8_t) 0;

```

The careful tuning of the spin count limit is not trivial since it can depend on the following:

- What is the ratio between critical and non-critical sections?
- How many threads are accessing the lock?
- What else is sharing hardware resources?

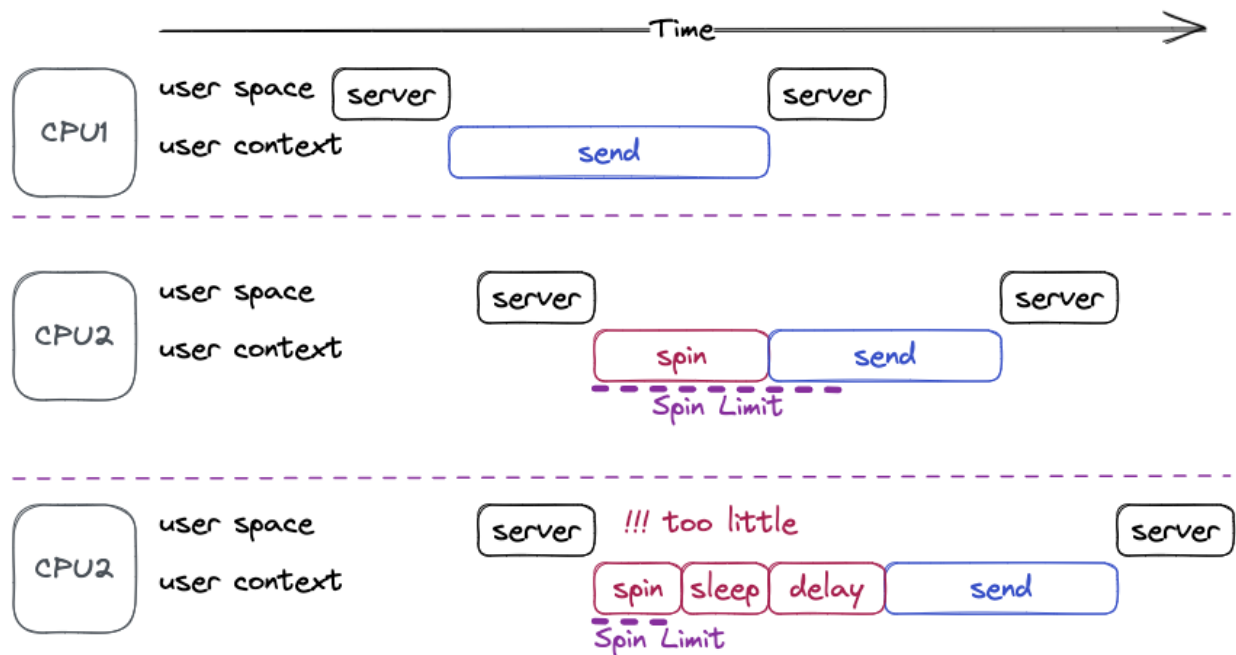


Figure 5.5: Hybrid Lock: spin count is important

Figure 5.5 illustrates the usage of hybrid lock. In one case, the spin limit is long enough; in the second case, the spin is too short, giving up the CPU anyway and wasting the spin time that could be used for other processing.

### 5.1.5 Timing Disturbance

Asynchronous timing disturbances from the NIC IRQ are not a big problem at first glance, consuming around 2-5  $\mu$ s. However, this microsecond delay is from hardware interrupts with the highest priority and can happen anytime.

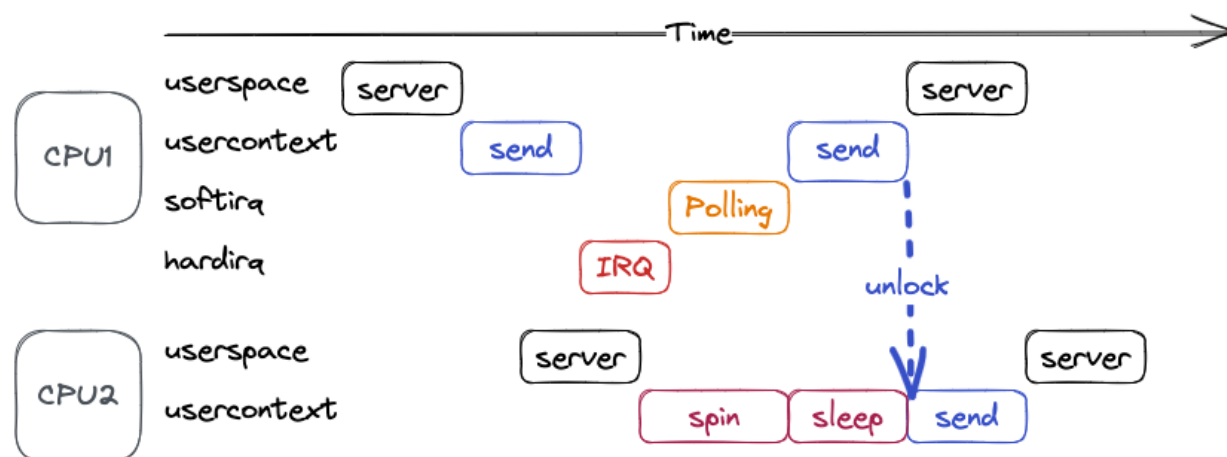


Figure 5.6: Asynchronous Timing Disturbance can cause trouble: may need longer spin count.;

Above in Figure 5.6, thread2 could have gotten the lock without sleep if interrupts were not happening. In this example, thread2 was not delayed much since nothing was running on CPU2. However, unlike in Figure 5.7, “another task” already takes over. Therefore, thread 2 must wait until the “another task” finishes or the scheduling time quantum is up.

Threaded NAPI is a newer kernel feature introduced in the Linux kernel 5.12. Using threaded NAPI allows polling to be executed on a separate thread instead of on the way returning from hardirq, so the asynchronous region is smaller. The details of threaded NAPI are described in Chapter 4. However, The most critical aspect of this timing disturbance is that it is **ASYNCHRONOUS**. As long as it happens, there might still be cases the spin limit can be reached, as shown in Figure 5.8.

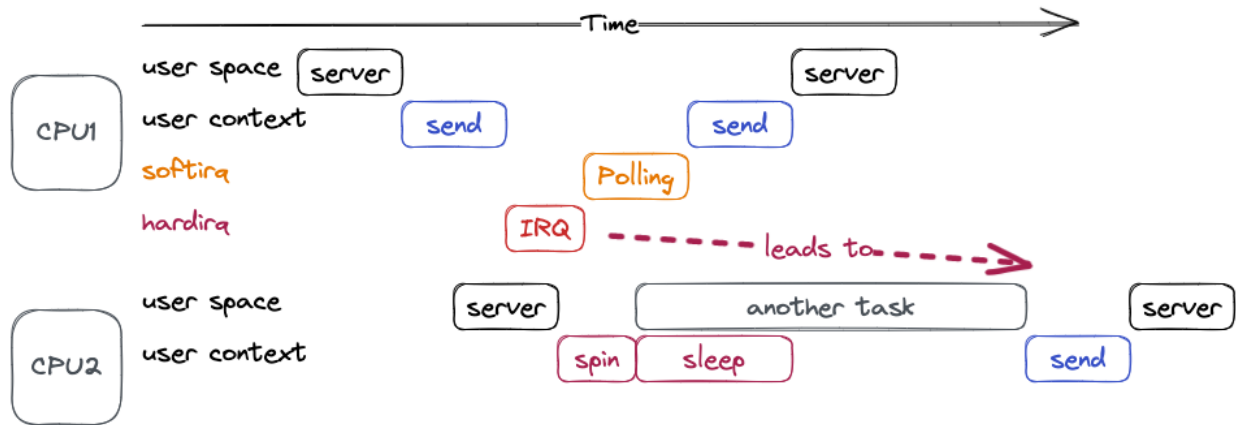


Figure 5.7: CPU can be “stolen” by other task.

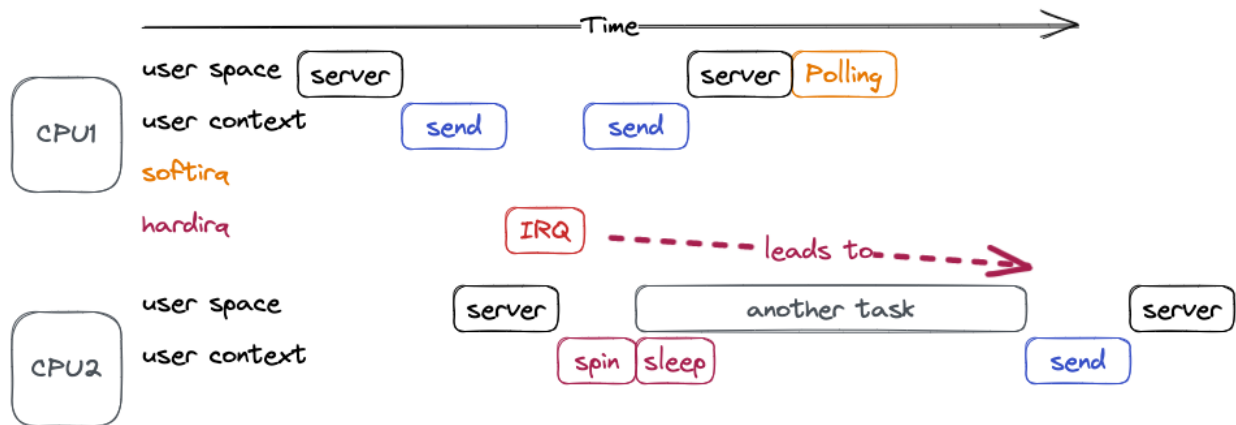


Figure 5.8: Even with threaded NAPI, asynchronous interrupt exits and thread2 is delayed.

## 5.2 QUANTITATIVE ANALYSIS

In this section, we examine how much of a difference in performance we can see from the critical section and locking mechanism interaction. The application used is the in-house workload generation that is described in Chapter 3.

For our experience, we noted three configs for NAPI: - MIX: everything is mixing in the same core(s); - SEP: using a dedicated core for interrupt handling; - tNAPI: threaded NAPI, a new feature introduced in the Linux kernel 5.12.

First, server threads and backgrounds are separated. Only server threads can be run on cores 1-4. HardIRQ and softIRQ of NIC are running on separated core 0. And by

default, the application uses `mutex_lock` as the pure sleeping lock. As shown in Figure 5.9, the server system has below linear scalability from 1 thread to 4 threads due to critical section existence.

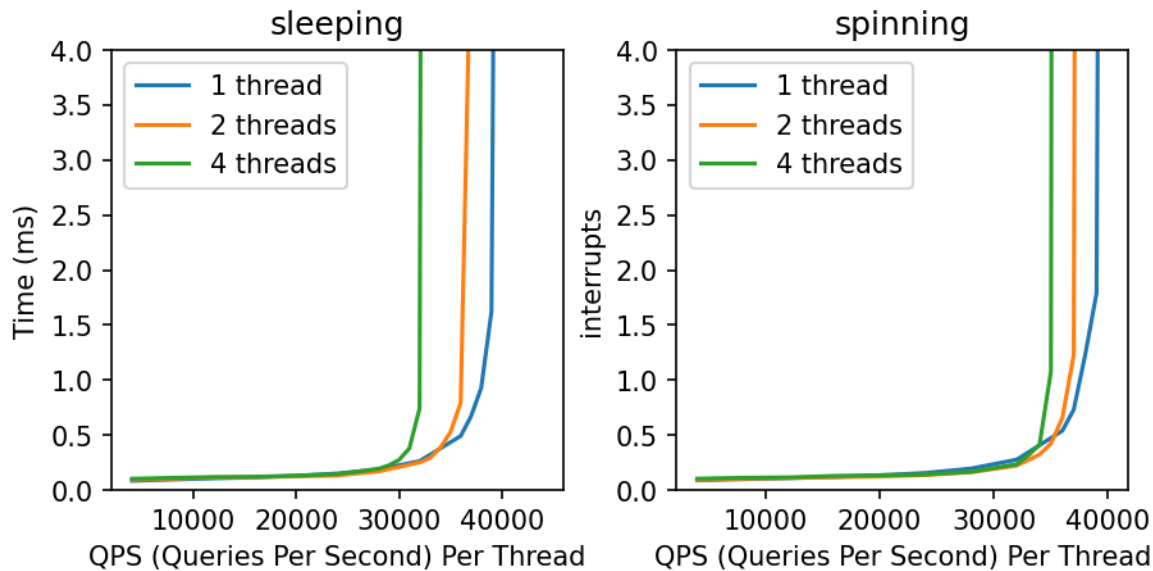


Figure 5.9: Spinning lock improves the threads scaling, saturation points are closer with spinning lock.

Since no other task can use core 1-4, we now have a set of the same application with an unlimited spin limit, which acts as a spinning lock. As a result, the slowdown between thread scaling is more minor, as shown in Figure 5.9.

Now, consider how long the spin limit was good enough. We shift from QPS on the x-axis to the spin limit. As shown in Figure 5.10, the performance is not improving after the spin limit of 1000. The spins afterward are mostly wasting CPU cycles and energy. The improvement of QPS per thread from 32000 to 36000 is 12.5%.

As we discussed earlier, additional hardware resources are only sometimes available. Here, we mix interrupts into our cores 1-4. First, we use the default NAPI in Figure 5.11. The overall performance is worse than the dedicated core, which is expected. Since synchronous interrupts are now happening, the overall rate of getting the lock successfully

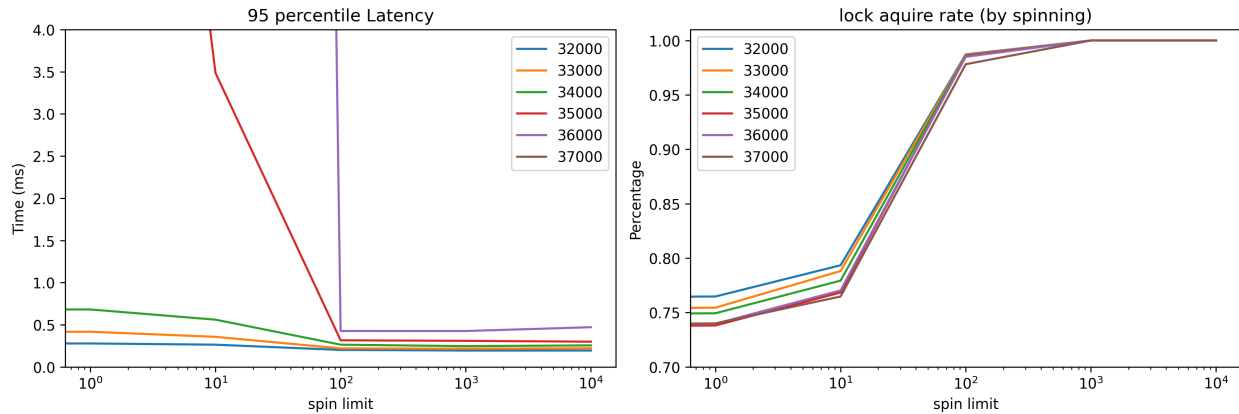


Figure 5.10: Spinning lock improvement, the server can spin forever if no other task can use the CPU. A longer spin limit increases the lock acquire rate by spinning (on the right), leading to better latency (on the left). Different color lines represent different QPS per thread.

is lower. The spin limit is more important here; with the spin limit of 1000, we can push the QPS to 34000 instead of 29000 if the spin limit is 0, which is a 17.2% improvement.

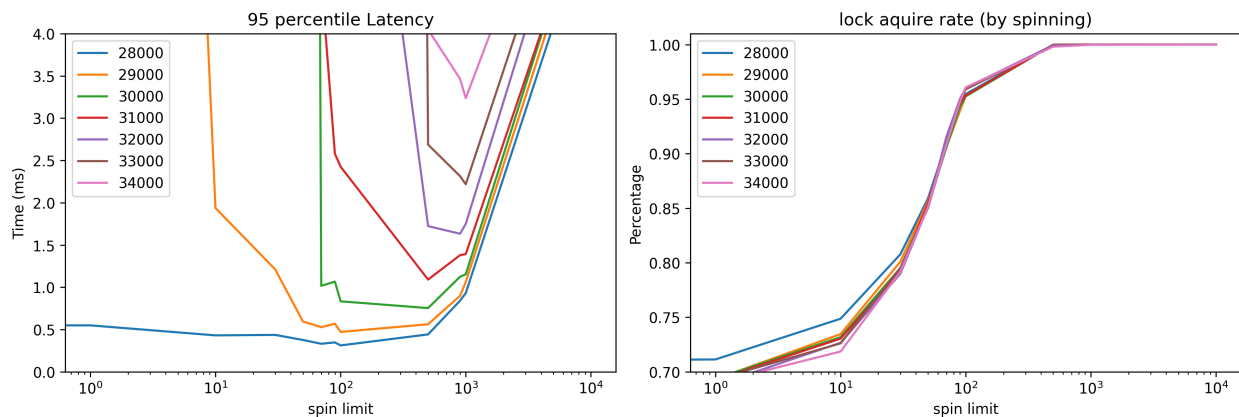


Figure 5.11: Server is more sensitive to spin limit when there is another task that shares the CPU. Here is NIC IRQ. Different color lines represent different QPS per thread. Similarly, the spin limit helps increase the lock acquire rate, but there is an upper bound before it spins too long.

Then we enable the newer feature, threaded NAPI, that delays softIRQ onto its thread, which can be scheduled later, reducing the asynchronous disturbance. Results are shown in Figure 5.12. With the help of tNAPI, we can perform better than using a separate core. The relative throughput improvement is similar. QPS increased from 31000 to 36000 is 16%, but the absolute numbers are higher in tNAPI compared to MIX, showing that reducing the time in asynchronous interrupts helps and the spin limit.

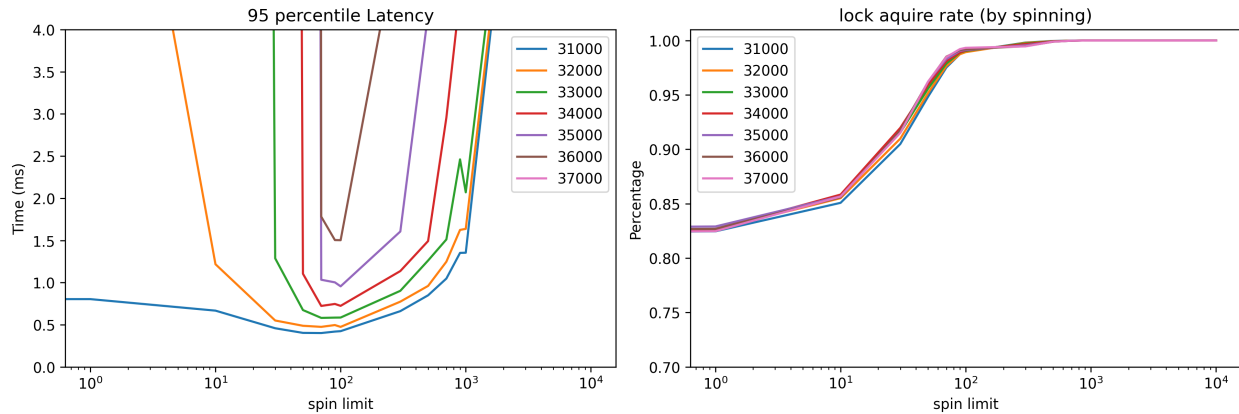


Figure 5.12: Threaded NAPI: reduce the asynchronous region allows a wide range of spin limits for acceptable performance. Smaller timing disturbance allows higher lock acquire to start.

But how much of the impact is coming from not getting the lock? Below in Figure 5.13, we showed that a higher spin limit does help, but only up to a point. In the success case, around 10  $\mu$ s in the beginning due to the mutex access time. As the thread has a higher spin limit, this time increases.

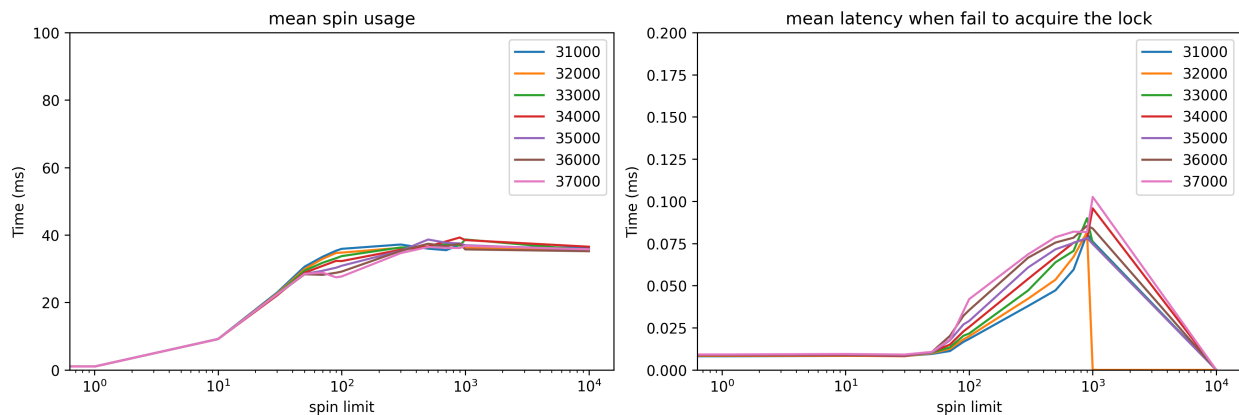


Figure 5.13: Left: Mean spin usage is not increased much after, but it helps for catching longer critical instances. Right: when the spin limit is higher, the one that fails to acquire the lock are longer latency ones, until all are acquired by spinning.

Again, this is when no other threads can take over the CPU. If the server is configured to have more threads (8) than the CPU numbers (4), as we mentioned earlier, the penalty for failing to acquire the lock can be much higher, as in Figure 5.14

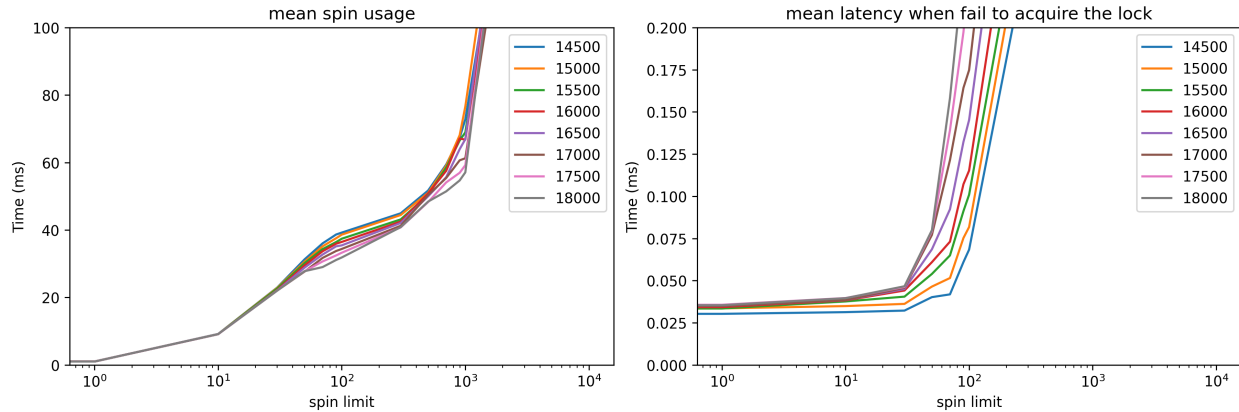


Figure 5.14: 99th tail on the time obtaining the lock: having others can use CPU leads to disaster.

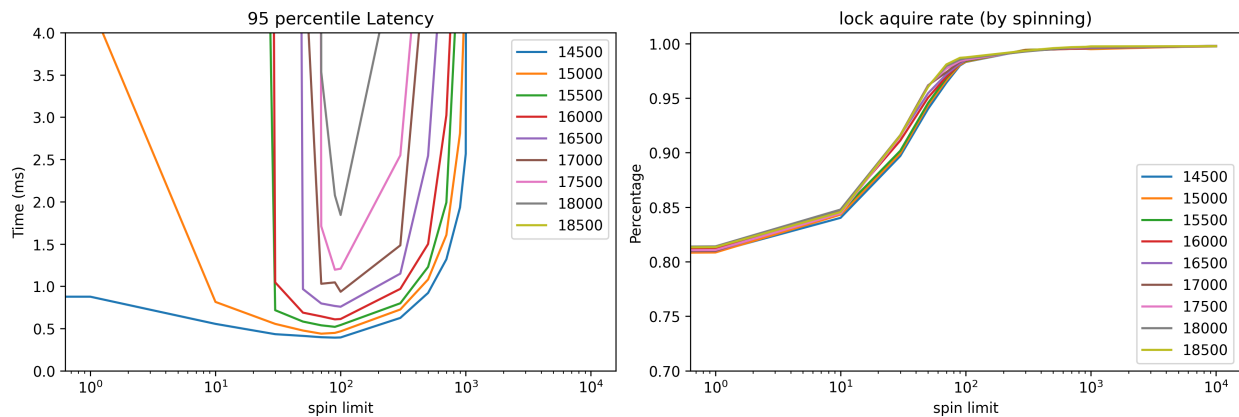


Figure 5.15: 8 threads Threaded NAPI: the spin limit is similar comparing to 4 threads example.

Again, threaded NAPI reduces but does not eliminate hardIRQ. What are the possible improvements we can make? Now, we illustrated the longer penalty in time for failing to acquire to lock and how to increase the rate of acquiring the lock successfully.

Figure 5.16 shows the cascading effects that lead to a longer tail when the server application fails to acquire the lock by spinning. Initially, changing the lock from pure sleeping on the left to hybrid lock in the middle and increasing the spin limit increases the chances of acquiring the lock by spinning. This reduces the ratios of penalties below 0.2 ms when failing to acquire the lock. However, if the spin limit is too high, it introduces longer latency due to hardware starvation. Thus, picking a reasonable spin limit in between is essential.

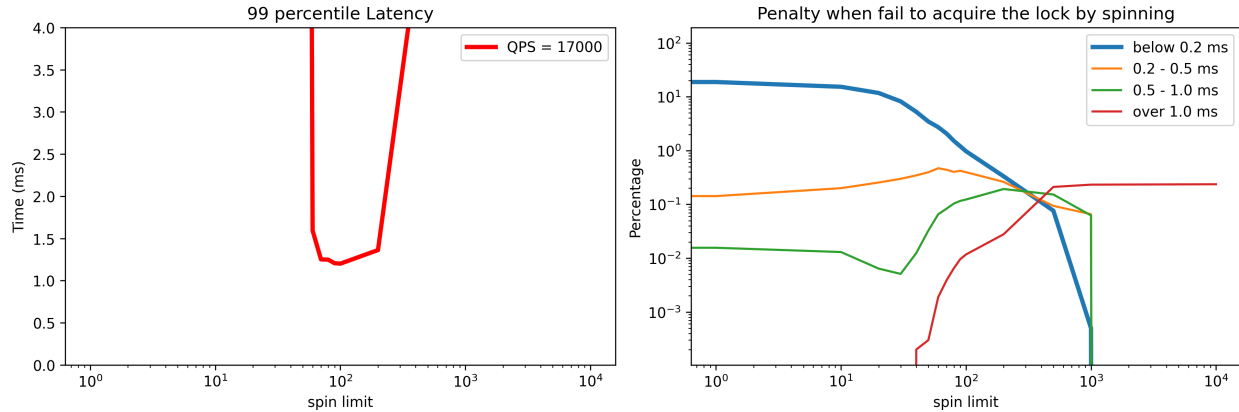


Figure 5.16: Cascading effects: increasing the spin limit for application reduces the penalties for shorter tails (blue line on the right). However, if the spin limit is too high, it increased even longer tail, which is unwanted.

### 5.3 POSSIBLE IMPROVEMENT

Here, we translate “the improvement from getting rid of hardIRQ” into “the issue brought in from hardIRQ.” First, let us look into CPU time spent on different purposes. Then, for a second period, the system can have active service time, RX (recv) critical, TX (transmission, send) critical section, and idle time, with misc times such as switching threads, etc. Figure 5.17 shows an example breakdown of time.

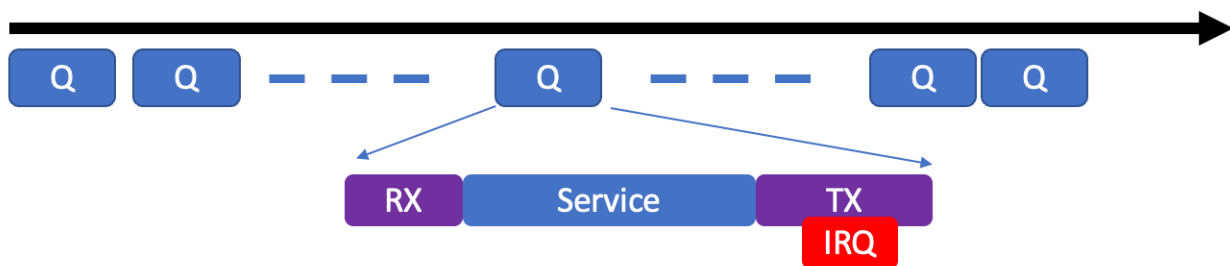


Figure 5.17: Break down of CPU time spend in different region.

For each query, we have:

$$T_{rx} + T_{svc} + T_{tx}$$

, where  $T_{rx}$  and  $T_{tx}$  are critical sections shared by all application threads.

For each second, we have:

1. Time in TX critical section:

$$QPS * T_{tx}$$

2. Asynchronous NIC interrupts are happening with a rate noted as IRQPS, then we can have IRQPS that occurs within a critical section

$$IRQPS * (QPS * T_{tx})$$

3. Affected Quests percentage

$$IRQPS * (QPS * T_{tx}) / QPS$$

$$= IRQPS * T_{tx}$$

The higher the IRQPS we have, the higher the percentage of quests that will get affected by IRQs. However, as we know, a higher rate does not mean the highest number.

The next question is, when do we have the most IRQPS? It is not when we have the highest QPS.

Here, in Figure 5.18, we showed the IRQPS, percentage of affected QPS, and affected QPS number from left to right.

We notice that the IRQPS number starts higher when QPS is higher when the spin limit is small. When the spin limit is higher, the IRQPS drops down more rapidly for higher QPS since more time is spent in  $T_{svc}$  the region, less time for the networking process.

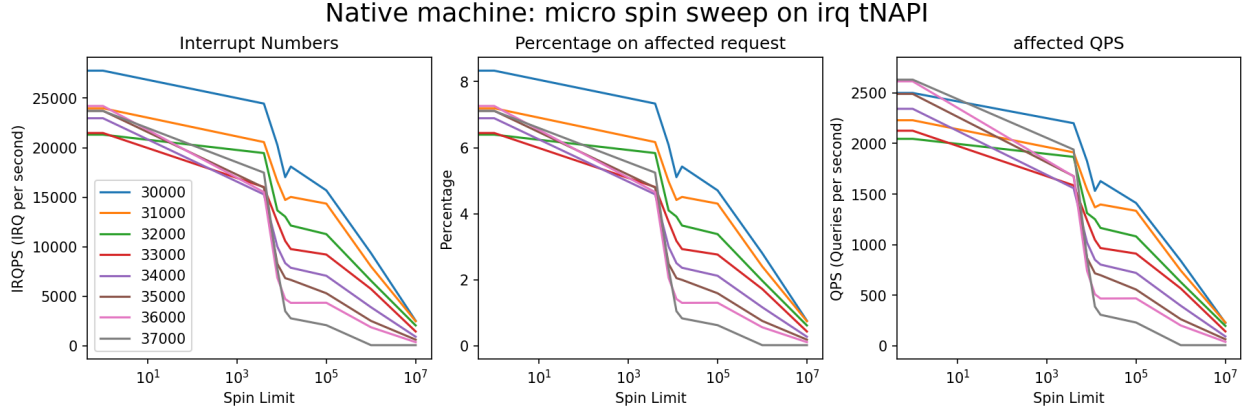


Figure 5.18: Higher IRQPS leads to higher percentage of request affected by IRQ.

Then, for affected QPS, similarly, within each figure, the highest QPS have the highest numbers of affected number and percentage, meaning the most affected region. But, as we move to a higher spin limit, lower QPS has a higher influence by IRQ.

Within the percentage, we denote  $P_{affected}(QPS, SPIN)$  Each data point, and will be used for later calculations. Here we assume the affected requests are evenly distributed across the time, meaning it has an equal chance of happening in  $T_{svc}$  ,  $T_{TX}$  .

#### 5.4 SUMMARY

This chapter showed how  $\mu s$  scale timing disturbance from NIC interrupt handling can lead to cascading effects on server applications with multiple threads sharing hardware resources. First, we conducted the walk-through on building the case. Then it leads to a quantitative analysis of an example workload. Then, we analyzed the improvement space to eliminate these cascading effects. In the next chapter, we propose a hardware accelerator that targets this.

## 6 INTERRUPT HANDLING ACCELERATOR

The previous two chapters introduced the threaded NAPI and how it can help the cascading effects from IRQ mixing with synchronization. This chapter proposes the first hardware accelerator that targets interrupt handling.

In modern-day, computer systems have become more and more complex. As a result, a single group of developers can only provide a partial-stack solution. Motivated by this, two key concepts have been proposed and developed: abstraction and encapsulation. The computing society has different models consisting of abstraction layers that define a high-level generalization of conceptual models, such as the OSI model for networking. On the other hand, encapsulation allows individual solutions to be portable and applied to different scenarios.

Design is often broken down into small pieces in different abstraction layers. This method of focusing on smaller targets speeds up research and development. The downside is each small design is limited by the scope of focus.

There may be a mismatch of trade-offs that comes from different abstraction layers. This is also true for our research focus: the server system. For a server system, it can break down into hardware level, kernel level, and application level. Within each level, there are more substrates of groups. Within each group, people are trying to provide a solution with better performance: low latency, low energy, and low power.

Also, on a higher level, the system requires portability and ease of use, which is not impossible to achieve. The kernel is an excellent example of compatibility introducing redundancy everywhere. Chapter 4 shows that the core task is to schedule the NAPI

thread. Different hardware components and layers of function calls need to be performed to accomplish this task.

Of course, different layers of function calls and condition boundaries ensure the Linux kernel is working smoothly. However, the specific NIC interrupt is small and happens frequently. It makes sense to seek an alternative approach.

A tightly coupled hardware accelerator can speed up the process, and the architecture view is shown in Figure 6.1.

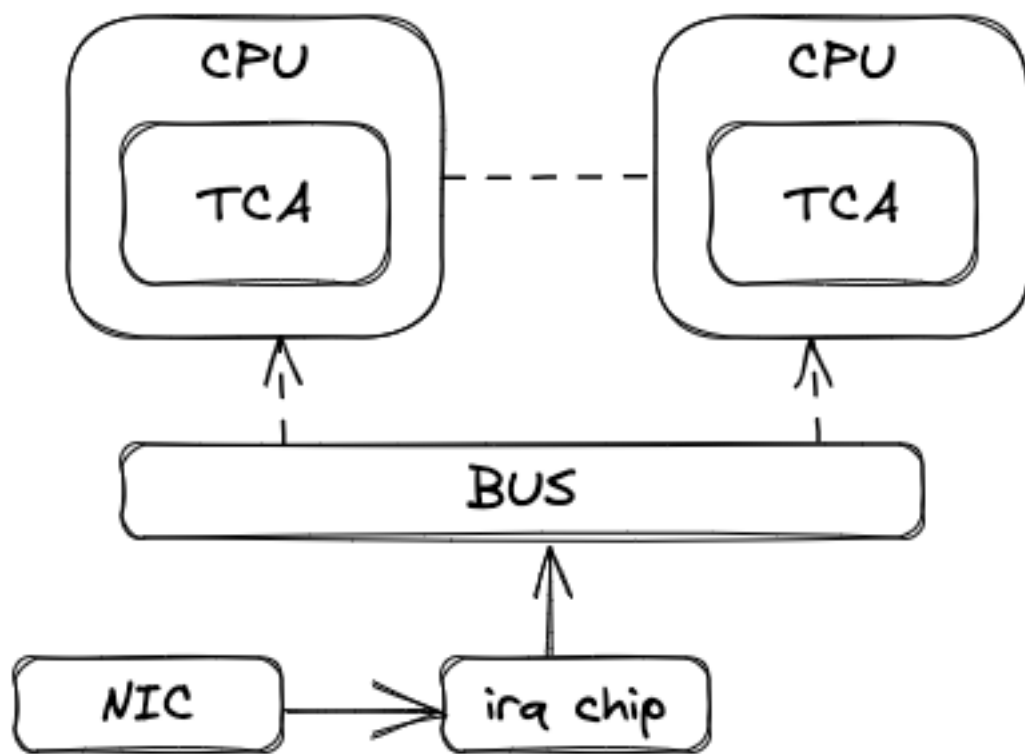


Figure 6.1: Chip Level overview: interrupt signal sent to CPU through irq chip and bus. The proposed accelerator is resident within the CPU, it does not alter the interface between devices.

In this chapter, we dive into details on how modern systems handle interrupts. Then, how does ethernet specific NAPI interrupt handling scheme help?

This chapter follows the organization below:

1. Chapter 6.1 dives into steps for the Linux kernel to handle an Ethernet interrupt and how it motivates a hardware accelerator solution.

2. Chapter [6.2](#) described the hardware accelerator design.
3. Chapter [6.3](#) shows how to configure the hardware accelerator.

## 6.1 OPPORTUNITY FOR HARDWARE ACCELERATION

For performing the hardware interrupt routine for the NIC, there are four major steps:

1. Generic interrupt
2. NIC devices interrupt
3. NAPI: switching from interrupt mode
4. NAPI: schedule polling thread

We show the kernel function call stacks and source files for each step and identify the essential step. Interrupt handling logic serves the attention needed from hardware. Thus, we start from NIC hardware as the source and lowest level and trace it up to the kernel-managed scheduler as the highest level. The complete list of function calls is included in [Appendix 10](#) for completeness.

### 6.1.1 *Generic Interrupt*

First, we discuss how the CPU can initiate and store the interrupt information. Then, locate the action for a specific interrupt number. In the x86 architecture, the irq chip device is called Programmable Interrupt Controller (PIC) or Advanced Programmable Interrupt Controller (APIC). In Arm, it is called Generic Interrupt Controller (GIC). This irq chip is in charge of receiving interrupt signals from devices and directing traffic to target CPUs.

## Initialization

When the system boots up, it needs to initialize a table of interrupt entries corresponding to different types of interrupts. In x86, it is called Interrupt Descriptor Table (IDT); in arm/arm64, it is called Interrupt Vector Table (IVT).

The table begins with reserved interrupts that are predefined in the Linux kernel. Then, there are non-reserved spaces for user-defined interrupts. These two terms are used in the x86 programming guide ([“Intel® 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4” 2023](#)). Here, the “user-defined” is a misleading term. It is not defined explicitly by a user in user space but is used for devices connected to the APIC. The connected devices will be registered under this “user-defined” interrupts region. Linux source file (arch/x86/include/asm/irq\_vectors.h) reflects this definition.

The specific interrupt relating to most device interrupts, including NIC, is called “common interrupts” in x86. The entry for “common interrupts” is defined in the below function:

---

```

1 // arch/x86/kernel/irq.c
2 DEFINE_IDTENTRY_IRQ(common_interrupt)
3 // arch/x86/include/asm/identry.h
4 #define DEFINE_IDTENTRY_IRQ(func)
5 DECLARE_IDTENTRY_IRQ(X86_TRAP_OTHER, common_interrupt)

```

---

Within the identry.h, there are other IDT entry definition macros, such as the simple exception for divider error or overflow error (DECLARE\_IDTENTRY). Also, irq\_enter and irq\_exit are two functions wrapping every hardware interrupt and added by definition.

As far as we know, I/O devices trigger either level or edge interrupt, x86 (manual section 10.1, page 363) and ARM (<https://developer.arm.com/documentation/duio552/a/Bhchgeei>) specifications indicated.

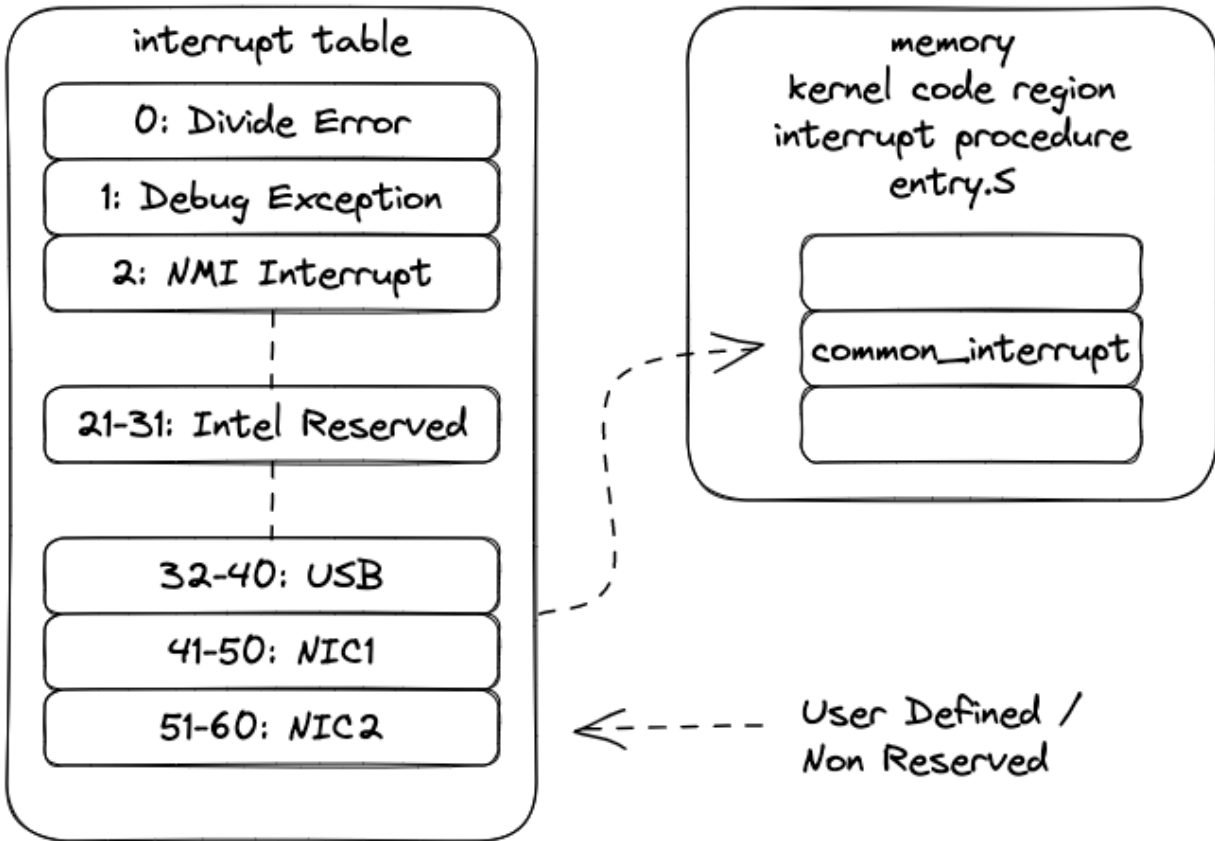


Figure 6.2: Interrupt table: it begins with predefined interrupts, end with user defined interrupts. When an interrupt happens, the CPU uses address stored in the interrupt table to local entry point, defined in the linux kernel

Furthermore, for devices registering interrupts going through the irq chip (APIC), two different interrupt handlers were set, as shown below:

---

```

1 // arch/x86/kernel/apic/io_apic.c
2 mp_register_handler {
3     ...
4     hdl = fasteoi ? handle_fasteoi_irq : handle_edge_irq;
5     __irq_set_handler(irq, hdl, 0, fasteoi ? "fasteoi" : "edge");
6 }

```

---

“handle\_fasteoi\_irq” is set for level-triggered irq, and “handle\_edge\_irq” is set for others. In our study, NIC devices are edge-triggered irq for both native machines (x86) and in simulation (ARM).

## Locating

Now, when the NIC device raises an interrupt, it passes through the irq chip (APIC for x86), and arrives on the target CPU at runtime. The CPU will take the interrupt based on the signal and interrupt table initialized.

The CPU does not know which interrupt to handle at this step. The CPU only cares and knows it is a `common_interrupt` entry point. During this step, the CPU switches from user context to hardware interrupt (`hardIRQ`). Then, it locates the specific interrupt handling routine based on the devices. The kernel function call stack is shown below:

---

```

1 // kernel/softirq.c
2 irq_enter
3 // arch/x86/kernel/irq.c
4 common_interrupt
5     handle_irq
6 // include/linux/irqdesc.h
7     generic_handle_irq_desc
8 // kernel/irq/chip.c
9     handle_edge_irq
10 // kernel/irq/handle.c
11         handle_irq_event
12             handle_irq_event_percpu
13                 __handle_irq_event_percpu
14                     res = action->handler(irq, action->dev_id);
15 // kernel/softirq.c
16 irq_exit

```

---

Full functions stacks are attached in the Appendix [10.1.1](#).

During this process, the conditions that need to be checked are:

---

```

1 // arch/x86/kernel/irq.c
2 common_interrupt
3     desc = __this_cpu_read(vector_irq[vector]);
4     !IS_ERR_OR_NULL(desc)
5 // kernel/irq/chip.c
6 handle_edge_irq

```

---

```

7      IRQD_IRQ_INPROGRESS not set
8      IRQD_WAKEUP_ARMED not set
9      IRQD_IRQ_DISABLED not set
10     irq has action

```

---

Here, we simplified and kept the important checks. The idea is to keep the minimum checks and only handle the most common case when everything is expected. The simplified generic irq layer pseudocode is as below:

---

```

1 void generic_interrupt() {
2     IRQ_NUM = read(IRQ_LINE);
3 start:
4     irq = vector.get(IRQ_NUM);
5     irq.action(); // nic_interrupt entry
6     write(IRQ.EOI, IRQ_NUM);
7     IRQ_NUM = read(IRQ_LINE);
8     if IRQ_NUM != 0xFF {
9         // end of the vector table
10        goto start;
11    }
12 }

```

---

At this point, the Linux kernel located the specific action that needs to be taken for NIC interrupt, which calls `igb_msix_ring`, if the `igb` driver is being used.

### 6.1.2 NIC Devices Interrupt

In the second step, the CPU interacts with the Ethernet device. Below we show the `igb_msix_ring` function in the kernel source code. Since there is only one function call in this step, we show the whole procedure in this listing.

---

```

1 // drivers/net/ethernet/intel/igb/igb_main.c
2 static irqreturn_t igb_msix_ring(int irq, void *data)

```

```

3 {
4     struct igb_q_vector *q_vector = data;
5
6     /* Write the ITR value calculated from the previous interrupt. */
7     igb_write_itr(q_vector);
8
9     napi_schedule(&q_vector->napi);
10
11     return IRQ_HANDLED;
12 }

```

---

From what we saw, the igb driver-specific function is short. It does two things: update Interrupt Throttle Rate (ITR) and call `napi_schedule`.

ITR is a hardware register that provides the tuning mechanism for interrupt arrival intervals. Setting this value to a higher number causes a longer time for the NIC devices to wait until sending the next interrupt signal. This can be overwritten by kernel tools such as `ethtool`. For better latency, this usually sets to 0.

---

```

1 ethtool -C eno0 rx-usecs 0

```

---

With no update to ITR, calling `igb_msix_ring` falls through to the generic napi layer.

### *Other NIC Drivers*

There are other common NIC drivers, such as E1000/E1000e. Here, the NIC layer can have more to do. One common addition is needing to read the ICR (interrupt control register). By reading the ICR, the kernel knows the specific reasons causing the interrupt. ICR read can generate an early return if none of the bits are set. The bit corresponding to causes could be new packets on receiving ring or transmission entry consumed. Since there might be a false interrupt when no bits are set, using this can ensure there are bits set corresponding to any interrupt. As long as the valid bits are set, it calls `napi_schedule`.

Reads to ICR also have an auto pairing write to the IMR (interrupt mask reset) register. Writes to IMR will disable future interrupts, and the CPU switches from interrupt mode.

It is worth noting that this step switches from interrupting, and the next step switches to polling. The mask will be set once polling is finished and the CPU resumes to interrupt mode.

In this case, the conditions that need to be checked are:

---

```

1 // drivers/net/ethernet/intel/e1000/e1000_main.c
2 e1000_intr
3     ICR is valid
4     __E1000_DOWN is not set
5     NAPIF_STATE_DISABLE is not set
6     NAPIF_STATE_SCHED is not set

```

---

To summarize, combining both the device drivers and the NIC interrupt layer pseudocode is shown below:

---

```

1 int nic_interrupt() {
2     # if E1000/E1000E
3         icr = read(ICR_REG);
4         if ( !icr.tx || !icr.rx)
5             return;
6     # endif
7     napi_schedule();
8     return IRQ_HANDLED;
9 }

```

---

### 6.1.3 NAPI: switching from interrupt mode

The NAPI layer, by design, is light and does the minimum amount of work. All it does is check the napi struct's state, then transit to polling mode. Each queue has its napi struct

for a multi-queue NIC setting, which causes individual interrupt numbers. If the current state is running already, interrupt handling can finish and exit early.

In this layer, these files and functions are touched:

---

```

1 // include/linux/netdevice.h
2 napi_schedule
3 // net/core/dev.c
4     __napi_schedule
5     ____napi_schedule

```

---

If the struct is valid, NAPI switches to polling mode. For classic NAPI, a softirq is scheduled. For threaded NAPI, the napi thread needs to wake up by calling `wake_up_process()`.

These conditions need to be checked:

---

```

1 // net/core/dev.c
2 napi_schedule_prep
3     NAPIF_STATE_SCHED flag is set
4 ____napi_schedule
5     NAPI_STATE_THREADED flag is set

```

---

To summarize, the napi layer pseudocode is shown below:

---

```

1 void napi_schedule(){
2     if (napi->state == running)
3         return;
4     napi->state = running;
5
6     if threaded_NAPI
7         schedule(napi->thread);
8     else
9         raise_softirq(napi);
10 }

```

---

Here, we keep both the path of classic softirq and threaded NAPI approaches. The interrupt handling is the same until this part.

#### 6.1.4 NAPI: schedule polling thread

This is the new addition to the threaded NAPI path.

As shown in the listing, there are more memory accesses for adding a task onto a running queue than other parts of the interrupt handling. The core of scheduling is modifying the list of runqueue. Runqueue is a per-core data structure that maintains the tasks that are being scheduled to run. Essentially, the interrupt process must write to these four memory places. All others are addition steps needed to achieve this one goal.

It can be split into two sections. The first section is the generic process of the waking process; the second section is related to a real-time (round-robin) process.

---

```

1 // kernel/sched/core.c
2 wake_up_process
3     try_to_wake_up
4         ttwu_queue
5             ttwu_do_activate
6                 activate_task
7                     enqueue_task
8                         specific runqueue task
9                             ttwu_do_wakeup

```

---

These conditions need to be checked within the generic schedule core:

---

```

1 // kernel/sched/core.c
2 try_to_wake_up
3     task_to_wake is not current_task
4     task_to_wake is not running
5     task_to_wake is runnable
6     CPU is not holding the scheduler lock

```

---

For a task that has a real-time priority, the specific functions are listed below:

---

```

1 // kernel/sched/rt.c
2 enqueue_task_rt
3     enqueue_rt_entity
4     __enqueue_rt_entity
5         list_add_tail

```

---

The only condition check that can lead to a false return is related to group scheduling, which is irrelevant. Thus, this part needs no condition check for the accelerator.

The core of scheduling a task is putting it onto the running queue. This is the core of the whole NIC interrupt process.

The runqueue has a data structure type of a doubly-linked list. Adding an element needs to modify four memory locations: tail->prev->next, napi->thread->prev, napi->thread->next, and tail->prev.

Then, the real time scheduling logic needs to update run queue stats, such as task\_running (number of tasks running); rt\_task\_running (number of real-time priority tasks running); rt\_queued (the real-time task is queued); priority array, etc. Also, it needs to keep thread information updated: on\_rq (the task is on runqueue); the state is running.

To summarize, the scheduling layer pseudocode is shown below:

---

```

1 int schedule(thread) {
2     if (thread == current_task ||
3         thread->on_rq) {
4         return;
5     }
6     rq->task_running++;
7     rq->rt_task_running++;
8     rq->rt_queued = 1;
9     rq->prio |= thread->prio;
10
11     old_prev = read(rq_list->tail->prev);
12     write(rq_list->tail->prev, thread);
13     write(thread->next, rq_list->tail);

```

```

14  write(thread->prev, old_prev);
15  write(old_prev->next, thread);
16
17  thread->on_rq = 1;
18  thread.state |= TASK_RUNNING;
19  }

```

---

## 6.2 ACCELERATOR DESIGN

We have described the essential steps that need to be performed for the NIC interrupt handling. With that, we proposed a tightly coupled accelerator (TCA) within the CPU pipeline. First, the TCA intercepts the interrupt signal. Then, when the specific interrupt signal arrives, the TCA will try to do the fast routine. Figure 6.3 shows the high-level view of the accelerator design, a zoomed-in version from Figure 6.1.

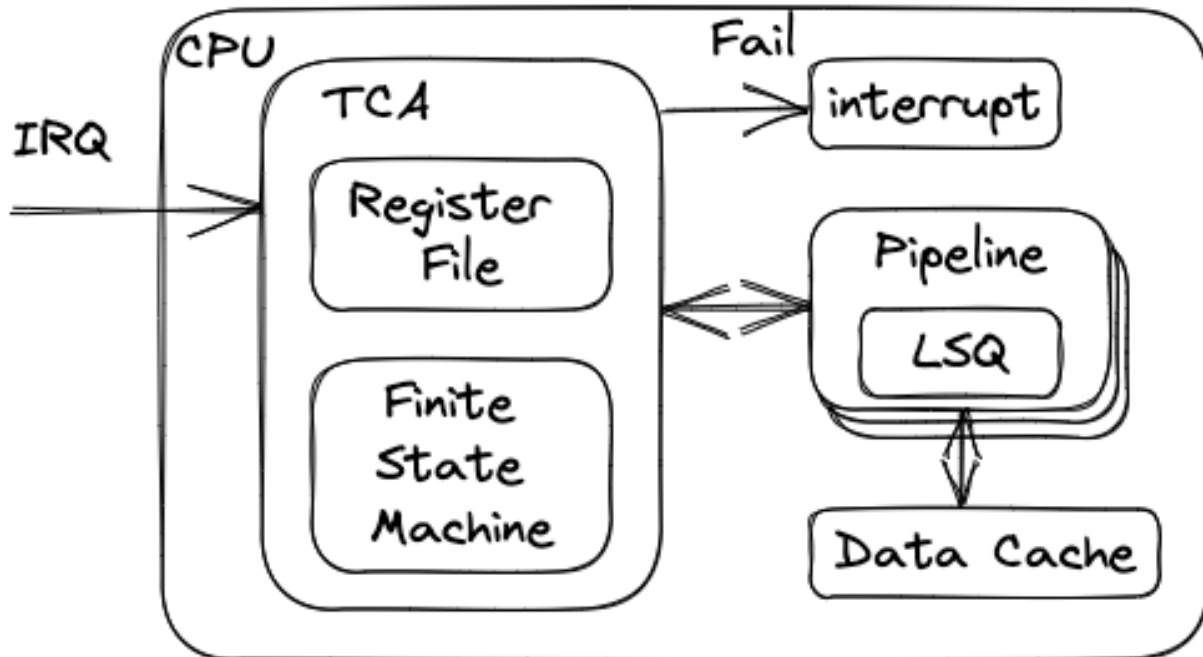


Figure 6.3: Accelerator overview: within the CPU, the IRQAcc behaves as Finite State Machine.

Unlike many TCAs that require input arguments for every invocation, the proposed IRQ TCA does not require input arguments. Instead, within the accelerator, two main parts of work are to be performed: conditions to be checked and action to take if conditions are. Both of these require addresses for data access in the memory. These values are held in registers within the TCA. The values to be checked against and written are obtained from kernel source code. Then, there is simple combinational logic for condition check and minimum calculation, such as increment.

The TCA contains a register file storing addresses and data and the combinational logic for actions. The TCA can issue memory read and write directly to the core pipeline's LSQ. As a result, the LSQ can use the existing memory hierarchy to enforce memory coherence and consistency for TCA memory access.

The TCA Finite State Machine (FSM) is shown in Figure 6.4.

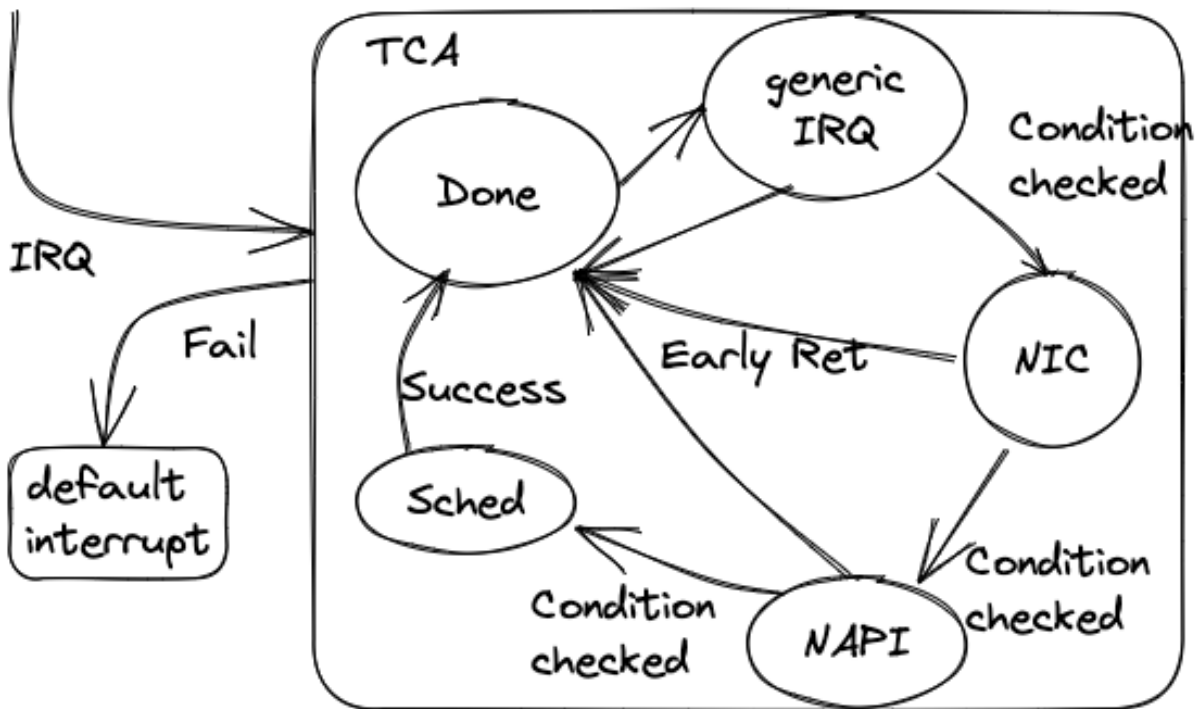


Figure 6.4: Accelerator FSM: whenever IRQ arrives, TCA checks condition, take actions if meet and processed; else, return pass IRQ to default path or early return success.

In our design, the FSM does not generate a signal output when the interrupt is successfully handled. For interrupt handling, there are no return signals to the irq chip. During the last stage of interrupt handling, write the interrupt number to a specific register indicating the end. If the TCA fails to handle the interrupt, it lets the same interrupt signal fall through to the default interrupt logic.

### 6.3 ACCELERATOR CONFIGURABILITY

We can use the TCA with hard-coded logic. But, it only works with a specific combination of hardware, kernel code, and runtime environment.

Above, we used x86 ISA, igb device driver, and real-time round-robin scheduler as an example. Besides these three parts, generic interrupt handling, NAPI, and generic scheduling will be the same.

Thus, it is ideal that the TCA has reconfigurability. We propose a set of instructions for pushing the conditions and providing corresponding actions to the TCA. Since we analyze the kernel code, only a specific range of behaviors can be allowed. The reconfigurability allows targeting different NIC devices, irq chips, or with identical hardware but kernel updates.

---

```
1 push_condition(address, value, ret);
```

---

For the condition that is pushed, address points to the data need to be read from memory, and the value to be checked against. The ret field indicates two paths when the condition is not met. A “fail” from the TCA returns the interrupt signal to the CPU generic interrupt logic. An early return means successful returns, and IRQ is handled.

---

```
1 push_action(read, address, reg);
```

---

```

2  push_action(write,address,reg);
3  push_action(add,dest,src,imm);

```

---

The more straightforward action addresses loading the value, source, and destination register for temporal storage within the TCA.

Of course, the TCA needs reset instructions. Since the reset rarely happens, such as kernel update or system bootup, we expect a single reset of the whole TCA, followed by push instruction sequences to configure conditions and actions.

```

1  reset();

```

---

Putting these together, we have:

```

1  reset();
2
3  push_condition(runqueue->lock,0,fail);
4  push_condition(irq_number,nic_irq,fail);
5
6  push_condition(cpu_context,in_hardIRQ,early_return);
7  push_condition(napi->thread,current_task,early_return);
8  push_condition(napi->thread->running,0,early_return);
9
10 ...
11 // sequence for rq->task_running = rq->task_running+1
12 push_action(read,rq->task_running,reg1);
13 push_action(add,reg1,reg1,1);
14 push_action(write,rq->task_running,reg1);
15 ...

```

---

Then, this code will be converted to assembly, embedded within the kernel, and executed when the system boots up.

This sequence needs to be handwritten based on the hardware and kernel combination. As the design becomes complicated, automation is expected in code generation, which will be discussed more in our Future Work section.

One can imagine obtaining all the addresses is cumbersome. To mitigate the complexity, we discuss how to get all the addresses quickly in the next section.

### 6.3.1 Obtaining Addresses

As we walked through in an earlier section, most registers values are defined when the kernel is compiled.

pahole is a useful tool in Linux that can be used to show the data structure layout of a binary. For a compiled kernel, we can use objdump to display the information for the compiled kernel binary with debug information. Then, pahole can be used to check individual data structures as shown below:

---

```
1 pahole -C task_struct vmlinux.arm64.5.15.binary
```

---

It shows all variables within this data structure, including type, name, starting offset, and size. The output is layedout nicely as below:

---

```
1 struct napi_struct {
2     struct list_head      poll_list;          /*    0    16 */
3     long unsigned int     state;               /*   16    8 */
4     ...
5     struct task_struct *  thread;             /* 384    8 */
6     ...
7 } __attribute__((__aligned__(8)));
```

---

The output shows that if a base address for the `napi_struct` is `A`, `napi_struct->state` is pointed by the address of `A+16`, and the `napi_struct->thread` has an address of `A+384`. Furthermore, now with the address of the napi thread, we can get the address of data related to this struct, such as `napi_struct->thread->on_rq` is `*(A+384)+96`, as shown below:

---

```

1  struct task_struct {
2      struct thread_info      thread_info;          /*      0      16 */
3      unsigned int            __state;               /*     16      4 */
4      ...
5      int                     on_rq;                 /*     96      4 */
6      int                     prio;                  /*    100      4 */
7      ...
8      unsigned int            rt_priority;           /*    112      4 */
9      ...
10 } __attribute__((__aligned__(8)));

```

---

With the combination of `pahole` and `objdump`, we can determine all the addresses with the base for each component.

## 6.4 SUMMARY

This chapter described the tightly coupled accelerator (TCA) for interrupt handling, named IRQAcc. First, we dived into detailed steps on handling NIC interrupts, from signaling from hardware devices to the CPU and corresponding kernel source codes on this path. Then, we described how a TCA within the CPU can help. Next, we walked through the architectural design within the accelerators and the interaction with the default interrupt handling logic within the CPU. Also, we described how the TCA accesses memory to achieve the same function. Finally, to generalize our design, we provide the reconfigurability of pushing conditions and actions into the TCA.

## 7 ADAPTIVE HYBRID LOCK ACCELERATOR

In the previous chapter, we proposed a TCA (Tightly-Coupled Accelerator) for NIC interrupt handling, motivated by the threaded NAPI and cascading effect from the interaction between it and the server application. In this chapter, we targeted another factor contributing to the cascading effect, described in Chapter 5: the adaptive hybrid lock. As a result, a TCA is applicable in this case.

For multi-thread applications, mutex is essential as the synchronization mechanism between different threads. In chapter 5, we illustrate that either pure spinning or sleeping is not ideal, and a hybrid of spinning and sleeping is preferred. Using the POSIX pthread library, we can implement this hybrid lock in user space. The hybrid lock keeps threads spinning for a short time and blocks for a longer waiting time after reaching the spin limit.

Here, we motivate with an example. For the application with multiple threads sharing the same file descriptors, two locks are being used to guard the receiving and transmitting critical sections.

Figure 7.1 shows that the two locks' desired spin limit ranges differ. The receiving spin limit range will be higher than the transmitting one. Picking the proper spin limit is challenging. This hints at a more intelligent choice of the spin limit under different scenarios, and based on the run time status, other spin limits are preferred. Thus, an adaptive hybrid lock that can adjust at runtime can be helpful.

Furthermore, frequent access to these data structures hinting a hardware solution for better efficiency.

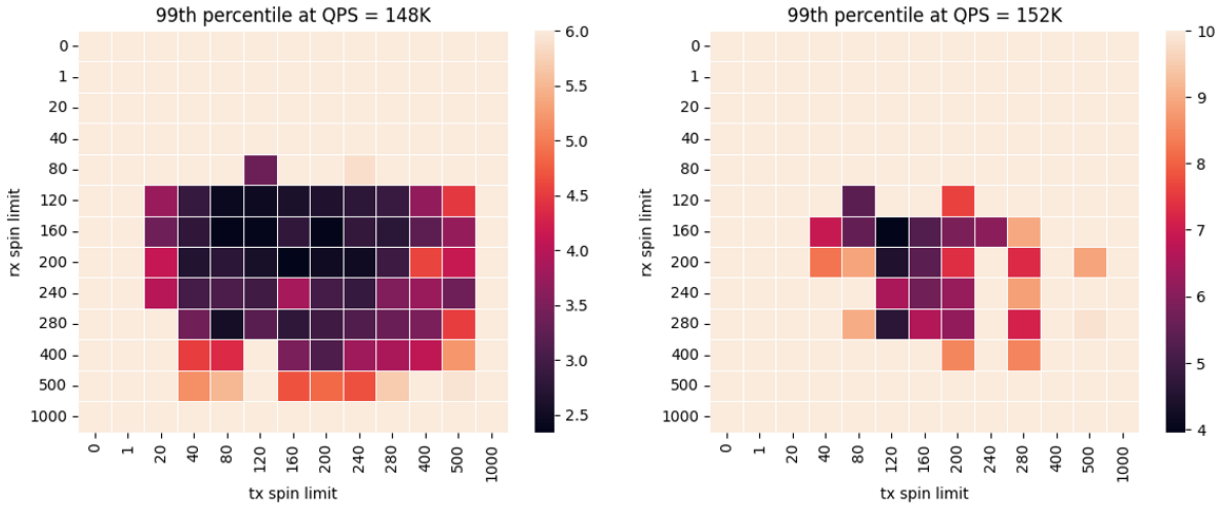


Figure 7.1: 2d heatmap of 99 percentile latency for two different loads. The range of ideal spin limit is different on two different locks, for two different loads

The LockAcc, instead of spinning (i.e., repeatedly loading the lock value), monitors invalid traffic to detect a change to the lock variable. This approach is similar to the one used by Intel’s monitor/mwait instructions, which place the kernel idle loop into a low-power CPU sleep state until a memory write wakes it up ([“Intel® 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4” 2023](#)).

Effectively, the TCA consumes virtually no energy while spinning relative to the instruction execution overhead of the CPU executing a regular spin loop.

The rest of this chapter is organized as follows:

1. Chapter 7.1 describes the existing method used in software to implement a hybrid lock.
2. Chapter 7.2 covers the sample adaptive spin limit tuning algorithm.
3. Chapter 7.3 proposes the accelerator design, offloading the adaptive hybrid lock into hardware.

## 7.1 EXISTING SYNCHRONIZATION DESIGN

One of the commonly used lock methods is the POSIX library mutex implementation. In our native machine experiment, we implemented a hybrid lock using the pthread mutex library. However, there are limits to the existing design in choosing the optimal spin limit and wasting CPU cycles on spinning. Therefore, we would like to introduce the current flow of the mutex.

The pthread mutex contains two parts: 1. glibc library pthread\_mutex 2. kernel provided futex subsystem

### 7.1.1 *Glibc*

Applications in modern Linux systems use a standard library for compatibility. POSIX (Portable Operating System Interface) is a collection of IEEE standards for this purpose. Pthread (POSIX thread) specifies the standard for threaded programming, including process ID, priority, and scheduling information. There are two implementations of pthreads: LinuxThreads and NPTL (Native POSIX Threads Library). Over the years, LinuxThreads were no longer supported since glibc 2.4. NPTL is the current standard for modern pthread implementation.

For an application that uses pthread\_mutex, the synopsis is as below:

---

```
1  #include <pthread.h>
2  int pthread_mutex_lock(pthread_mutex_t *mutex);
3  int pthread_mutex_trylock(pthread_mutex_t *mutex);
4  int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

---

`pthread_mutex_lock` is used to acquire the sleeping lock. During sleeping lock acquisition, the thread will be blocked if another thread currently owns the lock. The library call only returns when the lock is acquired.

`pthread_mutex_trylock` is used in the spinning lock mechanism. This library call also returns when the lock is not acquired, and a value of 0 indicates another thread currently has it.

Both sleeping lock and spinning lock use `pthread_mutex_unlock` to release the lock.

In addition to the compatibility, using a standard library in user space access allows lower latency in acquiring the lock if successful. The thread only makes a syscall when the lock is contended. Below we show the sleeping lock (`pthread_mutex_lock`) path within the glibc source code:

---

```

1 // nptl/pthread_mutex_lock.c
2 PTHREAD_MUTEX_LOCK(*mutex)
3     LLL_MUTEX_LOCK_OPTIMIZED(*mutex)
4     lll_lock(*mutex)
5 // sysdeps/nptl/lowlevellock.h
6 #define lll_lock(futex, private) __lll_lock (&(futex), private)
7 __lll_lock
8     if (atomic_compare_and_exchange_bool_acq (__futex, 1, 0))
9         __lll_lock_wait
10         futex_wait
11         syscall (__NR_futex, futexp, FUTEX_WAIT, val);

```

---

As shown in the source code trace, it always returns the acquired lock. When the lock is acquired without contention, no syscall is made. Otherwise, it makes the system in the last line, switching to kernel space.

On the contrary, the spinning lock will use futex less. This is because it only does `atomic_compare_and_exchange_val_acq` in userspace and returns immediately if not acquired.

### 7.1.2 *futex*

Within the kernel, Futex (Fast user-space locking) provides the locking subsystem. This is the default and recommended synchronization API of the mutex by the Linux kernel programming guide.

Library/Application that used this with the synopsis as below:

---

```

1 #include <linux/futex.h>          /* Definition of FUTEX_* constants */
2 #include <sys/syscall.h>         /* Definition of SYS_* constants */
3 #include <unistd.h>
4
5 long syscall( SYS_futex, uint32_t *uaddr, int futex_op, uint32_t val,
6               const struct timespec *timeout, /* or: uint32_t val2 */
7               uint32_t *uaddr2, uint32_t val3);

```

---

At this point, the futex subsystem will maintain entries from different threads when accessing the shared lock, including queues of waiting threads for the lock when it needs to be blocked. And the futex access is defined by the kernel source code:

---

```

1 // kernel/futex.c
2 SYSCALL_DEFINE6(futex, u32 __user *, uaddr, int, op, u32, val,
3               const struct __kernel_timespec __user *, utime,
4               u32 __user *, uaddr2, u32, val3)
5     do_futex
6         futex_wait
7         futex_wait_queue_me

```

---

Our design does not wish to replace the underneath futex subsystem. Instead, we would like to offload the user space library path to the hardware. The example access pseudocode is as below:

---

```

1 static uint32_t hybridLock(pthread_mutex_t *theLock) {

```

```

2  if (LockAcc_lock(theLock) == sucess)
3      return;
4  else
5      pthread_mutex_lock(theLock);
6  return spin;
7  }

```

---

## 7.2 ADAPTIVE HYBRID LOCK MECHANISM

In the last subsection, we showed that only the spinning part of the hybrid lock is kept in the LockAcc, and the sleeping part maintained by the futex is unchanged.

In addition to the lock address, LockAcc must maintain the spin limit corresponding to each lock. Therefore, we implemented an adaptive tuning algorithm for our LockAcc.

The LockAcc does so by observing and reacting to two events: First, if the lock is still busy after it has reached its spin limit, it concludes that spinning that long was fruitless and reduces the spin limit correspondingly, see Listing 7.1. Second, once it has gone to sleep (line 11) if it considers that it has returned quickly, it concludes that the CPU was otherwise idle, so there is no cost to spinning longer, and it adjusts its spin limit to a higher value (line 14).

An upper bound is set to justify if the step should be taken. Using this approach, the TCA adapts to runtime conditions and converges on a spin limit that provides the best possible latency and throughput for those conditions.

Listing 7.1: Hybrid Lock Spin Limit Update.

---

```

1  if (fail_on_spinning) {
2      if (penalty - spinLimit < upperbound)
3          spinLimit += 10 * spinStep;
4      else if (spinLimit is none zero) {
5          // punish for spinning too long
6          // but keep it positive

```

```

7         spinLimit -= spinStep;
8     }
9 }

```

---

The current model can be summarized as a user space adaptive hybrid lock based on the pthread mutex library and kernel space futex subsystem as the backend. A similar philosophy from interrupt handling can be applied here. The adaptive hybrid lock access is also a small and frequent execution pattern.

Below in Figure 7.2, we show the convergence time on the spin limit to reach a stable state. It takes less than 10 ms to get the range. It bounces up and down based on the anticipated sleeping time. A better algorithm should keep the range smaller. However, we showcase that the simple adaptive mechanism works well.

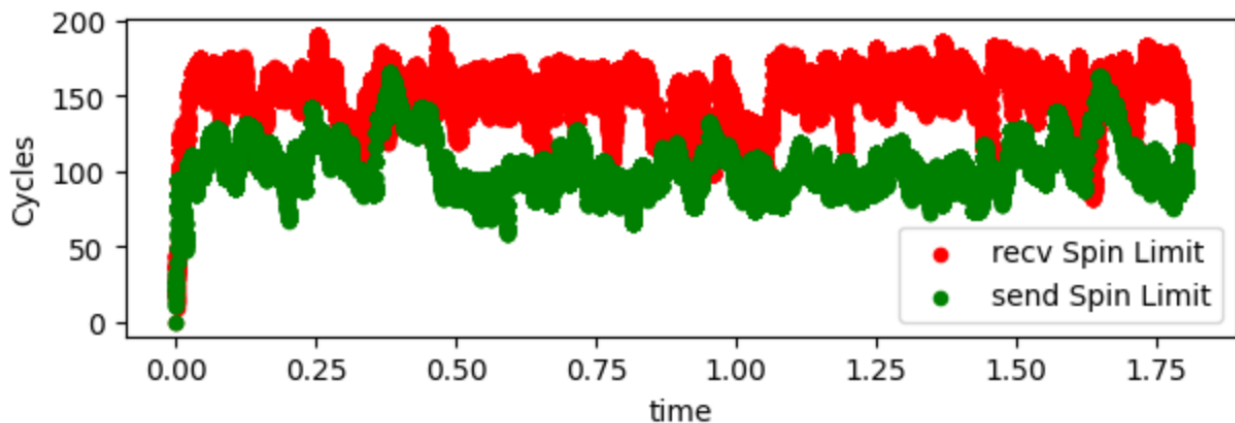


Figure 7.2: QPS=18000: The spin limit quickly move to the desired region. When the system under higher load, the recv spin limit is higher than the transmitting spin limit.

Below in Figure 7.3, we showed a slightly lower system load. The adaptive algorithm can still find the spin limit, but the range is different compared to a higher load. Under lower load, the recv spin limit does not go as high and leaves more space for send spin limit to go higher. Both of them are in the similar range.

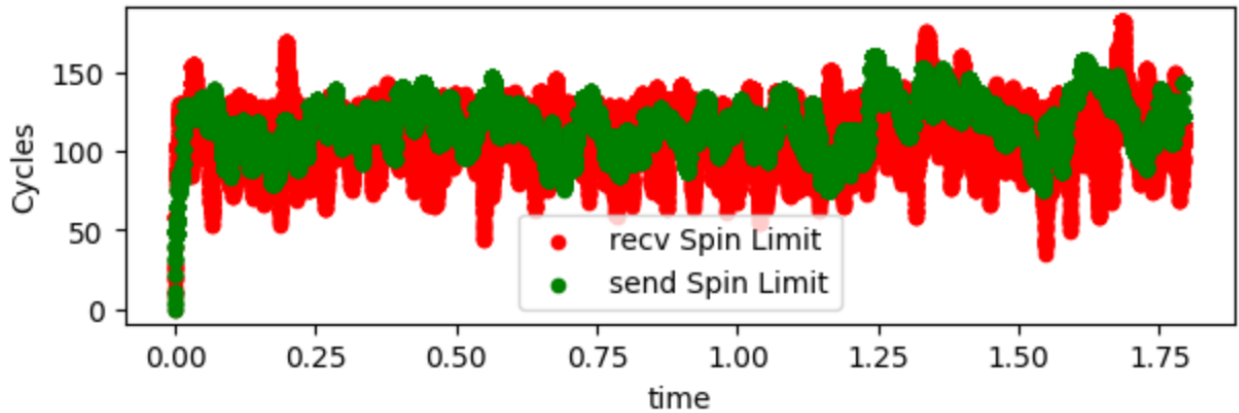


Figure 7.3: QPS=16000: The spin limit quickly move to the desired region. For lower system load, the spin limit for these locks are closer.

### 7.3 LOCKACC DESIGN

The adaptive algorithms can be applied to locks maintained in the software system library, but using a hardware accelerator can further enable energy reduction.

We propose a TCA (tightly coupled accelerator) for adaptive hybrid lock: LockAcc. Moving the user space spinning waiting into a TCA (tightly coupled accelerator) can improve latency and efficiency.

Like IRQAcc, the LockAcc only handles frequent and lightweight parts, which is the spinning waiting. There are two components of this LockAcc: adaptive spin limit and spin waiting until the limit.

The software adaptive spin limit can be implemented within the LockAcc, with the same algorithm described. The LockAcc maintains a list of spin limits and timestamps corresponding to different locks. In addition, LockAcc issues a timestamp read and evaluates spin limit adjustment when the lock fails.

Secondly, using a hardware sleep mechanism can save energy when spin waiting. For example, Intel provides instructions `uwait` and `umonitor` ([“Intel® 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4” 2023](#)). `Umonitor` instruction specifies a memory address to be

monitored, and the uwait instruction will put the CPU to lower power until the memory changes.

UMWAIT instruction takes input indicating the C-state to transition into, as shown below:

| bit      | state | wakeup time | power saving |
|----------|-------|-------------|--------------|
| bit[0]=0 | Co.2  | slower      | larger       |
| bit[0]=1 | Co.1  | faster      | smaller      |

These two states, Co.1 and Co.2, are between traditional Co (Polling) and C<sub>1</sub>, providing faster latency and energy savings. A Linux kernel patch report (*[PATCH 0/3] Sapphire Rapids Co.x Idle States Support - Artem Bityutskiy 2023*) shows 5~20% of power savings using these instructions without a noticeable latency increment. However, numbers from intel on the transition latency and specific power numbers have yet to be official.

Also, an IA32\_UMWAIT\_CONTROL MSR register controls the maximum TSC for the UMWAIT instruction. This can be used to instruct the hybrid lock spin limit. These two combined can be used to implement the hybrid spin lock, but only on intel-specific machines.

The LockAcc we proposed uses the same method, requiring access to the power management unit. Also, the LockAcc has access to the data cache within the pipeline, and the invalidate signal from memory coherence means the lock is available.

Using the LockAcc, the CPU will not repeatedly execute memory access instructions as in the software solution. Instead, the CPU will be in sleep mode and only wakes up under two conditions: the spin limit is reached, or the lock is acquired. With a reasonable spin limit, the lock should be acquired successfully primarily by the LockAcc. The adaptive spin limit algorithm is what we described in [7.2](#).

With these two pieces, we can have the LockAcc designed as below:

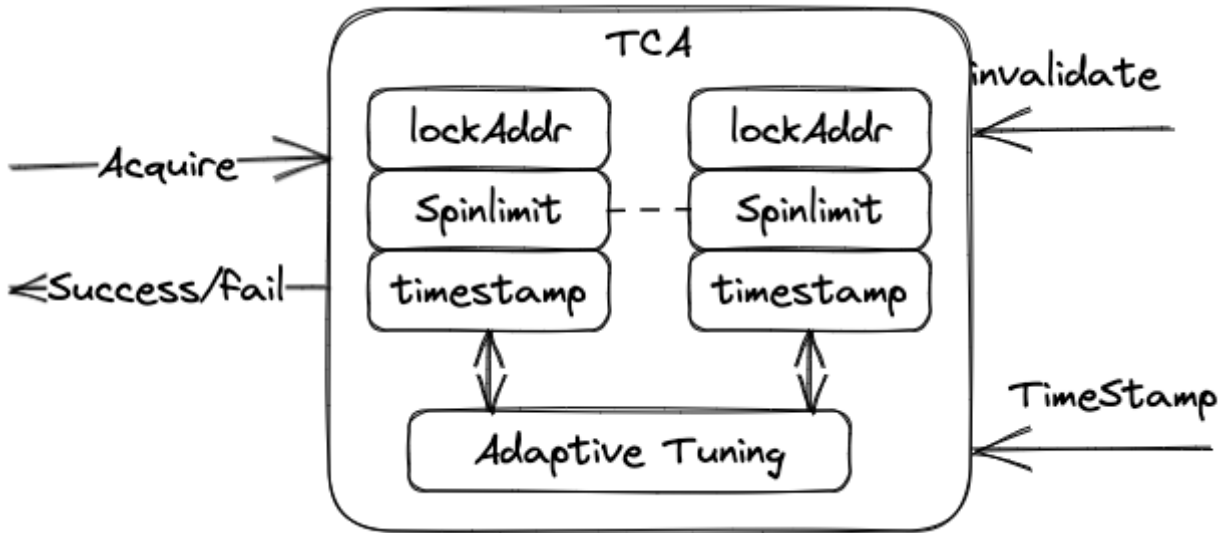


Figure 7.4: Using Accelerator maintained lock information

The LockAcc takes lock acquiring instruction from user space with a special instruction. The syntax is below:

---

```
1 tca_lock(theLock);
```

---

The LockAcc maintains copies of the lock (address and spin limit pair) and combinational logic for tuning the adaptive spin limit. LockAcc also issues memory read and write instructions through the pipeline using LSQ (Load Store Queue).

The LockAcc returns success when the lock is acquired within the spin limit or fails, indicating that the user should call default `pthread_mutex_lock` for the sleeping lock.

The LockAcc does not try to capture all the lock accesses and keep track of their spin limit. Instead, it takes an approach similar to a branch predictor with limited entries. For the expected workload, the specific lock is frequently used for the active region of a server application. Different TCA on the other core but accessing the same lock maintains different spin limits. Since the spin limit is a range of acceptable numbers and is altered adaptively, there are no benefits to sharing between cores.

The LockAcc maintains a first-in, first-out style (FIFO) policy of locks. When the newer application starts running, and all entries are used, the oldest one will be replaced. In the workload we studied, only two locks are used at any given moment for receiving and transmitting. In rare cases, the application may have actively used more locks than the entries provided, but this is beyond our design scope and thus not considered.

There is no architected state maintained across invocations, so LockAcc need not be virtualized or saved and restored across context switches, though optional support for doing so may provide performance benefits. Exploration of such opportunities is left to future work.

## 7.4 SUMMARY

In this chapter, we looked into the synchronization mechanism of adaptive hybrid locks. First, we described the existing solution of the POSIX pthread mutex library and futex system call within the Linux kernel. Next, we discussed the adaptive algorithm for searching for a better spin limit. Then, we proposed a TCA that can be used for offloading the spin waiting for part of the hybrid lock for better energy efficiency.

## 8 EVALUATION

In the previous two chapters, we described two Tightly Coupled Accelerators (TCAs): NIC interrupt handling (IRQAcc) and adaptive hybrid lock (LockAcc). This chapter will evaluate both designs separately for area cost and performance benefits. Also, we highlight energy efficiencies from these accelerators. This chapter follows:

1. Chapter 8.1 introduces the simulator and environment setup for evaluation.
2. Chapter 8.2 shows the performance benefits of using IRQAcc. There are no performance benefits from using LockAcc.
3. Chapter 8.3 shows the area cost and energy benefits of using IRQAcc and LockAcc.

### 8.1 METHODOLOGY

For Interrupt Handling Accelerator (IRQAcc), we implemented and evaluated it in the gem5 simulator. Gem5 (Nathan Binkert et al. 2011; Lowe-Power et al. 2020) is an event-driven simulator created based on the merge of two different simulators, m5(N.L. Binkert et al. 2006) for CPU simulation and GEMS(Martin et al. 2005) for memory. Gem5 provides full system simulation with a fully customized kernel and disk image.

We implemented the interrupt handling accelerator in both atomic CPU and in order CPU within the gem5 simulator. For both cases, accelerator probes interrupt signals sent to the CPU pipeline. Below, we show the example interaction between TCA and CPU pipeline.

---

```

1 // gem5/src/cpu/simple/base.cc
2 checkForInterrupts() {
3     ...
4     if (interrupt && tca.enabled) {
5         int tcaRet = thread->getCpuPtr()->tcaProcess();
6         if (tcaRet == success)
7             return
8             //default path
9             ...
10        interrupt->invoke(tc);
11        ...
12    }
13 }

```

---

Whenever TCA sees the NIC IRQ line raised by the irq chip, it follows the FSM shown in Figure 6.3. It checks the condition during each step and processes it to the next step, returning when it fails or early determination. Atomic CPU has no cache modeled, and all memory access is atomic within TCA, the same as other simulation components. TCA will send and receive memory access through the cache port in the in-order CPU. The ARM ISA is used for our evaluation since it currently has better community support with kernel binary and disk image generation. Kernel 5.15 and Ubuntu 22.04 is used for simulation, the same as our native machine setup.

|        | Value        |
|--------|--------------|
| ISA    | ARM          |
| Kernel | 5.15         |
| OS     | Ubuntu 22.04 |
| Memory | 8GB          |
| NIC    | E1000        |

As we covered earlier, different ISA and devices share the exact bottlenecks and can benefit from using these accelerators. We also implemented the TCA in TimingCPU and the InOrderCPU model. But the memory hierarchy does not perform similarly to native

machine results. We suspect that the memory model does not represent our intel server system. Moreover, we need more than cache and branch prediction access. Instead, we want to focus on the interaction between interrupt handling and server thread timing interactions. Thus, for the rest of our experiments, only AtomicCPU is used.

There is no simulation implementation of the LockAcc since using the TCA for adaptive lock does not improve the latency for accessing the lock. Instead, the advantages we focus on are energy cost benefits.

## 8.2 PERFORMANCE EVALUATION

In this section, we evaluate the performance benefits of IRQAcc. In Chapter 3, we described the workload used for our study and the workflow for microbenchmark generation. Thus, in this section, we use the generation microbenchmark for silo in our evaluation since it best suits the microservices region with sub-millisecond service time.

### Single-Threaded Performance

First, we start with a minimum setup with a single-threaded performance evaluation. Then, similar to the native setup, we have the list of notations:

1. MIX: server application and NIC IRQ are handled by the same core(s), non-threaded NAPI.
2. SEP: server application and NIC IRQ are handled by different cores(s), non-threaded NAPI.
3. TNAPI: server application and NIC IRQ are handled by the same core(s), with threaded NAPI, NAPI thread also handled by the same core(s).
4. In addition, TCA: same as TNAPI case, but hardIRQ is handled by TCA.

In Figure 8.1, we compared mean latency and tail latency (95 percentile and 99 percentile). The simulated results are very close to what we saw in the native machine,

comparing results to ???. Like native hardware, using the newer threaded NAPI (TNAPI) is better than MIX after QPS is 32000. TNAPI and SEP are close until around QPS is 38000 in mean latency. Tail latency is higher for TNAPI since server applications are more likely to be interrupted when the system is busier. However, using TNAPI within the same core allows throughput saturation to become QPS of 40000, from QPS of 39000, if SLA is set to be 2.5 at the 95 percentile.

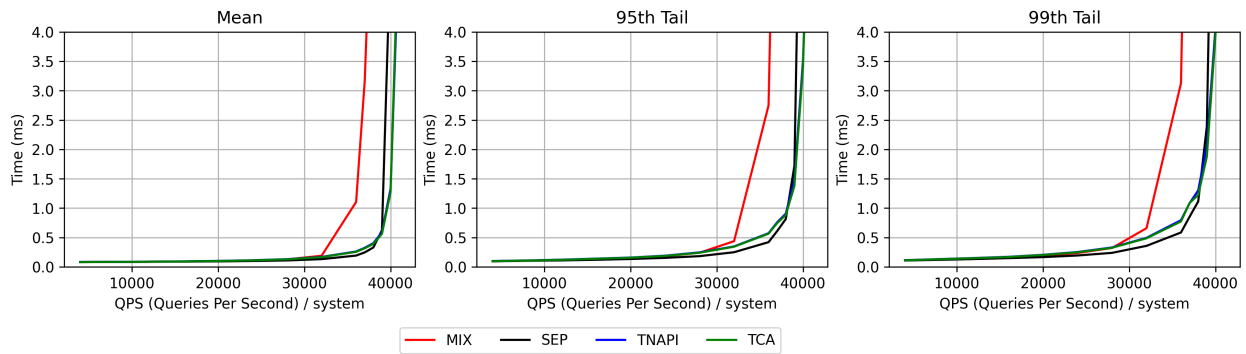


Figure 8.1: Single Thread Performance: Similar to native hardware, TNAPI performances better than MIX and SEP.

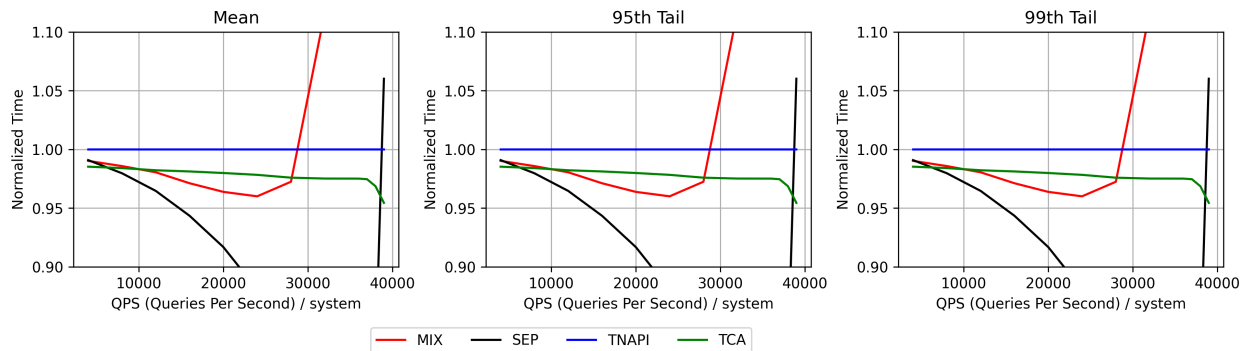


Figure 8.2: Normalized versions shows how the difference between different configuration more closely, and TCA is always better than TNAPI.

A closer comparison of normalized performance is shown in Figure 8.2. SEP performances are better at lower loads until the very end of the saturation point. MIX is better than TNAPI during low to medium loads since the networking stacks respond faster and do not hurt server applications much. However, the benefits of smaller timing

disturbances showed up when the system load became higher. Furthermore, using TCA, there is no timing disturbance. Therefore, it always performs better than TNAPI.

Below, Figure 8.3 evaluates the service time only, excluding time spent on waiting, network latency, etc. These show the partial causes of round-trip latency. For example, the MIX service time was increased when more interrupts needed to be handled within the asynchronous timing disturbance region. Also, the service time between SEP and TNAPI shows that the data movement between different cores is essential when the system becomes busier, and networking in other cores causes inter-processor communication can be an issue.

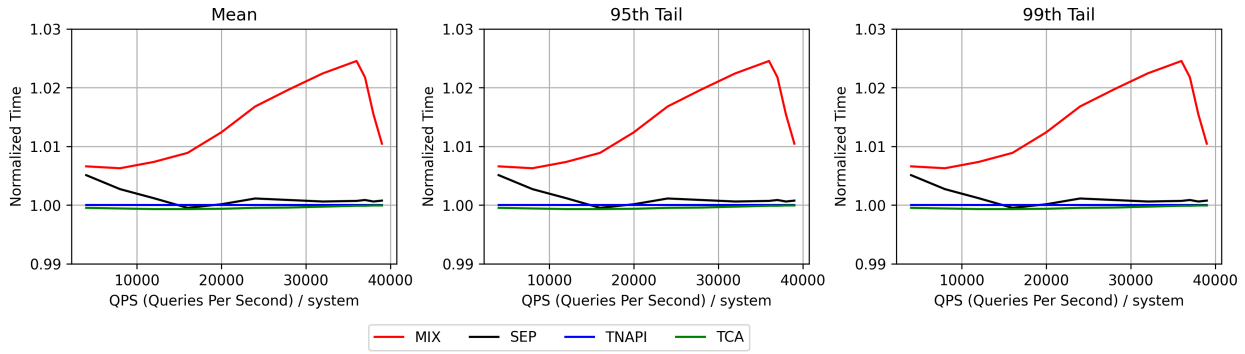


Figure 8.3: Normalized versions shows how the difference between different configuration more closely, and TCA is always better than TNAPI.

### 8.2.1 Multithreaded Performance

Then, we evaluate with a similar setup, but now with 8 threads running on a 4-core simulated environment. Below in Figure 8.4, we experiment with different ranges of RX spin limit and TX spin limit. This shows that there is a smaller range of RX with acceptable performance. Also, using TCA gives better performance when the RX spin limit is not the best, comparing the RX spin limit of 50 to 100.

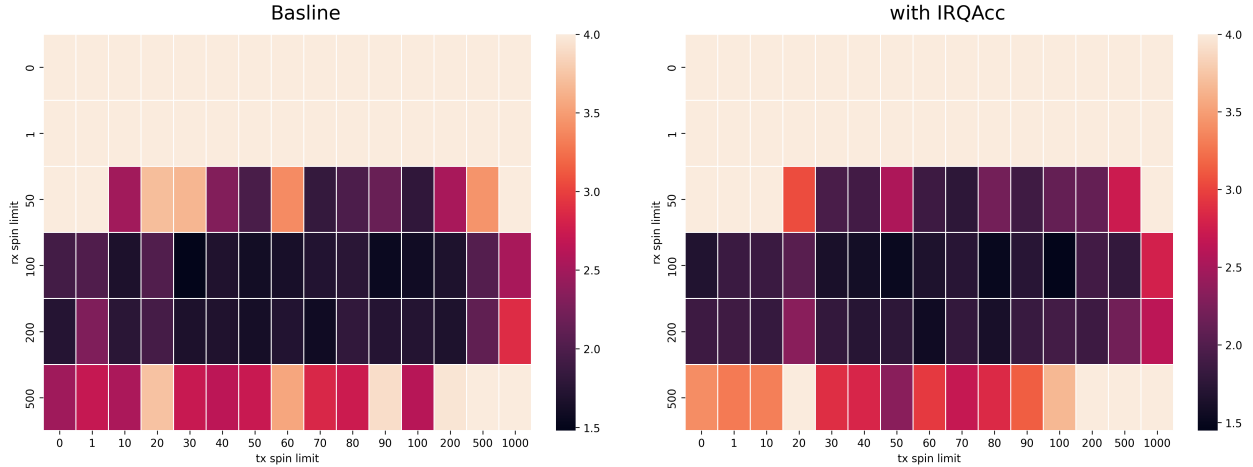


Figure 8.4: Similar to native hardware, the application is more sensitive to the RX spin limit than the TX spin limit. Darker color means better performance.

If we keep both TX and RX with the same spin limit, the results of different system loads are shown in ???. Using IRQAcc can help to reduce 99 percentile latency when the spin limit is not optimal.

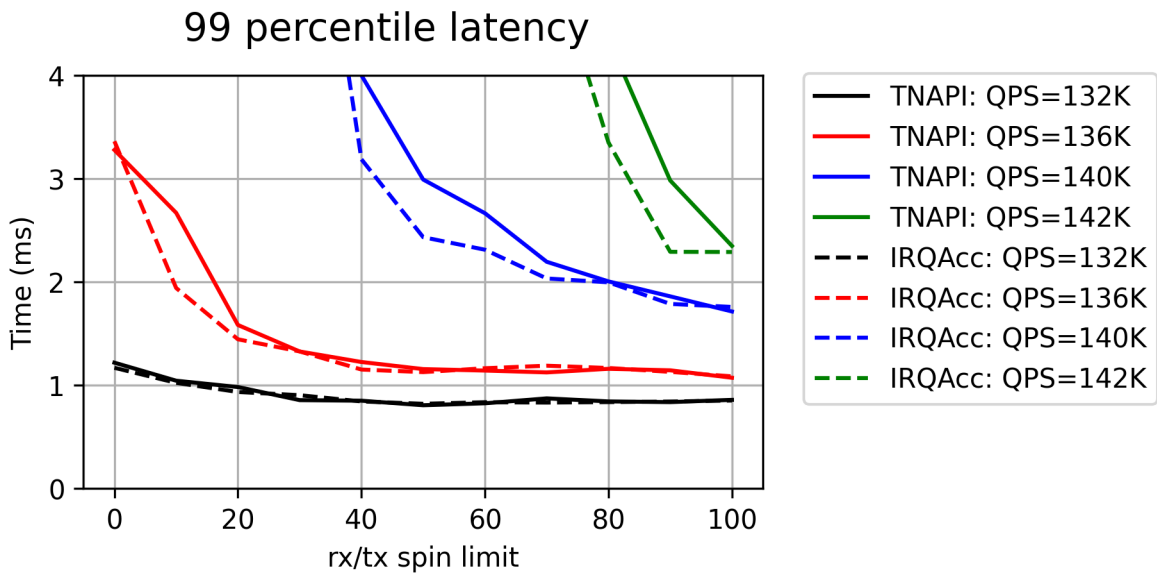


Figure 8.5: Using IRQAcc improves tail latency when spin limit is not optimized.

Then, we examine how much of the cascading effect can be reduced using TCA for IRQ. For any given query, what would be the latency if it failed to acquire the lock with the spin limit? We keep track of the timestamp before and after acquiring the lock.

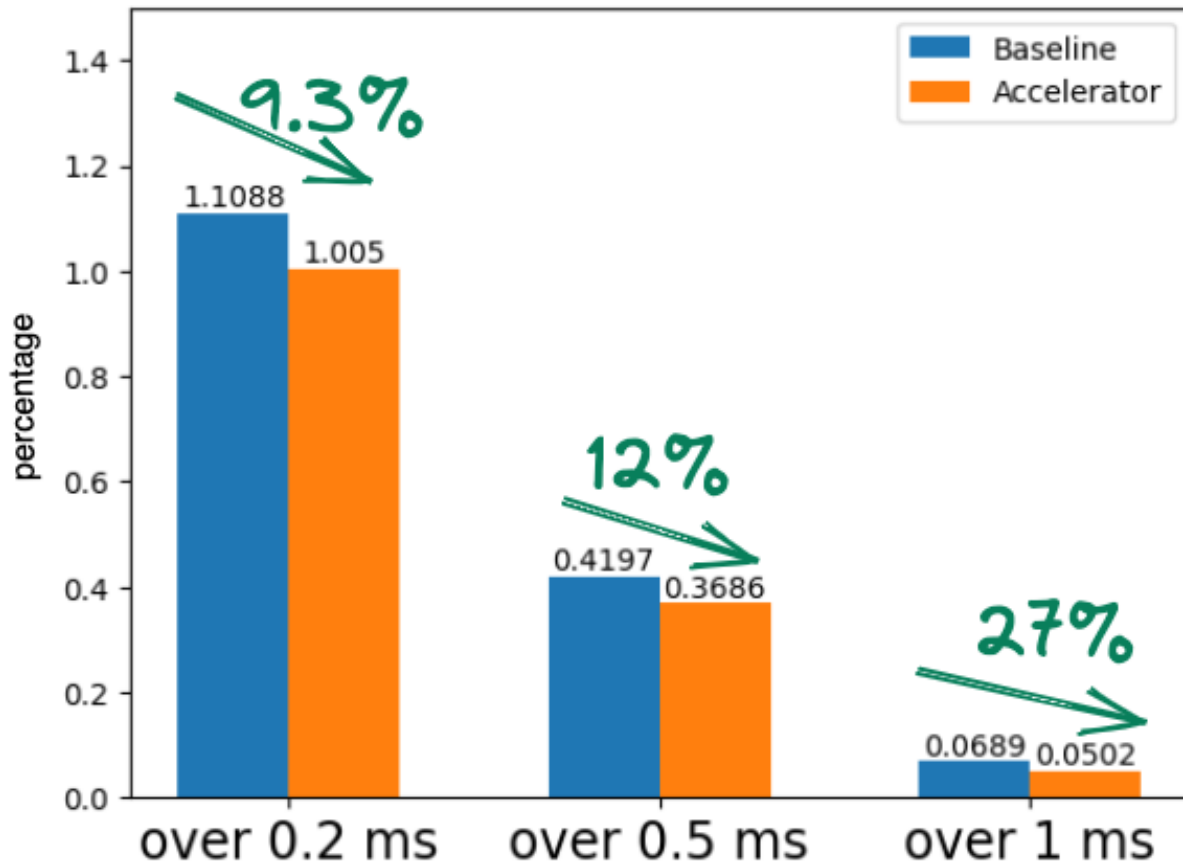


Figure 8.6: TCA helps reduce long latency on failure to acquire the lock.

In Figure 8.6, we showed the percentages of requests with lock acquiring latency longer than 0.2 ms, 0.5 ms, and 1 ms when RX spin limit and TX spin limit are the same. For fail latency longer than 1 ms, 27% of them are reduced using TCA. Similarly, 9.3% reduction for over 0.2 ms and 12% for over 0.5 ms.

### 8.3 AREA AND ENERGY EVALUATION

One of the benefits of using TCA is energy efficiency due to highly optimized hardware for specific small but frequently executed code. For IRQAcc, during our implementation, contains 31 steps (each step is one instruction) and uses 20 registers for addresses and

data for each core. Similarly, LockAcc has a minimum hardware requirement. Each lock it keeps requires 3 registers (address, spin limit, timestamp). Both IRQAcc and LockAcc then require ALU and combinational logic for calculation, FSM, and tuning mechanism.

Furthermore, using TCA for interrupt handling and hybrid lock access reduces energy savings for the CPU. Here, we are not introducing a new energy model for using these TCAs. Instead, we calculate how much work can be offloaded from the CPU to TCA, and gives the upper limit of CPU cycles that can be saved.

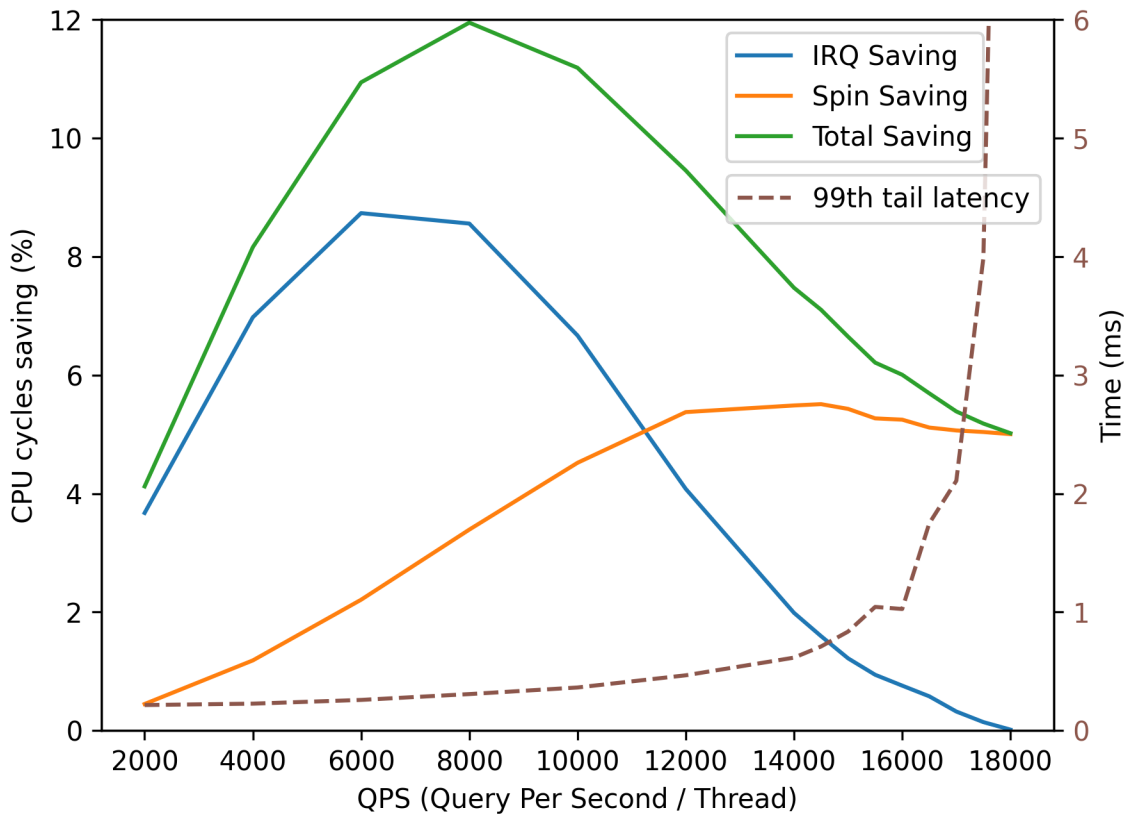


Figure 8.7: Energy Savings on using TCA: during medium to low load, frequent interrupts consume more CPU cycles. For a four-core, 8 threads setup, using both TCAs can save up to 12% CPU cycles on CPUo.

Figure 8.7 shows the potential energy savings for using TCAs. We are running the generated silo with 8 threads on a 4 cores system. Only CPUo serves interrupts since it yields the best performance. From low to medium load, NAPI is often switching between interrupts and polling mode. During this region, having the CPU serve hardIRQ does not

induce much latency to server query, but the redundant Linux kernel stack work can add up to energy consumption. Here, the energy savings from interrupt handling can be as high as over 8.7% of CPUo utilization for interrupt handling.

On the contrary, using TCA for hybrid lock access saves energy more when the system is busier and can be as high as 5%. During higher load, there is a higher chance of resource contention. Spin waiting on acquiring a lock is good for performance but consumes CPU power at the same time.

When the system is under high load (80~90%), the interrupts are rare, leaving fewer opportunities for energy savings. Overall, we can save up to 12% by using both TCAs.

## 8.4 SUMMARY

In this chapter, we described the implementation in gem5 for the NIC interrupt handling tightly coupled accelerator, IRQAcc. We then showed the performance benefits of using IRQAcc for a single thread. Furthermore, we showed that IRQAcc could relax the critical section spin limit more, reducing the longer latency tail. Finally, using IRQAcc and LockAcc, two main small and frequency tasks can be offloaded to hardware, improving energy efficiency.

## 9 CONCLUSIONS AND FUTURE DIRECTIONS

Today, data centers and server-scale systems contain large clusters of computers that exploit extreme levels of parallelism to execute complex workloads. Essential metrics, including latency, throughput, and energy efficiency, can be affected by the server application itself and the system environment they run on. Ultimately, it needs efforts from every aspect to pursue better solutions. In this work, we looked into evaluating server systems with easier benchmarking and proposed two tightly-coupled accelerators (TCAs) for better performance and energy efficiency.

### 9.1 SUMMARY OF CONTRIBUTIONS

1. In Chapter 3, we proposed a framework for Tail Latency Workload Analysis and Generation (Zhuo and M. Lipasti 2023). In this work, we dived into a detailed breakdown of an existing benchmark suite, abstracting service behavior into timing components. Then, using these parameterized timing components, a microbenchmark can be generated. These microbenchmarks can replicate the timing behaviors without complicated runtime environment requirements, allowing early-stage exploration of hardware or software improvements.
2. Chapter 4 looked into how NIC interrupt handling can impact server systems. Furthermore, we evaluated one of the recent major kernel feature updates on the networking stack: threaded NAPI. In Chapter 5, we also showed that the network interrupts and synchronization mechanisms could have cascading effects on multi-threaded server applications.

3. We proposed the first hardware accelerators in Chapter 6 based on the workload and intuitions found. IRQAcc offloads hardware interrupt handling and reduces CPU energy consumption while eliminating a substantial fraction of cascading tail latency effects.
4. We further study the impact of hybrid lock implementation on tail latency and propose a second novel TCA in Chapter 7 that adaptively chooses a hybrid lock spin limit that maximizes throughput and minimizes latency while providing further energy savings by offloading spinning activity from the CPU.

## 9.2 LIMITATIONS AND FUTURE DIRECTIONS

In this thesis, we have looked into workload generation and proposed two TCA for better tail latency-sensitive programs. However, there are assumptions we made to shape our focus better, which leads to limitations on application. There are directions for extending beyond what we have achieved in this thesis.

### 9.2.1 *TailWAG*

For TailWAG, we collected all experimental data from a single server machine. For this machine, we disabled simultaneous multithreading (hyper-threading in Intel), so only one logical core per physical core. We only used one node of 6 CPUs and disabled NUMA communication. We have fixed frequency and disabled deep sleep state. This selected set of environmental setups is standard in research for reducing noise and gives the best repeatable experiment.

On the other hand, there are also limitations of space we have explored. For our experiment, we only scale up to 4 threads on our data collection. This is due to the limited number of 6 cores per node. Any threads number beyond 6 will either requires

NUMA communications, hyper-threading, or oversharing hardware resources. One of the extensions is including these impacts in the workload analysis and generation. Still, we need to limit the complexity that this brings.

Furthermore, one limitation of the current microbenchmark is the scale of the target system. A user thread corresponds to any microbenchmark thread since we run it with native hardware. This becomes an issue when the target system scales up to tens or even hundreds of cores, and having the same number of threads running for TailWAG is unreasonable. One potential solution will be a hierarchical microbenchmark. Starting from the top level, each one breaks into different sub-components, is executed, and returns to the top recursively. It should allow only partial computation in each layer to reduce time and complexity. However, this requires sophisticated algorithms and models to ensure accuracy.

Future directions for this work include adding and evaluating additional parameters that may lead to better matches to application behavior and automating the parameter search process using machine learning techniques. One of the potential additional parameters is QPS-dependent variance, leading to more performance hiccups when the system is at a higher load: longer latency for higher load, higher variance for higher load, and a higher chance of noise for higher load. Moreover, in addition to the current log distributions, more distributions can be added for generating all latency-related randomization, including uniform and geometric distributions. Shared resource access is limited to two locks (recv and sent) in this paper. In more complicated and detailed cases, lock contention within the application may need to be adequately captured. Different threads are not always accessing the same critical section, and an array of locks could represent various shared resources. Both the length of the lock array and the probability of accessing the same lock can be tuned.

### 9.2.2 *Tightly Coupled Accelerators*

In this work, we also made assumptions about the system we are evaluating and the hardware accelerator as the solution. The same task can be applied using pure software techniques, such as a customized assembly code block in the kernel. The question is: will there be any difference if everything can be achieved in software?

Well, the current kernel is a pure software solution with many disadvantages, and we proposed to optimize it by offloading work onto specialized hardware. The benefits are limited even if we can implement the accelerator idea in pure software. For the LockAcc, the adaptive algorithm can be implemented in software, which is how we evaluated its effectiveness. However, the significant advantages of energy efficiency from hardware solutions are not applicable.

Hardware solution itself is also limited. It can provide a TCA that works for the existing CPU chip, NIC driver, interrupt chip, and kernel running on top of it. The current design of TCA uses push and pop instructions for acceleration. These instructions are handwritten at the moment. During future development, a better approach of Domain Specific Language (DSL) can be used to auto-generate compliant building blocks for accelerators. This allows better portability of TCAs to be used in any hardware-software combinations.

Furthermore, one assumption we made is the NAPI used in NIC, and our optimization is specifically on offloading the synchronous timing disturbance of hardware interrupt handling. The first network was connected over 50 years ago, in 1969, and NAPI is a mixture of interrupting and polling introduced in 2001. The most recent update is threaded NAPI, introduced in 2021, which offloads polling into a user thread. Still, the asynchronous interrupt is unchanged and unlikely to change in the foreseeable future since the components and efforts involved in this interface. Our solution can be applied to future Linux kernels based on the development trend.

In addition, our design is a tightly coupled accelerator embedded into the CPU pipeline, and interaction with the CPU is different. For IRQAcc, it does not preempt the current task. IRQAcc probes interrupt signals and only handles specific interrupt numbers. Default IRQ logic will still be used if TCA fails to handle the interrupt. However, the LockAcc does halt CPU execution since it puts the CPU into a lower power state until the lock is acquired or the spin limit is hit. One extension to explore would be allowing out-of-order execution of instructions after LockAcc.

Also, we tested with a limited number of applications from Tailbench and TailWAG-generated work. Exploring a broader set of workloads can provide a more detailed micro-architectural characterization of the potential benefits of offloading these activities from the CPU.

## 10 APPENDIX

### 10.1 KERNEL SOURCE CODE

#### 10.1.1 *Interrupt Handling Layer*

---

```

1 arch/x86/entry/entry_32.S
2 /**
3  * idtentry - Macro to generate entry stubs for simple IDT entries
4  * @vector:      Vector number
5  * @asmsym:      ASM symbol for the entry point
6  * @cfunc:      C function to be called
7  * @has_error_code: Hardware pushed error code on stack
8  */
9 .macro idtentry vector asmsym cfunc has_error_code:req
10 SYM_CODE_START(\asmsym)
11     ASM_CLAC
12     cld
13
14     .if \has_error_code == 0
15         pushl    $0        /* Clear the error code */
16     .endif
17
18     /* Push the C-function address into the GS slot */
19     pushl    $\cfunc
20     /* Invoke the common exception entry */
21     jmp handle_exception
22 SYM_CODE_END(\asmsym)
23 .endm

```

---



---

```

1 arch/x86/include/asm/idtentry.h

```

```

2  /**
3   * DEFINE_IDTENTRY_IRQ - Emit code for device interrupt IDT entry points
4   * @func:   Function name of the entry point
5   *
6   * The vector number is pushed by the low level entry stub and handed
7   * to the function as error_code argument which needs to be truncated
8   * to an u8 because the push is sign extending.
9   *
10  * irq_enter/exit_rcu() are invoked before the function body and the
11  * KVM L1D flush request is set. Stack switching to the interrupt stack
12  * has to be done in the function body if necessary.
13  */
14  #define DEFINE_IDTENTRY_IRQ(func)
15  static void __##func(struct pt_regs *regs, u32 vector);
16
17  __visible noinstr void func(struct pt_regs *regs,
18                             unsigned long error_code)
19  {
20      irqentry_state_t state = irqentry_enter(regs);
21      u32 vector = (u32)(u8)error_code;
22
23      instrumentation_begin();
24      kvm_set_cpu_lltf_flush_lld();
25      run_irq_on_irqstack_cond(__##func, regs, vector);
26      instrumentation_end();
27      irqentry_exit(regs, state);
28  }
29  static noinline void __##func(struct pt_regs *regs, u32 vector)

```

---

```

1  arch/x86/kernel/irq.c
2  /**
3   * common_interrupt() handles all normal device IRQ's (the special SMP
4   * cross-CPU interrupts have their own entry points).
5   */
6  DEFINE_IDTENTRY_IRQ(common_interrupt)
7  {
8      struct pt_regs *old_regs = set_irq_regs(regs);
9      struct irq_desc *desc;
10
11      /* entry code tells RCU that we're not quiescent. Check it. */
12      RCU_LOCKDEP_WARN(!rcu_is_watching(), "IRQ failed to wake up RCU");
13
14      desc = __this_cpu_read(vector_irq[vector]);
15      if (likely(!IS_ERR_OR_NULL(desc))) {

```

```

16     handle_irq(desc, regs);
17 } else {
18     ack_APIC_irq();
19
20     if (desc == VECTOR_UNUSED) {
21         pr_emerg_ratelimited("%s: %d.%u No irq handler for vector\n",
22                             __func__, smp_processor_id(),
23                             vector);
24     } else {
25         __this_cpu_write(vector_irq[vector], VECTOR_UNUSED);
26     }
27 }
28 set_irq_regs(old_regs);
29 }

```

---

Then, from edge level to locate specific action.

---

```

1 arch/x86/kernel/irq.c
2 static __always_inline void handle_irq(struct irq_desc *desc,
3                                         struct pt_regs *regs)
4 {
5     if (IS_ENABLED(CONFIG_X86_64))
6         generic_handle_irq_desc(desc);
7     else
8         __handle_irq(desc, regs);
9 }

```

---



---

```

1 include/linux/irqdesc.h
2 /*
3  * Architectures call this to let the generic IRQ layer
4  * handle an interrupt.
5  */
6 static inline void generic_handle_irq_desc(struct irq_desc *desc)
7 {
8     desc->handle_irq(desc);
9 }
10
11
12 ````c++
13 include/linux/irq.h

```

```

14  /*
15   * Built-in IRQ handlers for various IRQ types,
16   * callable via desc->handle_irq()
17   */
18  extern void handle_level_irq(struct irq_desc *desc);
19  extern void handle_fasteoi_irq(struct irq_desc *desc);
20  extern void handle_edge_irq(struct irq_desc *desc);
21  extern void handle_edge_eoi_irq(struct irq_desc *desc);
22  extern void handle_simple_irq(struct irq_desc *desc);
23  extern void handle_untracked_irq(struct irq_desc *desc);
24  extern void handle_percpu_irq(struct irq_desc *desc);
25  extern void handle_percpu_devid_irq(struct irq_desc *desc);
26  extern void handle_bad_irq(struct irq_desc *desc);
27  extern void handle_nested_irq(unsigned int irq);

```

---

Note, by this step, the CPU does not know which interrupt to handle specifically, it only cares and knows it is a common\_interrupt entry point. The CPU switches from user context to hardware interrupt (hardIRQ) during this step. Then, the next layer defines different types of interrupt entry points. As far as we know, I/O devices trigger either level or edge interrupt, both x86 (manual section 10.1, page 363) and ARM (<https://developer.arm.com/documentation/duio552/a/Bhchgeei>) specifications indicated.

### *generic irq*

---

```

1  kernel/irq/chip.c
2  /**
3   *  handle_edge_irq - edge type IRQ handler
4   *  @desc:  the interrupt description structure for this irq
5   *
6   *  Interrupt occurs on the falling and/or rising edge of a hardware
7   *  signal. The occurrence is latched into the irq controller hardware
8   *  and must be acked in order to be reenabled. After the ack another
9   *  interrupt can happen on the same source even before the first one
10  *  is handled by the associated event handler. If this happens it
11  *  might be necessary to disable (mask) the interrupt depending on the
12  *  controller hardware. This requires to reenale the interrupt inside
13  *  of the loop which handles the interrupts which have arrived while

```

```

14  * the handler was running. If all pending interrupts are handled, the
15  * loop is left.
16  */
17  void handle_edge_irq(struct irq_desc *desc)
18  {
19      raw_spin_lock(&desc->lock);
20
21      desc->istate &= ~(IRQS_REPLAY | IRQS_WAITING);
22
23      if (!irq_may_run(desc)) {
24          desc->istate |= IRQS_PENDING;
25          mask_ack_irq(desc);
26          goto out_unlock;
27      }
28
29      /*
30       * If its disabled or no action available then mask it and get
31       * out of here.
32       */
33      if (irqd_irq_disabled(&desc->irq_data) || !desc->action) {
34          desc->istate |= IRQS_PENDING;
35          mask_ack_irq(desc);
36          goto out_unlock;
37      }
38
39      kstat_incr_irqs_this_cpu(desc);
40
41      /* Start handling the irq */
42      desc->irq_data.chip->irq_ack(&desc->irq_data);
43
44      do {
45          if (unlikely(!desc->action)) {
46              mask_irq(desc);
47              goto out_unlock;
48          }
49
50          /*
51           * When another irq arrived while we were handling
52           * one, we could have masked the irq.
53           * Reenable it, if it was not disabled in meantime.
54           */
55          if (unlikely(desc->istate & IRQS_PENDING)) {
56              if (!irqd_irq_disabled(&desc->irq_data) &&
57                  irqd_irq_masked(&desc->irq_data))
58                  unmask_irq(desc);
59          }
60

```

```

61         handle_irq_event(desc);
62
63     } while ((desc->istate & IRQS_PENDING) &&
64             !irqd_irq_disabled(&desc->irq_data));
65
66 out_unlock:
67     raw_spin_unlock(&desc->lock);
68 }
69 EXPORT_SYMBOL(handle_edge_irq);

```

---



---

```

1 kernel/irq/handle.c
2 irqreturn_t handle_irq_event(struct irq_desc *desc)
3 {
4     irqreturn_t ret;
5
6     desc->istate &= ~IRQS_PENDING;
7     irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
8     raw_spin_unlock(&desc->lock);
9
10    ret = handle_irq_event_percpu(desc);
11
12    raw_spin_lock(&desc->lock);
13    irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
14    return ret;
15 }
16
17 irqreturn_t handle_irq_event_percpu(struct irq_desc *desc)
18 {
19     irqreturn_t retval;
20     unsigned int flags = 0;
21
22     retval = __handle_irq_event_percpu(desc, &flags);
23
24     add_interrupt_randomness(desc->irq_data.irq, flags);
25
26     if (!irq_settings_no_debug(desc))
27         note_interrupt(desc, retval);
28     return retval;
29 }
30
31 irqreturn_t __handle_irq_event_percpu(struct irq_desc *desc, unsigned int *flags)
32 {
33     irqreturn_t retval = IRQ_NONE;
34     unsigned int irq = desc->irq_data.irq;

```

```

35     struct irqaction *action;
36
37     record_irq_time(desc);
38
39     for_each_action_of_desc(desc, action) {
40         irqreturn_t res;
41
42         /*
43          * If this IRQ would be threaded under force_irqthreads, mark it so.
44          */
45         if (irq_settings_can_thread(desc) &&
46             !(action->flags & (IRQF_NO_THREAD | IRQF_PERCPU | IRQF_ONESHOT)))
47             lockdep_hardirq_threaded();
48
49         trace_irq_handler_entry(irq, action);
50         res = action->handler(irq, action->dev_id);
51         trace_irq_handler_exit(irq, action, res);
52
53         if (WARN_ONCE(!irqs_disabled(), "irq %u handler %pS enabled interrupts\n",
54                     irq, action->handler))
55             local_irq_disable();
56
57         switch (res) {
58             case IRQ_WAKE_THREAD:
59                 /*
60                  * Catch drivers which return WAKE_THREAD but
61                  * did not set up a thread function
62                  */
63                 if (unlikely(!action->thread_fn)) {
64                     warn_no_thread(irq, action);
65                     break;
66                 }
67
68                 __irq_wake_thread(desc, action);
69
70                 fallthrough;    /* to add to randomness */
71             case IRQ_HANDLED:
72                 *flags |= action->flags;
73                 break;
74
75             default:
76                 break;
77         }
78
79         retval |= res;
80     }
81

```

```

82     return retval;
83 }

```

---

### 10.1.2 NIC Driver

---

```

1  /**
2   * e1000_intr - Interrupt Handler
3   * @irq: interrupt number
4   * @data: pointer to a network interface device structure
5   */
6  static irqreturn_t e1000_intr(int irq, void *data)
7  {
8      struct net_device *netdev = data;
9      struct e1000_adapter *adapter = netdev_priv(netdev);
10     struct e1000_hw *hw = &adapter->hw;
11     u32 icr = er32(ICR);
12
13     if (unlikely(!icr))
14         return IRQ_NONE; /* Not our interrupt */
15
16     /* we might have caused the interrupt, but the above
17      * read cleared it, and just in case the driver is
18      * down there is nothing to do so return handled
19      */
20     if (unlikely(test_bit(__E1000_DOWN, &adapter->flags)))
21         return IRQ_HANDLED;
22
23     if (unlikely(icr & (E1000_ICR_RXSEQ | E1000_ICR_LSC))) {
24         hw->get_link_status = 1;
25         /* guard against interrupt when we're going down */
26         if (!test_bit(__E1000_DOWN, &adapter->flags))
27             schedule_delayed_work(&adapter->watchdog_task, 1);
28     }
29
30     /* disable interrupts, without the synchronize_irq bit */
31     ew32(IMC, ~0);
32     E1000_WRITE_FLUSH();
33
34     if (likely(napi_schedule_prep(&adapter->napi))) {
35         adapter->total_tx_bytes = 0;

```

```

36     adapter->total_tx_packets = 0;
37     adapter->total_rx_bytes = 0;
38     adapter->total_rx_packets = 0;
39     __napi_schedule(&adapter->napi);
40 } else {
41     /* this really should not happen! if it does it is basically a
42      * bug, but not a hard error, so enable ints and continue
43      */
44     if (!test_bit(__E1000_DOWN, &adapter->flags))
45         e1000_irq_enable(adapter);
46 }
47
48 return IRQ_HANDLED;
49 }

```

---

### 10.1.3 NAPI

---

```

1  include/linux/netdevice.h
2  /**
3   * napi_schedule - schedule NAPI poll
4   * @n: NAPI context
5   *
6   * Schedule NAPI poll routine to be called if it is not already
7   * running.
8   */
9  static inline void napi_schedule(struct napi_struct *n)
10 {
11     if (napi_schedule_prep(n))
12         __napi_schedule(n);
13 }

```

---



---

```

1  net/core/dev.c
2  /**
3   * __napi_schedule - schedule for receive
4   * @n: entry to schedule
5   *
6   * The entry's receive function will be scheduled to run.

```

```

7  * Consider using __napi_schedule_irqoff() if hard irqs are masked.
8  */
9  void __napi_schedule(struct napi_struct *n)
10 {
11     unsigned long flags;
12
13     local_irq_save(flags);
14     ____napi_schedule(this_cpu_ptr(&softnet_data), n);
15     local_irq_restore(flags);
16 }
17 EXPORT_SYMBOL(__napi_schedule);

```

---

```

1  /* Called with irq disabled */
2  static inline void ____napi_schedule(struct softnet_data *sd,
3                                     struct napi_struct *napi)
4  {
5      struct task_struct *thread;
6
7      if (test_bit(NAPI_STATE_THREADED, &napi->state)) {
8          /* Paired with smp_mb__before_atomic() in
9           * napi_enable()/dev_set_threaded().
10         * Use READ_ONCE() to guarantee a complete
11         * read on napi->thread. Only call
12         * wake_up_process() when it's not NULL.
13         */
14         thread = READ_ONCE(napi->thread);
15         if (thread) {
16             /* Avoid doing set_bit() if the thread is in
17              * INTERRUPTIBLE state, cause napi_thread_wait()
18              * makes sure to proceed with napi polling
19              * if the thread is explicitly woken from here.
20              */
21             if (READ_ONCE(thread->__state) != TASK_INTERRUPTIBLE)
22                 set_bit(NAPI_STATE_SCHED_THREADED, &napi->state);
23             wake_up_process(thread);
24             return;
25         }
26     }
27
28     list_add_tail(&napi->poll_list, &sd->poll_list);
29     __raise_softirq_irqoff(NET_RX_SOFTIRQ);
30 }

```

---

### 10.1.4 Generic Scheduler

---

```

1 kernel/sched/core.c
2 /**
3  * wake_up_process - Wake up a specific process
4  * @p: The process to be woken up.
5  *
6  * Attempt to wake up the nominated process and move it to the set of runnable
7  * processes.
8  *
9  * Return: 1 if the process was woken up, 0 if it was already running.
10 *
11 * This function executes a full memory barrier before accessing the task state
12 */
13 int wake_up_process(struct task_struct *p)
14 {
15     return try_to_wake_up(p, TASK_NORMAL, 0);
16 }

```

---



---

```

1
2 /**
3  * try_to_wake_up - wake up a thread
4  * @p: the thread to be awakened
5  * @state: the mask of task states that can be woken
6  * @wake_flags: wake modifier flags (WF_*)
7  *
8  * Conceptually does:
9  *
10 *   If (@state & @p->state) @p->state = TASK_RUNNING.
11 *
12 * If the task was not queued/runnable, also place it back on a runqueue.
13 *
14 * This function is atomic against schedule() which would dequeue the task.
15 *
16 * It issues a full memory barrier before accessing @p->state, see the comment
17 * with set_current_state().
18 *
19 * Uses p->pi_lock to serialize against concurrent wake-ups.
20 *
21 * Relies on p->pi_lock stabilizing:

```

```

22  * - p->sched_class
23  * - p->cpus_ptr
24  * - p->sched_task_group
25  * in order to do migration, see its use of select_task_rq()/set_task_cpu().
26  *
27  * Tries really hard to only take one task_rq(p)->lock for performance.
28  * Takes rq->lock in:
29  * - ttwu_runnable()      -- old rq, unavoidable, see comment there;
30  * - ttwu_queue()         -- new rq, for enqueue of the task;
31  * - psi_ttwu_dequeue()   -- much sadness :-( accounting will kill us.
32  *
33  * As a consequence we race really badly with just about everything. See the
34  * many memory barriers and their comments for details.
35  *
36  * Return: %true if @p->state changes (an actual wakeup was done),
37  *         %false otherwise.
38  */
39 static int
40 try_to_wake_up(struct task_struct *p, unsigned int state, int wake_flags)
41 {
42     unsigned long flags;
43     int cpu, success = 0;
44
45     preempt_disable();
46     if (p == current) {
47         /*
48          * We're waking current, this means 'p->on_rq' and 'task_cpu(p)
49          * == smp_processor_id()'. Together this means we can special
50          * case the whole 'p->on_rq && ttwu_runnable()' case below
51          * without taking any locks.
52          *
53          * In particular:
54          * - we rely on Program-Order guarantees for all the ordering,
55          * - we're serialized against set_special_state() by virtue of
56          *   it disabling IRQs (this allows not taking ->pi_lock).
57          */
58         if (!ttwu_state_match(p, state, &success))
59             goto out;
60
61         trace_sched_waking(p);
62         WRITE_ONCE(p->__state, TASK_RUNNING);
63         trace_sched_wakeup(p);
64         goto out;
65     }
66
67     /*
68      * If we are going to wake up a thread waiting for CONDITION we

```

```

69      * need to ensure that CONDITION=1 done by the caller can not be
70      * reordered with p->state check below. This pairs with smp_store_mb()
71      * in set_current_state() that the waiting thread does.
72      */
73      raw_spin_lock_irqsave(&p->pi_lock, flags);
74      smp_mb__after_spinlock();
75      if (!ttwu_state_match(p, state, &success))
76          goto unlock;
77
78      trace_sched_waking(p);
79
80      /*
81       * Ensure we load p->on_rq _after_ p->state, otherwise it would
82       * be possible to, falsely, observe p->on_rq == 0 and get stuck
83       * in smp_cond_load_acquire() below.
84       *
85       * sched_ttwu_pending()          try_to_wake_up()
86       *   STORE p->on_rq = 1          LOAD p->state
87       *   UNLOCK rq->lock
88       *
89       * __schedule() (switch to task 'p')
90       *   LOCK rq->lock              smp_rmb();
91       *   smp_mb__after_spinlock();
92       *   UNLOCK rq->lock
93       *
94       * [task p]
95       *   STORE p->state = UNINTERRUPTIBLE    LOAD p->on_rq
96       *
97       * Pairs with the LOCK+smp_mb__after_spinlock() on rq->lock in
98       * __schedule(). See the comment for smp_mb__after_spinlock().
99       *
100      * A similar smb_rmb() lives in try_invoke_on_locked_down_task().
101      */
102      smp_rmb();
103      if (READ_ONCE(p->on_rq) && ttwu_runnable(p, wake_flags))
104          goto unlock;
105
106  #ifndef CONFIG_SMP
107      /*
108       * Ensure we load p->on_cpu _after_ p->on_rq, otherwise it would be
109       * possible to, falsely, observe p->on_cpu == 0.
110       *
111       * One must be running (->on_cpu == 1) in order to remove oneself
112       * from the runqueue.
113       *
114       * __schedule() (switch to task 'p')    try_to_wake_up()
115       *   STORE p->on_cpu = 1          LOAD p->on_rq

```

```

116      *   UNLOCK rq->lock
117      *
118      * __schedule() (put 'p' to sleep)
119      *   LOCK rq->lock           smp_rmb();
120      *   smp_mb__after_spinlock();
121      *   STORE p->on_rq = 0           LOAD p->on_cpu
122      *
123      * Pairs with the LOCK+smp_mb__after_spinlock() on rq->lock in
124      * __schedule(). See the comment for smp_mb__after_spinlock().
125      *
126      * Form a control-dep-acquire with p->on_rq == 0 above, to ensure
127      * schedule()'s deactivate_task() has 'happened' and p will no longer
128      * care about it's own p->state. See the comment in __schedule().
129      */
130  smp_acquire__after_ctrl_dep();
131
132  /*
133   * We're doing the wakeup (@success == 1), they did a dequeue (p->on_rq
134   * == 0), which means we need to do an enqueue, change p->state to
135   * TASK_WAKING such that we can unlock p->pi_lock before doing the
136   * enqueue, such as ttwu_queue_wakelist().
137   */
138  WRITE_ONCE(p->__state, TASK_WAKING);
139
140  /*
141   * If the owning (remote) CPU is still in the middle of schedule() with
142   * this task as prev, considering queueing p on the remote CPUs wake_list
143   * which potentially sends an IPI instead of spinning on p->on_cpu to
144   * let the waker make forward progress. This is safe because IRQs are
145   * disabled and the IPI will deliver after on_cpu is cleared.
146   *
147   * Ensure we load task_cpu(p) after p->on_cpu:
148   *
149   * set_task_cpu(p, cpu);
150   *   STORE p->cpu = @cpu
151   * __schedule() (switch to task 'p')
152   *   LOCK rq->lock
153   *   smp_mb__after_spin_lock()           smp_cond_load_acquire(&p->on_cpu)
154   *   STORE p->on_cpu = 1           LOAD p->cpu
155   *
156   * to ensure we observe the correct CPU on which the task is currently
157   * scheduling.
158   */
159  if (smp_load_acquire(&p->on_cpu) &&
160      ttwu_queue_wakelist(p, task_cpu(p), wake_flags | WF_ON_CPU))
161      goto unlock;
162

```

```

163      /*
164      * If the owning (remote) CPU is still in the middle of schedule() with
165      * this task as prev, wait until it's done referencing the task.
166      *
167      * Pairs with the smp_store_release() in finish_task().
168      *
169      * This ensures that tasks getting woken will be fully ordered against
170      * their previous state and preserve Program Order.
171      */
172      smp_cond_load_acquire(&p->on_cpu, !VAL);
173
174      cpu = select_task_rq(p, p->wake_cpu, wake_flags | WF_TTWU);
175      if (task_cpu(p) != cpu) {
176          if (p->in_iowait) {
177              delayacct_blkio_end(p);
178              atomic_dec(&task_rq(p)->nr_iowait);
179          }
180
181          wake_flags |= WF_MIGRATED;
182          psi_ttwu_dequeue(p);
183          set_task_cpu(p, cpu);
184      }
185      #else
186          cpu = task_cpu(p);
187      #endif /* CONFIG_SMP */
188
189      ttwu_queue(p, cpu, wake_flags);
190      unlock:
191      raw_spin_unlock_irqrestore(&p->pi_lock, flags);
192      out:
193      if (success)
194          ttwu_stat(p, task_cpu(p), wake_flags);
195      preempt_enable();
196
197      return success;
198  }

```

---

```

1  static void ttwu_queue(struct task_struct *p, int cpu, int wake_flags)
2  {
3      struct rq *rq = cpu_rq(cpu);
4      struct rq_flags rf;
5
6      if (ttwu_queue_wakelist(p, cpu, wake_flags))
7          return;

```

```

8
9     rq_lock(rq, &rf);
10    update_rq_clock(rq);
11    ttwu_do_activate(rq, p, wake_flags, &rf);
12    rq_unlock(rq, &rf);
13 }

```

---

```

1
2 static void
3 ttwu_do_activate(struct rq *rq, struct task_struct *p, int wake_flags,
4                 struct rq_flags *rf)
5 {
6     int en_flags = ENQUEUE_WAKEUP | ENQUEUE_NOCLOCK;
7
8     lockdep_assert_rq_held(rq);
9
10    if (p->sched_contributes_to_load)
11        rq->nr_uninterruptible--;
12
13    #ifdef CONFIG_SMP
14        if (wake_flags & WF_MIGRATED)
15            en_flags |= ENQUEUE_MIGRATED;
16        else
17    #endif
18        if (p->in_iowait) {
19            delayacct_blkio_end(p);
20            atomic_dec(&task_rq(p)->nr_iowait);
21        }
22
23    activate_task(rq, p, en_flags);
24    ttwu_do_wakeup(rq, p, wake_flags, rf);
25 }

```

---

```

1 void activate_task(struct rq *rq, struct task_struct *p, int flags)
2 {
3     enqueue_task(rq, p, flags);
4
5     p->on_rq = TASK_ON_RQ_QUEUED;
6 }

```

---

---

```

1 static inline void enqueue_task(struct rq *rq, struct task_struct *p, int flags)
2 {
3     if (!(flags & ENQUEUE_NOCLOCK))
4         update_rq_clock(rq);
5
6     if (!(flags & ENQUEUE_RESTORE)) {
7         sched_info_enqueue(rq, p);
8         psi_enqueue(p, flags & ENQUEUE_WAKEUP);
9     }
10
11     uclamp_rq_inc(rq, p);
12     p->sched_class->enqueue_task(rq, p, flags);
13
14     if (sched_core_enabled(rq))
15         sched_core_enqueue(rq, p);
16 }

```

---



---

```

1 /*
2  * Mark the task runnable and perform wakeup-preemption.
3  */
4 static void ttwu_do_wakeup(struct rq *rq, struct task_struct *p, int wake_flags,
5                          struct rq_flags *rf)
6 {
7     check_preempt_curr(rq, p, wake_flags);
8     WRITE_ONCE(p->__state, TASK_RUNNING);
9     trace_sched_wakeup(p);
10
11 #ifdef CONFIG_SMP
12     if (p->sched_class->task_woken) {
13         /*
14          * Our task @p is fully woken up and running; so it's safe to
15          * drop the rq->lock, hereafter rq is only used for statistics.
16          */
17         rq_unpin_lock(rq, rf);
18         p->sched_class->task_woken(rq, p);
19         rq_repin_lock(rq, rf);
20     }
21
22     if (rq->idle_stamp) {
23         u64 delta = rq_clock(rq) - rq->idle_stamp;
24         u64 max = 2*rq->max_idle_balance_cost;
25
26         update_avg(&rq->avg_idle, delta);

```

```

27
28     if (rq->avg_idle > max)
29         rq->avg_idle = max;
30
31     rq->wake_stamp = jiffies;
32     rq->wake_avg_idle = rq->avg_idle / 2;
33
34     rq->idle_stamp = 0;
35 }
36 #endif
37 }

```

---

### 10.1.5 Round Robin Scheduler

---

```

1 kernel/sched/rt.c
2 /*
3  * Adding/removing a task to/from a priority array:
4  */
5 static void
6 enqueue_task_rt(struct rq *rq, struct task_struct *p, int flags)
7 {
8     struct sched_rt_entity *rt_se = &p->rt;
9
10    if (flags & ENQUEUE_WAKEUP)
11        rt_se->timeout = 0;
12
13    enqueue_rt_entity(rt_se, flags);
14
15    if (!task_current(rq, p) && p->nr_cpus_allowed > 1)
16        enqueue_pushable_task(rq, p);
17 }

```

---



---

```

1 static void enqueue_rt_entity(struct sched_rt_entity *rt_se, unsigned int flags)
2 {
3     struct rq *rq = rq_of_rt_se(rt_se);
4
5     dequeue_rt_stack(rt_se, flags);

```

```

6     for_each_sched_rt_entity(rt_se)
7         __enqueue_rt_entity(rt_se, flags);
8     enqueue_top_rt_rq(&rq->rt);
9 }

```

---

```

1 static void __enqueue_rt_entity(struct sched_rt_entity *rt_se, unsigned int flags)
2 {
3     struct rt_rq *rt_rq = rt_rq_of_se(rt_se);
4     struct rt_prio_array *array = &rt_rq->active;
5     struct rt_rq *group_rq = group_rt_rq(rt_se);
6     struct list_head *queue = array->queue + rt_se_prio(rt_se);
7
8     /*
9      * Don't enqueue the group if its throttled, or when empty.
10     * The latter is a consequence of the former when a child group
11     * get throttled and the current group doesn't have any other
12     * active members.
13     */
14     if (group_rq && (rt_rq_throttled(group_rq) || !group_rq->rt_nr_running)) {
15         if (rt_se->on_list)
16             __delist_rt_entity(rt_se, array);
17         return;
18     }
19
20     if (move_entity(flags)) {
21         WARN_ON_ONCE(rt_se->on_list);
22         if (flags & ENQUEUE_HEAD)
23             list_add(&rt_se->run_list, queue);
24         else
25             list_add_tail(&rt_se->run_list, queue);
26
27         __set_bit(rt_se_prio(rt_se), array->bitmap);
28         rt_se->on_list = 1;
29     }
30     rt_se->on_rq = 1;
31
32     inc_rt_tasks(rt_se, rt_rq);
33 }

```

---

## 10.2 HOW TO CALCULATE THE TIME

Accurate time calculation is important. There are multiple ways to measure the time.

(Paoloni 2010)

<https://stackoverflow.com/questions/12631856/difference-between-rdtscp-rdtsc-memory-and-cpuid-rdtsc>

---

```

1 static __inline__ uint64_t getStartCycle(void)
2 {
3     uint64_t high, low;
4     asm volatile("cpuid" ::: "%rax", "%rbx", "%rcx", "%rdx");
5     asm volatile("rdtsc" : "=a" (low), "=d" (high));
6     return (high << 32) | low;
7 }

```

---



---

```

1 static __inline__ uint64_t getEndCycle(void)
2 {
3     uint64_t high, low;
4     asm volatile("rdtscp" : "=a" (low), "=d" (high));
5     asm volatile("cpuid" ::: "%rax", "%rbx", "%rcx", "%rdx");
6     return (high << 32) | low;
7 }

```

---

For better effort tie measuring, the above two functions should be used:

---

```

1 begin = getStartCycle()
2 do something
3 end = getEndCycle()
4
5 time = end- begin;

```

---

cpuid instruction is used for ensuring ordering, but costly, so only used before the start of time, and after the end of time.

### 10.2.1 Resolution

*On a x86 server machine*

Assume we running on a 3.4GHz core. With the below loop, we measured the resolution of our hybrid loc:

---

```

1  ``c++
2  repeat = 100;
3  begin = getStartCycle();
4  for (int i = 0; i < repeat; i++){
5      asm volatile("mfence");
6  }
7  end = getEndCycle();
8
9  loop_cycle = (end-begin)/100;
10 loop_latency_ns = loop_cycle / 3.4;

```

---

mfence is used to slow down the CPU and increase CPI (Cycles per Instruction). Higher the repeat, the higher confidence in the result. We get loop\_cycle around 45 cycles, and loop\_latency\_ns is around 13.5 ns. So for our hybrid lock design, each cycle of spinning is 13.5 ns.

---

```

1  uint64_t begin = getStartCycle();
2  for (int i = 0; i < 1000; i++) {
3      hybridLock(&microLock, 1);
4      asm volatile("mfence");
5      pthread_mutex_unlock(&microLock);
6      asm volatile("mfence");
7  }
8  uint64_t end = getEndCycle();

```

---

Using this, the measured lock and unlock delay is around 66 ns on a 3.4GHz CPU, around 220 cycles.

Back-to-back clock access is 24 ns on a 3.4 GHz CPU, which is 80 cycles.

*On a raspberry pi*

Since it is an arm 64 bits ISA, “mfence” is not valid.

---

```

1  static uint16_t hybridLock(pthread_mutex_t *theLock, uint16_t spinLimit) {
2      uint16_t spin = 0;
3      while (spin < spinLimit) {
4          if(pthread_mutex_trylock(theLock)==0) {
5              return spin;
6          }
7      #if defined(__x86_64__) || defined(__amd64__)
8          asm volatile("mfence");
9      #elif defined(__aarch64__)
10         asm volatile("dmb sy" : : : "memory");
11     #endif
12         spin++;
13     }
14     pthread_mutex_lock(theLock);
15     return spin;
16 }

```

---

The data Memory Barrier (dmb) instruction ensures memory ordering within arm architecture. Based on the arm note: > All explicit memory accesses that appear in program order before the DMB instruction are observed before any explicit memory accesses that appear in program order after the DMB instruction. It does not affect the ordering of any other instructions executing on the processor.

---

```

1  for (int i = 0; i < 1000; i++) {
2      hybridLock(&microLock,1);
3      asm volatile("dmb sy" : : : "memory");
4      pthread_mutex_unlock(&microLock);
5      asm volatile("dmb sy" : : : "memory");

```

6 }

---

The result is 130 ns per loop with a 1.0GHz CPU and 100 ns per loop with a 1.3GHz CPU. Transfer to cycles is 130 cycles.

Back-to-back clock access is 18 ns on a 1.0 GHz CPU, which is 18 cycles. Note, on the arm CPU, the cycles counter increment by a fixed ratio. On raspberry pi, the frequency is 54 MHz, so 18 cycles is the minimum resolution of access.

## BIBLIOGRAPHY

- [PATCH 0/3] *Sapphire Rapids Co.x Idle States Support* - Artem Bityutskiy (2023).  
 URL: <https://lore.kernel.org/linux-pm/20230306123418.720679-1-dedekind1@gmail.com/> (visited on 05/17/2023).
- Announcing Project Monterey - Redefining Hybrid Cloud Architecture* - VMware vSphere Blog (2023). URL: <https://blogs.vmware.com/vsphere/2020/09/announcing-project-monterey-redefining-hybrid-cloud-architecture.html> (visited on 05/26/2023).
- Badr, Mario and Natalie Enright Jerger (June 2014).  
 “SynFull: Synthetic Traffic Models Capturing Cache Coherent Behaviour”.  
 In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*.  
 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA).  
 Minneapolis, MN, USA: IEEE, pp. 109–120. ISBN: 978-1-4799-4394-4 978-1-4799-4396-8.  
 DOI: [10.1109/ISCA.2014.6853236](https://doi.org/10.1109/ISCA.2014.6853236). URL:  
<http://ieeexplore.ieee.org/document/6853236/> (visited on 12/02/2022).
- Barroso, Luiz André and Urs Hölzle (Dec. 2007).  
 “The Case for Energy-Proportional Computing”. In: *Computer* 40.12, pp. 33–37.  
 ISSN: 0018-9162. DOI: [10.1109/MC.2007.443](https://doi.org/10.1109/MC.2007.443). URL:  
<http://ieeexplore.ieee.org/document/4404806/> (visited on 05/25/2023).
- Barroso, Luiz André, Hölzle Urs, and Ranganathan Parthasarathy (2019).  
*The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Springer Nature.
- Belay, Adam, George Prekas, Christos Kozyrakis, et al. (n.d.).  
 “IX: A Protected Dataplane Operating System for High Throughput and Low Latency”.  
 In: (), p. 18.
- Belay, Adam, George Prekas, Mia Primorac, et al. (Jan. 16, 2017).  
 “The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane”. In: *ACM Transactions on Computer Systems* 34.4, pp. 1–39.  
 ISSN: 0734-2071, 1557-7333. DOI: [10.1145/2997641](https://doi.org/10.1145/2997641).  
 URL: <https://dl.acm.org/doi/10.1145/2997641> (visited on 09/21/2022).
- Binkert, N.L. et al. (July 2006). “The M5 Simulator: Modeling Networked Systems”.  
 In: *IEEE Micro* 26.4, pp. 52–60. ISSN: 0272-1732. DOI: [10.1109/MM.2006.82](https://doi.org/10.1109/MM.2006.82). URL:  
<http://ieeexplore.ieee.org/document/1677503/> (visited on 05/22/2023).
- Binkert, Nathan et al. (May 31, 2011). “The Gem5 Simulator”.  
 In: *ACM SIGARCH Computer Architecture News* 39.2, pp. 1–7. ISSN: 0163-5964.  
 DOI: [10.1145/2024716.2024718](https://doi.org/10.1145/2024716.2024718).  
 URL: <https://dl.acm.org/doi/10.1145/2024716.2024718> (visited on 09/20/2022).
- Boguslavsky, L. et al. (May 1, 1994). “Optimal Strategies for Spinning and Blocking”.  
 In: *Journal of Parallel and Distributed Computing* 21.2, pp. 246–254. ISSN: 0743-7315.

- DOI: [10.1006/jpdc.1994.1056](https://doi.org/10.1006/jpdc.1994.1056).  
 URL: <https://doi.org/10.1006/jpdc.1994.1056> (visited on 05/26/2023).
- Bovet, Daniel P. and Marco Cesati (2006). *Understanding the Linux Kernel*. 3rd ed. Beijing ; Sebastopol, CA: O'Reilly. 923 pp. ISBN: 978-0-596-00565-8.
- BPF Documentation — The Linux Kernel Documentation* (2022).  
 URL: <https://www.kernel.org/doc/html/latest/bpf/index.html> (visited on 09/20/2022).
- Brown, Jeffrey et al. (Mar. 2011).  
 “IBM Power Edge of Network Processor: A Wire-Speed System on a Chip”.  
 In: *IEEE Micro* 31.2, pp. 76–85. ISSN: 0272-1732. DOI: [10.1109/MM.2011.3](https://doi.org/10.1109/MM.2011.3). URL: <http://ieeexplore.ieee.org/document/5708124/> (visited on 05/26/2023).
- Cai, Qizhe et al. (Aug. 9, 2021). “Understanding Host Network Stack Overheads”.  
 In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*.  
 SIGCOMM '21: ACM SIGCOMM 2021 Conference. Virtual Event USA: ACM, pp. 65–77. ISBN: 978-1-4503-8383-7. DOI: [10.1145/3452296.3472888](https://doi.org/10.1145/3452296.3472888).  
 URL: <https://dl.acm.org/doi/10.1145/3452296.3472888> (visited on 03/26/2023).
- Chen, Tianshi, Zidong Du, and Ninghui Sun (n.d.). “DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning”. In: ().
- Chen, Yu-Ting et al. (Oct. 2013).  
 “Accelerator-Rich CMPs: From Concept to Real Hardware”.  
 In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*.  
 2013 IEEE 31st International Conference on Computer Design (ICCD).  
 Asheville, NC, USA: IEEE, pp. 169–176. ISBN: 978-1-4799-2987-0.  
 DOI: [10.1109/ICCD.2013.6657039](https://doi.org/10.1109/ICCD.2013.6657039). URL: <http://ieeexplore.ieee.org/document/6657039/> (visited on 05/26/2023).
- Cooper, Brian F. et al. (June 10, 2010).  
 “Benchmarking Cloud Serving Systems with YCSB”.  
 In: *Proceedings of the 1st ACM Symposium on Cloud Computing*.  
 SOCC '10: ACM SIGMOD-SIGOPS Workshop on Cloud Computing in Conjunction with SIGMOD 2010. Indianapolis Indiana USA: ACM, pp. 143–154.  
 ISBN: 978-1-4503-0036-0. DOI: [10.1145/1807128.1807152](https://doi.org/10.1145/1807128.1807152).  
 URL: <https://dl.acm.org/doi/10.1145/1807128.1807152> (visited on 05/25/2023).
- Corbet, Jonathan (2023). *NAPI Polling in Kernel Threads*.  
 URL: <https://lwn.net/Articles/833840/> (visited on 04/16/2023).
- Dean, Jeffrey and Luiz André Barroso (Feb. 2013). “The Tail at Scale”.  
 In: *Communications of the ACM* 56.2, pp. 74–80. ISSN: 0001-0782, 1557-7317.  
 DOI: [10.1145/2408776.2408794](https://doi.org/10.1145/2408776.2408794).  
 URL: <https://dl.acm.org/doi/10.1145/2408776.2408794> (visited on 09/20/2022).
- Delimitrou, Christina and Christos Kozyrakis (July 23, 2018).  
 “Amdahl’s Law for Tail Latency”. In: *Communications of the ACM* 61.8, pp. 65–72.  
 ISSN: 0001-0782, 1557-7317. DOI: [10.1145/3232559](https://doi.org/10.1145/3232559).  
 URL: <https://dl.acm.org/doi/10.1145/3232559> (visited on 09/20/2022).

- Dong, Yaozu et al. (Jan. 2010). “High Performance Network Virtualization with SR-IOV”. In: *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA). Bangalore, India: IEEE, pp. 1–10. ISBN: 978-1-4244-5658-1. DOI: [10.1109/HPCA.2010.5416637](https://doi.org/10.1109/HPCA.2010.5416637). URL: <http://ieeexplore.ieee.org/document/5416637/> (visited on 05/03/2023).
- Dutu, Alexandru et al. (May 2020). “Independent Forward Progress of Work-groups”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). Valencia, Spain: IEEE, pp. 1022–1035. ISBN: 978-1-72814-661-4. DOI: [10.1109/ISCA45697.2020.00087](https://doi.org/10.1109/ISCA45697.2020.00087). URL: <https://ieeexplore.ieee.org/document/9138982/> (visited on 06/15/2023).
- Falsafi, Babak et al. (2016). “Unlocking Energy”. In: *USENIX Annual Technical Conference*.
- Ferdman, Michael et al. (n.d.). “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware”. In: ().
- Generic Mutex Subsystem — The Linux Kernel Documentation* (2023). URL: <https://docs.kernel.org/locking/mutex-design.html> (visited on 04/20/2023).
- Gleixner, Thomas and Ingo Molnar (2010). *Linux/Genericirq.Rst at v5.15 · Torvalds/Linux · GitHub*. URL: <https://github.com/torvalds/linux/blob/v5.15/Documentation/core-api/genericirq.rst> (visited on 09/20/2022).
- Gope, Dibakar, David J. Schlais, and Mikko H. Lipasti (June 24, 2017). “Architectural Support for Server-Side PHP Processing”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture. ISCA '17: The 44th Annual International Symposium on Computer Architecture*. Toronto ON Canada: ACM, pp. 507–520. ISBN: 978-1-4503-4892-8. DOI: [10.1145/3079856.3080234](https://doi.org/10.1145/3079856.3080234). URL: <https://dl.acm.org/doi/10.1145/3079856.3080234> (visited on 09/20/2022).
- Guerraoui, Rachid, Hugo Guiroux, et al. (Feb. 28, 2018). “Lock–Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems”. In: *ACM Transactions on Computer Systems* 36.1, pp. 1–149. ISSN: 0734-2071, 1557-7333. DOI: [10.1145/3301501](https://doi.org/10.1145/3301501). URL: <https://dl.acm.org/doi/10.1145/3301501> (visited on 04/28/2023).
- Guerraoui, Rachid, Michal Kapalka, and Jan Vitek (June 2007). “STMBench7: A Benchmark for Software Transactional Memory”. In: *ACM SIGOPS Operating Systems Review* 41.3, pp. 315–324. ISSN: 0163-5980. DOI: [10.1145/1272998.1273029](https://doi.org/10.1145/1272998.1273029). URL: <https://dl.acm.org/doi/10.1145/1272998.1273029> (visited on 12/02/2022).
- Hauswald, Johann, Yiping Kang, et al. (June 13, 2015). “DjiNN and Tonic: DNN as a Service and Its Implications for Future Warehouse Scale Computers”.

In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15: The 42nd Annual International Symposium on Computer Architecture. Portland Oregon: ACM, pp. 27–40. ISBN: 978-1-4503-3402-0.

DOI: [10.1145/2749469.2749472](https://doi.org/10.1145/2749469.2749472).

URL: <https://dl.acm.org/doi/10.1145/2749469.2749472> (visited on 05/25/2023).

Hauswald, Johann, Michael A. Laurenzano, et al. (Mar. 14, 2015).

“Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15: Architectural Support for Programming Languages and Operating Systems. Istanbul Turkey: ACM, pp. 223–238. ISBN: 978-1-4503-2835-7.

DOI: [10.1145/2694344.2694347](https://doi.org/10.1145/2694344.2694347).

URL: <https://dl.acm.org/doi/10.1145/2694344.2694347> (visited on 05/25/2023).

“Intel® 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4” (2023). In.

Johnson, Ryan et al. (2009).

“Shore-MT: A Scalable Storage Manager for the Multicore Era”.

In: *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09*. The 12th International Conference. Saint Petersburg, Russia: ACM Press, p. 24. ISBN: 978-1-60558-422-5.

DOI: [10.1145/1516360.1516365](https://doi.org/10.1145/1516360.1516365).

URL: <http://portal.acm.org/citation.cfm?doid=1516360.1516365> (visited on 11/27/2022).

Jouppi, Norman P., Doe Hyun Yoon, et al. (June 2021).

“Ten Lessons From Three Generations Shaped Google’s TPUV4i : Industrial Product”.

In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). Valencia, Spain: IEEE, pp. 1–14. ISBN: 978-1-66543-333-4.

DOI: [10.1109/ISCA52012.2021.00010](https://doi.org/10.1109/ISCA52012.2021.00010).

URL: <https://ieeexplore.ieee.org/document/9499913/> (visited on 09/20/2022).

Jouppi, Norman P., Cliff Young, et al. (June 24, 2017).

“In-Datacenter Performance Analysis of a Tensor Processing Unit”.

In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17: The 44th Annual International Symposium on Computer Architecture. Toronto ON Canada: ACM, pp. 1–12. ISBN: 978-1-4503-4892-8.

DOI: [10.1145/3079856.3080246](https://doi.org/10.1145/3079856.3080246).

URL: <https://dl.acm.org/doi/10.1145/3079856.3080246> (visited on 09/20/2022).

Kanev, Svilen, Juan Pablo Darago, et al. (June 13, 2015).

“Profiling a Warehouse-Scale Computer”.

In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15: The 42nd Annual International Symposium on Computer Architecture.

- Portland Oregon: ACM, pp. 158–169. ISBN: 978-1-4503-3402-0.  
DOI: [10.1145/2749469.2750392](https://doi.org/10.1145/2749469.2750392).  
URL: <https://dl.acm.org/doi/10.1145/2749469.2750392> (visited on 09/20/2022).
- Kanev, Svilen, Kim Hazelwood, et al. (Oct. 2014). “Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications”.  
In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*.  
2014 IEEE International Symposium on Workload Characterization (IISWC).  
Raleigh, NC, USA: IEEE, pp. 31–40. ISBN: 978-1-4799-6454-3 978-1-4799-6452-9.  
DOI: [10.1109/IISWC.2014.6983037](https://doi.org/10.1109/IISWC.2014.6983037). URL:  
<http://ieeexplore.ieee.org/document/6983037/> (visited on 09/20/2022).
- Kanev, Svilen, Sam Likun Xi, et al. (Apr. 4, 2017).  
“Mallacc: Accelerating Memory Allocation”.  
In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17: Architectural Support for Programming Languages and Operating Systems. Xi’an China: ACM, pp. 33–45.  
ISBN: 978-1-4503-4465-4. DOI: [10.1145/3037697.3037736](https://doi.org/10.1145/3037697.3037736).  
URL: <https://dl.acm.org/doi/10.1145/3037697.3037736> (visited on 05/26/2023).
- Kang, Ki-Dong et al. (Oct. 18, 2021). “NMAP: Power Management Based on Network Packet Processing Mode Transition for Latency-Critical Workloads”.  
In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*.  
MICRO ’21: 54th Annual IEEE/ACM International Symposium on Microarchitecture.  
Virtual Event Greece: ACM, pp. 143–154. ISBN: 978-1-4503-8557-2.  
DOI: [10.1145/3466752.3480098](https://doi.org/10.1145/3466752.3480098).  
URL: <https://dl.acm.org/doi/10.1145/3466752.3480098> (visited on 04/15/2023).
- Karandikar, Sagar, Howard Mao, et al. (June 2018). “FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).  
Los Angeles, CA: IEEE, pp. 29–42. ISBN: 978-1-5386-5984-7.  
DOI: [10.1109/ISCA.2018.00014](https://doi.org/10.1109/ISCA.2018.00014).  
URL: <https://ieeexplore.ieee.org/document/8416816/> (visited on 09/20/2022).
- Karandikar, Sagar, Albert Ou, et al. (Mar. 9, 2020). “FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design”.  
In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20: Architectural Support for Programming Languages and Operating Systems. Lausanne Switzerland: ACM, pp. 715–731. ISBN: 978-1-4503-7102-5. DOI: [10.1145/3373376.3378455](https://doi.org/10.1145/3373376.3378455).  
URL: <https://dl.acm.org/doi/10.1145/3373376.3378455> (visited on 09/20/2022).
- Kasture, Harshad and Daniel Sanchez (Sept. 2016). “Tailbench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications”.

- In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*.  
 2016 IEEE International Symposium on Workload Characterization (IISWC).  
 Providence, RI, USA: IEEE, pp. 1–10. ISBN: 978-1-5090-3896-1.  
 DOI: [10.1109/IISWC.2016.7581261](https://doi.org/10.1109/IISWC.2016.7581261). URL:  
<http://ieeexplore.ieee.org/document/7581261/> (visited on 08/11/2022).
- Kernelnewbies (2023). *Linux\_5.12 - Linux Kernel Newbies*.  
 URL: [https://kernelnewbies.org/Linux\\_5.12#Kernel\\_thread-based\\_NAPI\\_polling](https://kernelnewbies.org/Linux_5.12#Kernel_thread-based_NAPI_polling) (visited on 04/16/2023).
- Kocberber, Onur et al. (Dec. 7, 2013).  
 “Meet the Walkers: Accelerating Index Traversals for in-Memory Databases”. In:  
*Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*.  
 MICRO-46: The 46th Annual IEEE/ACM International Symposium on  
 Microarchitecture. Davis California: ACM, pp. 468–479. ISBN: 978-1-4503-2638-4.  
 DOI: [10.1145/2540708.2540748](https://doi.org/10.1145/2540708.2540748).  
 URL: <https://dl.acm.org/doi/10.1145/2540708.2540748> (visited on  
 05/26/2023).
- Koehn, Philipp et al. (2007).  
 “Moses: Open Source Toolkit for Statistical Machine Translation”.  
 In: *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and  
 Demonstration Sessions - ACL '07*. The 45th Annual Meeting of the ACL.  
 Prague, Czech Republic: Association for Computational Linguistics, p. 177.  
 DOI: [10.3115/1557769.1557821](https://doi.org/10.3115/1557769.1557821).  
 URL: <http://portal.acm.org/citation.cfm?doid=1557769.1557821>  
 (visited on 11/27/2022).
- Leverich, Jacob and Christos Kozyrakis (2014).  
 “Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service”.  
 In: *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*.  
 The Ninth European Conference. Amsterdam, The Netherlands: ACM Press, pp. 1–14.  
 ISBN: 978-1-4503-2704-6. DOI: [10.1145/2592798.2592821](https://doi.org/10.1145/2592798.2592821).  
 URL: <http://dl.acm.org/citation.cfm?doid=2592798.2592821> (visited on  
 09/27/2022).
- Li, Jialin et al. (Nov. 3, 2014).  
 “Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency”.  
 In: *Proceedings of the ACM Symposium on Cloud Computing*.  
 SOCC '14: ACM Symposium on Cloud Computing. Seattle WA USA: ACM, pp. 1–14.  
 ISBN: 978-1-4503-3252-1. DOI: [10.1145/2670979.2670988](https://doi.org/10.1145/2670979.2670988).  
 URL: <https://dl.acm.org/doi/10.1145/2670979.2670988> (visited on  
 09/20/2022).
- Li Deng (Nov. 2012). “The MNIST Database of Handwritten Digit Images for Machine  
 Learning Research [Best of the Web]”.  
 In: *IEEE Signal Processing Magazine* 29.6, pp. 141–142. ISSN: 1053-5888.  
 DOI: [10.1109/MSP.2012.2211477](https://doi.org/10.1109/MSP.2012.2211477). URL:  
<http://ieeexplore.ieee.org/document/6296535/> (visited on 11/27/2022).

*Linux: Starting and Stopping a Kernel Thread* (2023).

URL: <https://web.archive.org/web/20130110210314/http://www.scs.ch/~frey/linux/kernelthreads.html> (visited on 04/23/2023).

LinuxFoundation (2016). *Networking:Napi*.

URL: <https://wiki.linuxfoundation.org/networking/napi> (visited on 09/20/2022).

Liu, Daofu et al. (Mar. 14, 2015).

“PuDianNao: A Polyvalent Machine Learning Accelerator”.

In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15: Architectural Support for Programming Languages and Operating Systems. Istanbul Turkey: ACM, pp. 369–381. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694358.

URL: <https://dl.acm.org/doi/10.1145/2694344.2694358> (visited on 05/26/2023).

*Lock Types and Their Rules — The Linux Kernel Documentation* (2023).

URL: <https://docs.kernel.org/locking/locktypes.html> (visited on 04/20/2023).

Lowe-Power, Jason et al. (Sept. 29, 2020). *The Gem5 Simulator: Version 20.0+*.

arXiv: 2007.03152 [cs].

URL: <http://arxiv.org/abs/2007.03152> (visited on 09/20/2022). preprint.

Mao, Yandong, Eddie Kohler, and Robert Tappan Morris (2012).

“Cache Craftiness for Fast Multicore Key-Value Storage”.

In: *Proceedings of the 7th ACM European Conference on Computer Systems - EuroSys '12*. The 7th ACM European Conference. Bern, Switzerland: ACM Press, p. 183.

ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168855.

URL: <http://dl.acm.org/citation.cfm?doid=2168836.2168855> (visited on 11/27/2022).

Martin, Milo M. K. et al. (Nov. 2005).

“Multifacet’s General Execution-Driven Multiprocessor Simulator (GEMS) Toolset”.

In: *ACM SIGARCH Computer Architecture News* 33.4, pp. 92–99. ISSN: 0163-5964.

DOI: 10.1145/1105734.1105747.

URL: <https://dl.acm.org/doi/10.1145/1105734.1105747> (visited on 05/22/2023).

Marty, Michael et al. (Oct. 27, 2019).

“Snap: A Microkernel Approach to Host Networking”.

In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*.

SOSP '19: ACM SIGOPS 27th Symposium on Operating Systems Principles.

Huntsville Ontario Canada: ACM, pp. 399–413. ISBN: 978-1-4503-6873-5.

DOI: 10.1145/3341301.3359657.

URL: <https://dl.acm.org/doi/10.1145/3341301.3359657> (visited on 03/24/2023).

Michael, Schott (n.d.). *Shared-Memory Synchronization*.

Nowatzki, Tony et al. (Mar. 2016).

“Pushing the Limits of Accelerator Efficiency While Retaining Programmability”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

- 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). Barcelona, Spain: IEEE, pp. 27–39. ISBN: 978-1-4673-9211-2.  
DOI: [10.1109/HPCA.2016.7446051](https://doi.org/10.1109/HPCA.2016.7446051). URL: <http://ieeexplore.ieee.org/document/7446051/> (visited on 05/26/2023).
- Ousterhout, John K (n.d.). “Scheduling Techniques for Concurrent Systems”. In: (). *Overview of Single Root I/O Virtualization (SR-IOV) - Windows Drivers | Microsoft Learn* (2023). URL: <https://learn.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-> (visited on 05/03/2023).
- Paoloni, Gabriele (Sept. 2010). *How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures*.
- Pavlo, Andrew et al. (June 29, 2009). “A Comparison of Approaches to Large-Scale Data Analysis”. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS ’09: International Conference on Management of Data. Providence Rhode Island USA: ACM, pp. 165–178. ISBN: 978-1-60558-551-2.  
DOI: [10.1145/1559845.1559865](https://doi.org/10.1145/1559845.1559865).  
URL: <https://dl.acm.org/doi/10.1145/1559845.1559865> (visited on 05/25/2023).
- Pemberton, Nathan and Alon Amid (Mar. 2021). “FireMarshal: Making HW/SW Co-Design Reproducible and Reliable”. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Stony Brook, NY, USA: IEEE, pp. 299–309. ISBN: 978-1-72818-643-6.  
DOI: [10.1109/ISPASS51385.2021.00052](https://doi.org/10.1109/ISPASS51385.2021.00052).  
URL: <https://ieeexplore.ieee.org/document/9408192/> (visited on 09/20/2022).
- Rizzo, Luigi (n.d.). “Netmap: A Novel Framework for Fast Packet I/O”. In: (), p. 12.
- Salim, Jamal Hadi, Robert Olsson, and Alexey Kuznetsov (n.d.). “Beyond Softnet”. In: *5th Annual Linux Showcase & Conference*. Vol. 5. USENIX Association, p. 9.
- Sanchez, Daniel and Christos Kozyrakis (June 23–27, 2013). “ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems”. In: p. 12.
- Schlais, David J., Heng Zhuo, and Mikko H. Lipasti (Aug. 2020). “Modeling Architectural Support for Tightly-Coupled Accelerators”. In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Boston, MA, USA: IEEE, pp. 253–262. ISBN: 978-1-72814-798-7.  
DOI: [10.1109/ISPASS48437.2020.00045](https://doi.org/10.1109/ISPASS48437.2020.00045). (Visited on 08/15/2022).
- Sezer, Sakir et al. (July 2013). “Are We Ready for SDN? Implementation Challenges for Software-Defined Networks”. In: *IEEE Communications Magazine* 51.7, pp. 36–43. ISSN: 0163-6804.  
DOI: [10.1109/MCOM.2013.6553676](https://doi.org/10.1109/MCOM.2013.6553676). URL: <http://ieeexplore.ieee.org/document/6553676/> (visited on 05/03/2023).

- Souza, Rayanne et al. (Jan. 2020). "Real-time Performance Assessment Using Fast Interrupt Request on a Standard Linux Kernel". In: *Engineering Reports* 2.1. ISSN: 2577-8196, 2577-8196. DOI: [10.1002/eng2.12114](https://doi.org/10.1002/eng2.12114). URL: <https://onlinelibrary.wiley.com/doi/10.1002/eng2.12114> (visited on 10/11/2022).
- spec (2015). *Specjbb*. URL: <https://www.spec.org/jbb2015/> (visited on 11/27/2022).
- Sriraman, Akshitha, Abhishek Dhanotia, and Thomas F. Wenisch (June 22, 2019). "SoftSKU: Optimizing Server Architectures for Microservice Diversity @scale". In: *Proceedings of the 46th International Symposium on Computer Architecture. ISCA '19: The 46th Annual International Symposium on Computer Architecture*. Phoenix Arizona: ACM, pp. 513–526. ISBN: 978-1-4503-6669-4. DOI: [10.1145/3307650.3322227](https://doi.org/10.1145/3307650.3322227). URL: <https://dl.acm.org/doi/10.1145/3307650.3322227> (visited on 09/20/2022).
- Sriraman, Akshitha and Thomas F. Wenisch (Sept. 2018). "µSuite: A Benchmark Suite for Microservices". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 2018 IEEE International Symposium on Workload Characterization (IISWC). Raleigh, NC: IEEE, pp. 1–12. ISBN: 978-1-5386-6780-4. DOI: [10.1109/IISWC.2018.8573515](https://doi.org/10.1109/IISWC.2018.8573515). URL: <https://ieeexplore.ieee.org/document/8573515/> (visited on 09/20/2022).
- Thereska, Eno et al. (Nov. 3, 2013). "IOFlow: A Software-Defined Storage Architecture". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP '13: ACM SIGOPS 24th Symposium on Operating Systems Principles*. Farmington Pennsylvania: ACM, pp. 182–196. ISBN: 978-1-4503-2388-8. DOI: [10.1145/2517349.2522723](https://doi.org/10.1145/2517349.2522723). URL: <https://dl.acm.org/doi/10.1145/2517349.2522723> (visited on 05/03/2023).
- Threaded Network Device Interrupts / Linux Plumbers Conf: Developing the Kernel, Libraries and Utilities / September 23-25, 09 / Portland, OR* (2023). URL: <https://blog.linuxplumbersconf.org/ocw/proposals/53> (visited on 04/17/2023).
- Tu, Stephen et al. (Nov. 3, 2013). "Speedy Transactions in Multicore In-Memory Databases". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. SOSP '13: ACM SIGOPS 24th Symposium on Operating Systems Principles*. Farmington Pennsylvania: ACM, pp. 18–32. ISBN: 978-1-4503-2388-8. DOI: [10.1145/2517349.2522713](https://doi.org/10.1145/2517349.2522713). URL: <https://dl.acm.org/doi/10.1145/2517349.2522713> (visited on 09/20/2022).
- Ubuntu Kernel Lifecycle and Enablement Stack* (2023). Ubuntu. URL: <https://ubuntu.com/kernel/lifecycle> (visited on 04/16/2023).

- Unreliable Guide To Hacking The Linux Kernel — The Linux Kernel Documentation* (2023).  
URL: <https://www.kernel.org/doc/html/v4.14/kernel-hacking/hacking.html?highlight=softirq> (visited on 04/23/2023).
- Virtanen, Pauli et al. (Mar. 2, 2020).  
“SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”.  
In: *Nature Methods* 17.3, pp. 261–272. ISSN: 1548-7091, 1548-7105.  
DOI: [10.1038/s41592-019-0686-2](https://doi.org/10.1038/s41592-019-0686-2).  
URL: <http://www.nature.com/articles/s41592-019-0686-2> (visited on 09/20/2022).
- Walker, Willie et al. (2004).  
“Sphinx-4: A Flexible Open Source Framework for Speech Recognition”. In: p. 18.
- Wang, Lei et al. (Feb. 2014).  
“BigDataBench: A Big Data Benchmark Suite from Internet Services”. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*.  
2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). Orlando, FL, USA: IEEE, pp. 488–499. ISBN: 978-1-4799-3097-5.  
DOI: [10.1109/HPCA.2014.6835958](https://doi.org/10.1109/HPCA.2014.6835958). URL:  
<http://ieeexplore.ieee.org/document/6835958/> (visited on 05/25/2023).
- Wang, Wei (2023). *Implement Kthread Based Napi Poll*.  
URL: <https://lwn.net/ml/netdev/20201002222514.1159492-1-weiwan@google.com/> (visited on 04/16/2023).
- Yu-Wen Huang et al. (Mar. 2005). “Analysis, Fast Algorithm, and VLSI Architecture Design for H.264/AVC Intra Frame Coder”.  
In: *IEEE Transactions on Circuits and Systems for Video Technology* 15.3, pp. 378–401.  
ISSN: 1051-8215. DOI: [10.1109/TCSVT.2004.842620](https://doi.org/10.1109/TCSVT.2004.842620). URL:  
<http://ieeexplore.ieee.org/document/1397780/> (visited on 05/26/2023).
- Xapian (2022). URL: <https://xapian.org> (visited on 11/27/2022).
- Yan, Qiao et al. (Spr. 2016). “Software-Defined Networking (SDN) and Distributed Denial of Service (DDoS) Attacks in Cloud Computing Environments: A Survey, Some Research Issues, and Challenges”.  
In: *IEEE Communications Surveys & Tutorials* 18.1, pp. 602–622.  
ISSN: 1553-877X, 2373-745X. DOI: [10.1109/COMST.2015.2487361](https://doi.org/10.1109/COMST.2015.2487361).  
URL: <https://ieeexplore.ieee.org/document/7289347/> (visited on 05/03/2023).
- Yang, Albert Mingkun and Tobias Wrigstad (Dec. 31, 2022).  
“Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK”.  
In: *ACM Transactions on Programming Languages and Systems* 44.4, pp. 1–34.  
ISSN: 0164-0925, 1558-4593. DOI: [10.1145/3538532](https://doi.org/10.1145/3538532).  
URL: <https://dl.acm.org/doi/10.1145/3538532> (visited on 11/27/2022).
- Zhang, Ruiyi et al. (2023). “(M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels”. In: *USENIX-Security Usenix Security Symposium*.
- Zhang, Yanqi, Yu Gan, and Christina Delimitrou (Mar. 2019).  
“μqSim: Enabling Accurate and Scalable Simulation for Interactive Microservices”.  
In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2019 IEEE International Symposium on Performance Analysis of Systems

and Software (ISPASS). Madison, WI, USA: IEEE, pp. 212–222. ISBN: 978-1-72810-746-2. DOI: [10.1109/ISPASS.2019.00034](https://doi.org/10.1109/ISPASS.2019.00034).

URL: <https://ieeexplore.ieee.org/document/8695669/> (visited on 09/20/2022).

Zhang, Yunqi et al. (June 2016). “Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).

Seoul, South Korea: IEEE, pp. 456–468. ISBN: 978-1-4673-8947-1.

DOI: [10.1109/ISCA.2016.47](https://doi.org/10.1109/ISCA.2016.47). URL:

<http://ieeexplore.ieee.org/document/7551414/> (visited on 09/27/2022).

Zhuo, Heng and Mikko Lipasti (Feb. 25, 2023).

“TailWAG: Tail Latency Workload Analysis and Generation”.

In: *Proceedings of the 5th International Workshop on Benchmarking in the Data Center*.

BID ’23: 5th International Workshop on Benchmarking in the Data Center.

Montreal QC Canada: ACM, pp. 1–9. ISBN: 9798400701177.

DOI: [10.1145/3583060.3583170](https://doi.org/10.1145/3583060.3583170). (Visited on 05/30/2023).