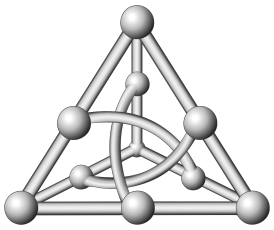
A Fast and Scalable Feedback-Driven Scheduler for Datacenter Applications

Mayco Souza Berghetti

Advisor: Prof. Ronaldo Alves Ferreira, Ph.D.

Co-advisor: Prof. Fabrício Barbosa de Carvalho, D.Sc.



College of Computing
Federal University of Mato Grosso do Sul
2025

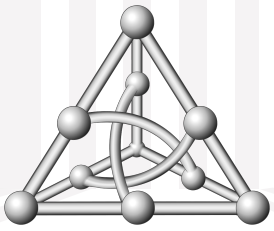
A Fast and Scalable Feedback-Driven Scheduler for Datacenter Applications

Mayco Souza Berghetti

Master's Thesis

Advisor: Prof. Ronaldo Alves Ferreira, Ph.D.

Co-advisor: Prof. Fabrício Barbosa de Carvalho, D.Sc.



College of Computing
Federal University of Mato Grosso do Sul
2025

Abstract

Microsecond-scale datacenter applications demand strict latency guarantees while operating under high load and variable service times. This environment often involves a mix of extremely short and long requests, where short requests—lasting just a few microseconds—are frequently delayed by longer ones due to Head-of-Line (HOL) blocking, leading to higher latencies, especially at the tail. However, existing approaches to mitigate HOL blocking, such as centralized dispatching, fine-grained preemption, and resource reservation, face fundamental scalability limitations. This work introduces Synergy, a cooperative, application-aware scheduling system that uses direct feedback from applications to prioritize short requests, dynamically adapts scheduling parameters, and avoids unnecessary preemptions. Synergy adopts a decentralized architecture with distributed queues, job-aware preemption, and dynamic quantum sizing. By eliminating centralized classification and using real-time application measurements, Synergy effectively mitigates HOL blocking without compromising throughput. Synergy outperforms state-of-the-art systems, achieving up to 43% higher throughput while meeting microsecond-scale service-level objectives.

Keywords: *datacenter, head-of-line blocking, user-level scheduler.*

"The important thing is not to stop questioning.
Curiosity has its own reason for existing."

— Albert Einstein

Agradecimentos

Dedico este trabalho à minha mãe do coração, Carolina Berghetti. Suas palavras de encorajamento, sempre presentes nos momentos mais desafiadores, foram fundamentais para que eu seguisse em frente com coragem e determinação.

Agradeço, com profundo carinho, à minha esposa, Nadia Tatiane dos Santos Ojeda, ao meu irmão, Marlon Souza Berghetti, e à minha mãe, Marinete Souza da Silva Berghetti. O apoio e o incentivo constante de vocês foram essenciais para a realização desta etapa tão importante da minha vida. Sou sinceramente grato por estarem ao meu lado em cada passo desta jornada.

Minha sincera gratidão ao Professor Ronaldo Alves Ferreira, meu orientador, por sua dedicação, paciência e comprometimento ao longo desta jornada acadêmica. Suas orientações e ensinamentos foram de valor inestimável, contribuindo significativamente para meu crescimento pessoal e profissional. Graças ao seu apoio, tive a honra de apresentar parte do nosso trabalho no Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) em 2024, uma experiência enriquecedora e marcante. Sou imensamente grato por sua confiança, disponibilidade e por ser uma fonte de inspiração.

Agradeço profundamente ao Fabrício Barbosa de Carvalho por ter me acompanhado ao longo desta jornada. Sua orientação atenta, generosidade ao compartilhar conhecimentos e constante incentivo foram fundamentais para o desenvolvimento deste trabalho. Sua colaboração teve um impacto significativo no meu crescimento técnico e acadêmico ao longo do processo.

Estendo também meu agradecimento ao amigo Maximilian Jaderson de Melo, cuja paciência e disposição em compartilhar seu conhecimento foram essenciais nos primeiros passos desta caminhada. Sua ajuda inicial fez toda a diferença e sou profundamente grato por isso.

Expresso minha sincera gratidão aos professores Luciano Paschoal Gaspary (UFRGS), Nahri Balesdent Moreano (UFMS) e Carlos Alberto da Silva (UFMS) por gentilmente aceitarem integrar minha banca examinadora e pelas valiosas contribuições oferecidas, as quais foram fundamentais para o aprimoramento desta dissertação.

Por fim, agradeço ao Instituto Federal de Mato Grosso do Sul (IFMS) pelo suporte institucional e pelas condições oferecidas ao longo desta jornada acadêmica.

Agradeço também à Universidade Federal de Mato Grosso do Sul (UFMS) pela valiosa oportunidade de formação, pelos recursos disponibilizados e pelo ambiente propício ao desenvolvimento deste trabalho.

Acknowledgements

I dedicate this work to my heart mother, Carolina Berghetti. Your words of encouragement, always present during the most challenging moments, were essential in helping me move forward with courage and determination.

I express my deep affection and gratitude to my wife, Nadia Tatiane dos Santos Ojeda, my brother, Marlon Souza Berghetti, and my mother, Marinete Souza da Silva Berghetti. Your unwavering support and constant encouragement were crucial to the completion of this important chapter in my life. I am truly grateful for having you by my side every step of the way.

My sincere gratitude goes to Professor Ronaldo Alves Ferreira, my advisor, for his dedication, patience, and commitment throughout this academic journey. His guidance and teachings were of immeasurable value, contributing significantly to my personal and professional growth. Thanks to his support, I had the honor of presenting part of our work at the Brazilian Symposium on Computer Networks and Distributed Systems (SBRC) in 2024—an enriching and memorable experience. I am deeply thankful for his trust, availability, and for being a constant source of inspiration.

I am also deeply grateful to Fabrício Barbosa de Carvalho for accompanying me throughout this journey. His attentive guidance, generosity in sharing knowledge, and continuous encouragement were fundamental to the development of this work. His collaboration had a significant impact on my technical and academic growth throughout the process.

I would also like to extend my thanks to my friend Maximilian Jaderson de Melo, whose patience and willingness to share his knowledge were essential in the early stages of this journey. His initial support made all the difference, and I am profoundly thankful for it.

I express my sincere gratitude to Professors Luciano Paschoal Gaspary (UFRGS), Nahri Balesdent Moreano (UFMS), and Carlos Alberto da Silva (UFMS) for kindly agreeing to serve on my examination committee and for their valuable contributions, which were fundamental to the improvement of this dissertation.

Finally, I am grateful to the Federal Institute of Mato Grosso do Sul (IFMS) for the institutional support and the conditions provided throughout this academic journey. I also thank the Federal University of Mato Grosso do Sul (UFMS) for the

valuable educational opportunity, the resources made available, and the academic environment that enabled the development of this work.

List of Acronyms

API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
c-FCFS	Centralized First Come First Serve
CI	Compiler Interrupts
CPU	Central Processing Unit
d-FCFS	Descentralized First Come First Serve
DARC	Dynamic Application-aware Reserved Cores
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
DRAM	Dynamic Random-Access Memory
EAL	Environment Abstraction Layer
FCFS	First Come First Serve
HOL Blocking	Head-of-Line Blocking
I/O	Input/Output
IP	Internet Protocol
IPC	Inter-Processor Communication
IPI	Inter-Processor Interrupts
JBSQ	Join-Bounded-Shortest-Queue
JIQ	Join-Idle-Queue
JSQ	Join-Shortest-Queue
kRPS	Thousand Request per Second
LLC	Last-Level Cache
LLVM	Low Level Virtual Machine
MRPS	Million Request per Second
MSR	Model Specific Register
NAPI	New API
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access

PCI	Peripheral Component Interconnect
POSIX	Portable Operating System Interface
PS	Processor Sharing
RTC	Run-to-Completion
RTT	Round-Trip Time
SLO	Service-Level Objective
SR-IOV	Single Root I/O Virtualization
TCP	Transmission Control Protocol
TID	Thread Identifier
TLS	Thread-Local Storage
UDP	User Datagram Protocol
UINTR	User Interrupts
UITT	User-Interrupt Target Table
UPID	User Posted-Interrupt Descriptor
VF	Virtual Function

List of Algorithms

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Datacenter applications, such as real-time analytics, online gaming, and social networks, demand response times at microsecond scales to meet strict service-level objectives (SLOs) [70]. These applications are composed of complex, latency-sensitive workflows where every microsecond matters [71]. The challenge lies not only in processing large numbers of concurrent requests but also in ensuring that even the slightest delays are minimized [1, 19, 29].

At microsecond timescales, traditional software architectures struggle to keep up with the demands of high-throughput, low-latency workloads. Processing delays are compounded by factors such as contention for CPU cores [50], memory bandwidth [25], and sudden bursts of requests that introduce queuing and scheduling inefficiencies [19, 62]. To make matters worse, datacenter workloads often exhibit service times with high dispersion, where a mix of extremely short and long requests must coexist [12, 20, 29, 33, 36, 45, 53]. Short requests, taking just a few microseconds, are often delayed by longer ones—an issue known as *Head-of-Line (HOL) Blocking*—leading to higher latencies, especially at the tail [19, 62].

To address the HOL Blocking problem and bound tail latency, recent research has explored kernel-bypass systems with a variety of scheduling strategies [7, 25, 30, 34, 35, 50, 53, 55, 70], including employing centralized dispatchers for load balancing [20,

36, 51], preempting long requests to prioritize short ones [30, 33, 36, 45, 69], and intra-server resource reservation [20, 21, 51]. Unfortunately, these approaches often scale poorly and force servers to run at low utilization (*e.g.,* below 40%) to meet strict SLOs [5].

Each of these strategies faces fundamental limitations that reduce their effectiveness at scale. Centralized dispatchers [20, 33, 36, 50], despite effectively distributing load, become bottlenecks under high load, which limits throughput and causes servers to remain underutilized. Fine-grained preemptive schedulers [33, 36, 69] introduce substantial overhead due to indiscriminate context switching and interrupt handling. Even optimized variants [33, 45] struggle with workloads exhibiting high service time variance, leading to poor cache performance and increased CPU costs. Resource reservation strategies [20] further rely on external classifiers to distinguish request types, which adds redundant classification effort and often leads to incorrect predictions when service times depend on dynamic application states. This situation calls for more nuanced, workload-aware scheduling approaches.

In datacenters, where enterprises have full control over the application stack—including the application, operating system, kernel-bypass system, and application scheduler—there is significant potential to design more cooperative and effective scheduling mechanisms. This control can provide the scheduler with rich application-level knowledge—such as request types, service times, and workload patterns—so it can make more informed and fine-grained decisions. Additionally, the scheduler can incorporate real-time feedback from the application and measurement data to adjust scheduling parameters dynamically, allowing it to better meet strict SLOs, reduce tail latencies, and improve overall resource utilization.

1.1 Main Contributions

This work introduces Synergy, a system designed to work in close cooperation with applications to scale efficiently on multicore architectures while addressing the fundamental limitations of existing systems. By leveraging direct application feedback, Synergy enables differentiated treatment of requests based on their expected service times, allowing it to prioritize short requests and mitigate HOL Blocking without compromising throughput or scalability. Synergy combines decentralized scheduling—where requests are distributed across multiple queues and scheduled independently—with dynamic load balancing and job-aware preemption to meet microsecond-scale SLOs even under high-load, high-variance conditions typical of datacenter workloads.

Specifically, Synergy adopts a decentralized dispatcher, where the NIC (Network Interface Card) distributes requests across multiple queues, each mapped to a dedicated worker. To handle load imbalances, Synergy employs work stealing [55], allowing underutilized workers to pull requests from overloaded ones and dynamically rebalance the load. This mechanism ensures efficient resource utilization [46] while preserving the benefits of decentralized dispatching. By eliminating the need for a centralized dispatcher—a well-known source of contention in prior systems [20, 33, 36, 50]—Synergy avoids a critical bottleneck.

In Synergy, applications classify requests and provide feedback to the scheduler, enabling differentiated handling of short and long requests. Short requests run to completion without interruption to minimize latency. Long requests, by contrast, can be preempted and resumed later to avoid delaying others and mitigate HOL blocking. Preempted requests are placed in a common wait queue rather than being reinserted into the original worker's queue, which allows them to be redistributed more effectively across workers than with queue-length-based strategies, such as work stealing alone. This strategy ensures that long requests do

not accumulate unevenly and that idle workers can resume deferred computation, improving overall responsiveness and resource utilization.

In addition to scheduling decisions, Synergy uses application feedback to compute the scheduling quantum dynamically based on the service times of the requests and operator-defined parameters. This design eliminates the drawbacks of a large fixed quantum, which can unnecessarily delay short requests [35, 36, 45], and offers flexibility to prioritize different request types. Also, Synergy employs job- and load-aware conditional preemption instead of time-based preemption. This approach avoids unnecessary context switches when a worker queue is empty, which minimizes overhead and improves overall performance.

Finally, by delegating request classification to the application, Synergy eliminates the need for a centralized classifier [20] and supports more flexible and accurate classification. This approach accounts for scenarios where service times depend not only on the request type but also on factors such as the specific operations pipelined within a request [58] or the popularity of a search term [18].

We implement Synergy as a libOS using DPDKDPDK to bypass the Linux kernel and compare it with Shinjuku [36], Perséphone [20], Concord [33], and Tiny Quanta [45], which use different techniques to mitigate HOL blocking. Through application-aware design and several optimizations, Synergy significantly improves throughput while meeting microsecond-scale latency SLOs. For example, in the Extreme workload (§6.2), Synergy increases throughput by 24–43% over prior systems and effectively mitigates HOL blocking up to 81% system load. We also perform a comprehensive evaluation of Synergy's internal mechanisms and behavior under diverse conditions. This evaluation includes an ablation study, sensitivity experiments with varying configuration parameters, tests under load imbalance, and evaluations of scalability with an increasing number of cores. The source code of Synergy and the scripts for reproducing our results are available at `https://github.com/Synergy-repo/synergy`.

1.2 Thesis Organization

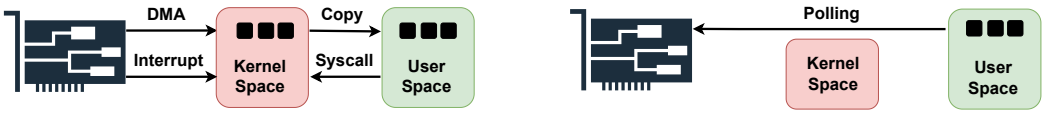
The remainder of this work is organized as follows. Chapter 2 introduces foundational concepts necessary to fully understand the work. Chapter 3 reviews the most relevant related work. Chapter 4 analyzes the design space explored by previous systems and presents the motivation behind the proposal developed in Chapter 5. Chapter 6 evaluates the proposed approach in comparison to existing solutions. Chapter 7 explores practical considerations and potential applications of the proposed feedback technique and provides a summary of its benefits. Finally, Chapter 8 summarizes the main contributions and concludes the thesis.

Chapter 2

Background

This chapter briefly overviews the background material necessary to understand the contributions of this work. Section 2.1 introduces packet processing in general-purpose operating systems, such as Linux, and presents techniques that improve processing speed. Section 2.2 describes request distribution strategies. Section 2.3 explains how to manage load balancing effectively. Finally, Section 2.4 examines different task scheduling models.

2.1 Kernel and Kernel-Bypass Packet Processing



(a) Kernel-based. (b) Kernel-bypass.

Figure 2.1: Kernel-based vs. Kernel-bypass packet processing.

The interval between the arrival of data at the NIC (Network Interface Card) and its consumption by the application involves several processing steps [10, 40, 68]. General-purpose operating systems like Linux typically use an interrupt-driven model for network processing. As illustrated in Figure 2.1a, when a packet arrives,

the NIC transfers it to main memory via DMA (*Direct Memory Access*) and raises an interrupt to notify the CPU. The operating system (OS) then processes the packet through the protocol stack. OS places the data in the reception queue of the corresponding socket based on the flow identified in the packet header. When the application performs a system call such as `recv`, the OS copies the packet data from kernel space to a user-space buffer.

Modern versions of Linux introduce optimizations to improve packet processing, including interrupt coalescence and NAPI (New API) [40]. With interrupt coalescence, the NIC intentionally delays interrupt generation to batch multiple incoming packets, thereby reducing interrupt overhead during high traffic. However, this batching introduces additional latency for the earliest packets in the batch.

NAPI replaces the purely interrupt-driven model with a hybrid approach that combines interrupts and polling. During high traffic, the system turns off interrupts and relies on a dedicated CPU core to periodically poll the NIC's receive queue and retrieve packets. This polling strategy improves performance and reduces interrupt handling overhead, although it may waste CPU cycles when traffic is low.

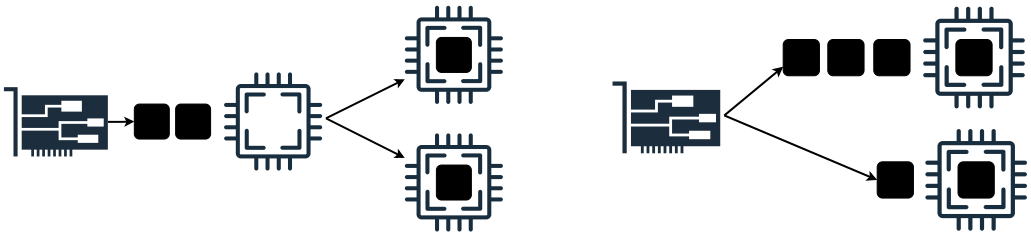
Instead of relying on the traditional kernel-based packet processing model, many high-performance networking systems adopt a technique known as kernel bypass, which allows user-space applications to interact directly with hardware (*e.g.,* the NIC) [7, 20, 25, 33, 36, 38, 45, 50, 53, 55], as illustrated in Figure 2.1b. These systems rely on frameworks such as DPDK [31], Netmap [59], and technologies like SR-IOV (Single Root I/O Virtualization) [53] to bypass the kernel and implement custom user-level network protocol stacks, thereby enabling low-latency and high-throughput network I/O.

By moving many traditionally kernel-level functions into user space, these systems reduce context switches, eliminate data copying between kernel and user memory, and give applications tighter control over scheduling and resource allocation. Additionally, full network polling is commonly employed in

high-performance kernel-bypass systems, improving latency and throughput compared to conventional kernel-based packet processing. For example, Shenango [50] improves CPU efficiency for many co-located latency-sensitive and batch applications by combining user-level networking with user-level thread scheduling. Kernel bypass systems work particularly well for microsecond-scale applications commonly found in modern datacenters, where even small delays in packet handling can significantly impact overall latency and system responsiveness.

2.2 Request Distribution

Request distribution strategies fall into two broad categories: centralized and decentralized models. In the centralized model, the system places all incoming requests into a single queue and assigns a dedicated core to dispatch these requests to available workers. Several systems adopt the centralized model by dedicating a core to perform request distribution between workers [20, 36, 51], as illustrated in Figure 2.2a. Although centralized dispatching offers better load balancing in theory [67], it suffers from scalability constraints in practice, particularly due to contention and synchronization overhead [46].



(a) Centralized. (b) Decentralized.

Figure 2.2: Request distribution strategies.

In contrast, the decentralized model distributes incoming requests across multiple queues. While this approach introduces additional complexity in maintaining load balance, it significantly improves scalability by reducing contention and increasing parallelism. Many systems implement the decentralized model by exploiting

hardware features available in modern Network Interface Cards (NICs) to implement decentralized dispatching, as depicted in Figure 2.2b. Examples include Receive Side Scaling (RSS) [61] and Intel Flow Director [52] mechanisms, both of which distribute network packets across multiple receive queues in the NIC. Recent high-performance I/O frameworks [7, 25, 55] widely adopt these mechanisms.

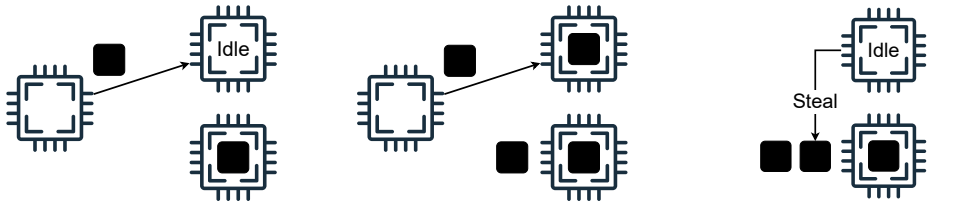
In the case of RSS, the NIC determines packet distribution by computing a hash over selected packet header fields, *e.g.,* 5-tuple of the packet. Rather than directly selecting a queue using the hash, the NIC uses it as an index into an indirection table stored in hardware. This indirection table maps hash values to receive queues, allowing flexible assignment of flows and enabling multiple distinct flows to share the same queue. Some systems dynamically adjust the indirection table at runtime to mitigate imbalances resulting from uneven flow distribution [3, 13–15].

2.3 Load Balance Management

Balancing the workload between workers ensures more efficient utilization of available resources, improves overall throughput, and reduces request latency. Various load-balancing policies can be employed to achieve this goal. Some rely on real-time knowledge of the system state, while others operate with minimal or no system awareness. These strategies generally reflect a trade-off between scalability and balance quality: the more precise the load distribution, the more overhead the system incurs, which can hinder scalability.

We now discuss the principal load balancing strategies that are most relevant to this work:

- **Join-Idle-Queue (JIQ)**, illustrated in Figure 2.3a, assigns incoming requests exclusively to idle workers. It achieves efficient load distribution with minimal overhead under moderate load, but it depends on tracking real-time worker availability. Although JIQ distributes load effectively, it scales poorly because



(a) Join-Idle-Queue (b) Join-B-Shortest-Queue (c) Work-stealing

Figure 2.3: Load balancing strategies.

it must continuously track which workers are idle.

- **Join-Bounded-Shortest-Queue (JBSQ)**, illustrated in Figure 2.3b, dispatches requests to any worker whose queue length is below a predefined threshold. Compared to JIQ, JBSQ reduces scalability limitations while offering a moderate load balance. Like JIQ, JBSQ implementations typically rely on a centralized dispatcher.

- **Join-Shortest-Queue (JSQ)**, illustrated in Figure 2.3b, dispatches requests to one of the workers with the fewest tasks in their queue. By relaxing the precision of load information, JSQ improves scalability relative to JIQ and JBSQ at the cost of slightly less optimal load distribution.

- **Work-stealing**, illustrated in Figure 2.3c, makes idle workers steal tasks from busier workers to balance the workload between workers. This technique avoids centralized coordination and scales well, although it generally yields less optimal load balance than centralized methods.

Overall, these policies reflect a fundamental trade-off: centralized strategies like JIQ and JBSQ can provide better load balance but suffer from limited scalability, while JSQ offers a middle ground by trading some balance precision for improved scalability. Decentralized approaches like work-stealing prioritize scalability further, at the cost of even less precise load distribution.

2.4 Task Scheduling Models

Task processing often follows the First-Come-First-Served (FCFS) model, where workers execute tasks in a run-to-completion (RTC) manner, strictly respecting their arrival order. When systems combine this execution model with centralized or decentralized request dispatching, they implement the scheduling policies known as c-FCFS and d-FCFS, respectively. Another widely used execution model is Processor Sharing (PS), in which each task runs for a fixed time slice (or quantum) before yielding the processor to allow another task to execute. According to queueing theory, FCFS performs best in workloads with low variability in processing times, whereas PS offers better performance under workloads characterized by high variability or heavy-tailed distributions [67].

To implement PS, systems must interrupt tasks frequently. Generally-purpose operating systems typically involve configuring hardware timers, such as the Advanced Programmable Interrupt Controller (APIC), to generate periodic interrupts. For instance, before executing a task, the system programs the APIC to trigger an interrupt after a specific time interval. Upon receiving the interrupt, the scheduler can preempt the current task and switch to another if needed.

In contrast to traditional kernel-level scheduling, several recent systems move scheduling decisions to user space [25, 30, 33, 36, 42, 50, 56], aiming to avoid the costly context switches associated with transitions between user and kernel modes. User-level schedulers can operate cooperatively, where the application developer manually inserts yield points in the code. Examples of libraries supporting this cooperative model include GNU Portable Threads [26], C++ coroutines [16], and Windows fibers [48]. Alternatively, user-level scheduling can be preemptive, which requires mechanisms capable of interrupting the currently running task.

Currently, the available mechanisms to support preemption in user space include: POSIX signals [9, 63], user-level interrupts [30, 42], and, compiler-based

interrupts [33, 45]. The following sections expand on the discussion of these alternatives.

2.4.1 Signals

POSIX-compliant systems use signals as the primary mechanism for event signaling and asynchronous communication in user space. Programs can generate signals programmatically (*e.g.,* via `alarm`) or explicitly send them between threads or processes (*e.g.,* using `tgkill`), which makes signals a viable tool for implementing user-space task preemption [9, 63].

Although POSIX signals offer a standardized and flexible interface for handling asynchronous events, they introduce considerable overhead, especially in latency-sensitive or high-frequency preemption scenarios. Much of this overhead stems from the need to transition between user and kernel modes. When a thread receives a signal, the kernel interrupts its execution, saves the processor state, and invokes a user-defined signal handler. Once the handler completes, the kernel must restore the thread's previous context, typically by issuing a `sigreturn` system call to resume execution at the point of interruption. Additionally, when delivering a signal to a process, the kernel must select an appropriate thread as the recipient, a process that may require inter-core communication and incur further scheduling overhead.

2.4.2 User Interrupts

User Interrupts (Uintr) [64] are a recent hardware mechanism that enables user-space applications to send and receive interrupts without kernel involvement, eliminating system call overhead. Intel's Sapphire Rapids processors support this feature through a set of Model Specific Registers (MSRs), as described below.

To receive user interrupts, an interrupt vector must be written to the `UINV`

register, allowing the processor to differentiate user interrupts from other types. When a user interrupt occurs, the processor transfers control to a user-space handler, which the system must preconfigure in the `IA32_UINTR_HANDLER` register. Each thread that can receive user interrupts maintains a User Posted-Interrupt Descriptor (UPID), pointed to by the `IA32_UINTR_PD` register. The UPID structure holds key information, such as the processor APIC ID that the thread is currently running.

To send user interrupts, software uses the `SENDUIPI` instruction, a new unprivileged instruction that triggers an interrupt to a designated user-level target. Each logical processor that can send user interrupts maintains a User-Interrupt Target Table (UITT), whose base address is stored in the `IA32_UINTR_TT` register. Each UITT entry corresponds to a receive thread and contains a pointer to the UPID structure of the target thread. When the `SENDUIPI` instruction is executed with a given index, the processor looks up the corresponding UITT entry, retrieves the UPID of the target, and delivers the interrupt using the local APIC to the target's logical processor. The receiving thread will then handle the interrupt using the handler address stored in `IA32_UINTR_HANDLER`.

By enabling fully user-space interrupt handling, Uintr removes system call overhead and achieves performance gains of up to 17.3× compared to traditional asynchronous Inter-Process Communication (IPC) methods [64]. This efficiency has motivated several recent research efforts that explore the use of Uintr in low-latency user-level systems [2, 30, 35, 42].

2.4.3 Compiler Interrupts

Compiler Interrupts (CI) implement a preemption mechanism based on compile-time instrumentation, where the compiler injects checks into the application code to determine whether the currently running task should yield the CPU. The compiler strategically places these yield checks (*e.g.,* at function boundaries), and the

application evaluates them during execution to decide whether a context switch should occur. Consequently, the frequency of context switching directly depends on how often the program includes these yield points.

One of the main advantages of CI is its low context-switching overhead. Unlike asynchronous preemption methods, such as signals, that require the system to save the full processor state, CI yields control with minimal state saving, entirely in user space. This design enables high efficiency in systems that require fine-grained task switching while avoiding the latency and complexity of system calls or signal handling.

Despite these advantages, CI introduces runtime overhead due to the inserted yield checks. When placed too frequently, especially within performance-critical sections such as tight loops, these checks can degrade performance significantly. For example, developers of the Go programming language reported performance slowdowns of up to 95% when applying aggressive instrumentation [27]. Therefore, effective use of CI requires balancing preemption granularity and system performance by carefully selecting where to place yield points, ideally avoiding the critical execution paths.

Chapter 3

Related Work

This chapter reviews the most relevant research related to this work. Section 3.1 presents works that specialize in the network protocol stack. Section 3.2 presents works that address different types of interference within a server. Finally, Section 3.3 covers works that tackle the problem of Head-of-Line (HOL) blocking. We defer the comparison of existing approaches with our's to Chapter 4.

3.1 Network Protocol Stack

The protocol stack forms a fundamental component of any network packet processing system. General-purpose operating systems typically implement the stack within the kernel. This design enhances security by isolating applications and simplifies networked software development. However, it also introduces significant processing overhead. Several factors contribute to this overhead, including frequent transitions between user and kernel modes, extensive security checks, and the reliance on large, generic data structures to manage connection state. Furthermore, kernel stacks support a broad range of TCP use cases, which reduce cache efficiency and introduce complex, multi-layered control flows that stall the processor pipeline.

To mitigate these performance bottlenecks, many high-performance systems

bypass the kernel entirely. They use virtualization technologies such as SR-IOV [65] or user-space frameworks like DPDK [31] to access the NIC directly and implement custom protocol stacks in user space. Although these approaches offer substantial performance gains, they often sacrifice security and generality in favor of speed [53].

This section examines systems that optimize the protocol stack to improve network application performance, exploring trade-offs between throughput, latency, and safety.

3.1.1 Arrakis

Arrakis [53] is an operating system designed to minimize the overhead along the application data path by leveraging hardware virtualization technologies. Traditional applications that rely on the Linux kernel and the POSIX socket API often suffer from performance limitations due to overhead introduced by kernel components, such as the protocol stack, scheduling, and packet copying. These factors prevent applications from fully utilizing the underlying hardware capabilities in terms of both throughput and latency.

To address this, Arrakis removes the kernel from the application's data path and delegates it to the role of a control plane responsible for ensuring isolation and security across applications. Arrakis bypasses the kernel during packet processing and eliminates costs associated with kernel scheduling, context switching, and system calls. Furthermore, implementing the protocol stack in user space enables a streamlined and efficient version that significantly reduces processing overhead. In practice, Arrakis achieves up to a 4× reduction in time spent within the protocol stack compared to Linux.

Arrakis employs SR-IOV [65], a technology originally developed to reduce virtualization overhead. SR-IOV allows the creation of virtual functions (VFs) on compatible network devices, with each VF appearing as a distinct PCI device that

can be independently assigned, typically to virtual machines. This direct hardware access eliminates the need for device emulation by a hypervisor. The hypervisor configures the physical NIC, including creating and managing VFs and setting up filters for packet (de)multiplexing between virtual entities.

In the context of Arrakis, the system repurposes SR-IOV to assign virtual functions directly to applications instead of virtual machines. The NIC performs packet demultiplexing and delivers packets directly to the corresponding application queues. Arrakis provides two socket interfaces: a standard POSIX-compatible API and a more efficient native interface. The native interface avoids copying packets between the user and kernel space, enabling higher performance.

Compared to Linux, Arrakis reduces total packet processing time by up to 8.8× using the native socket API and by 2.3× using the POSIX API. For real-world workloads, such as Memcached [47], Arrakis delivers up to a 1.7× throughput improvement. The system also demonstrates near-linear scalability with increasing core counts, up to six cores. Beyond this threshold, performance begins to degrade due to background processes managed by Barrelfish [4], the research operating system on which Arrakis is built.

3.1.2 mTCP

mTCP [34] introduces a user-level TCP/IP protocol stack designed for scalability on multicore systems, with a focus on ease of use and implementation. To achieve high scalability, mTCP applies several optimization techniques, including batch processing of packets and events, using lightweight, per-core data structures that avoid sharing between cores, and enforcing stream affinity per core to reduce synchronization overhead.

mTCP builds upon the PacketShader I/O engine [28] to enable direct access to the NIC from user space, extending it with a new event-driven API tailored for

high-performance networking. It also provides a drop-in replacement for applications using the POSIX socket API by offering analogous functions—*e.g.,* `accept` becomes `mtcp_accept`. For asynchronous event handling, it includes interfaces such as `mtcp_epoll`.

The mTCP stack runs as a dedicated thread pinned to the same core as the application thread. Although this architecture introduces some context-switching overhead, mTCP mitigates the impact through lock-free data structures and batching mechanisms that amortize processing costs across multiple events.

One of the core challenges addressed by mTCP is scaling the handling of large numbers of short-lived TCP connections. The Linux kernel suffers from several bottlenecks in such scenarios. For instance, when a socket queue is shared by multiple application threads, it introduces contention. Similarly, the global file descriptor space causes contention, as the kernel must locate the lowest available file descriptor during socket creation (*e.g.,* via the `accept` syscall), which becomes a bottleneck under high concurrency. Additionally, large data structures such as `sk_buff` introduce memory and cache inefficiencies when handling high packet rates.

Thanks to its architectural design and optimizations, mTCP achieves near-linear scalability regarding the number of CPU cores. Moreover, porting applications to mTCP is relatively straightforward: only a small number of code changes are required for programs already using the POSIX socket API—for example, adapting lighttpd required modifying only 65 lines of code. Performance evaluations show that mTCP improves TCP throughput by up to 3.2× compared to the Linux kernel.

3.1.3 TAS

TAS [38] accelerates TCP packet processing for datacenter applications by separating the handling of common and uncommon cases. TAS processes the common case, which includes in-order packet delivery, no fragmentation, and rare

retransmissions, on a fast path. The system delegates exceptional conditions, such as timeouts or out-of-order packets, to a slow path. TAS design uses the observation that most TCP traffic in datacenter environments follows a predictable and optimized pattern.

TAS consists of three main components that communicate via shared memory: the fast path, the slow path, and the protocol stack. The system implements the fast and slow paths as user-space processes pinned to dedicated CPU cores, each running in a separate thread. This architectural separation enables TAS to scale independently of the application workload. TAS exposes the protocol stack as a user-space library compatible with the POSIX socket interface, allowing unmodified applications to interact seamlessly.

TAS achieves high performance and low latency by isolating common-case processing in the fast path and bypassing the kernel through direct NIC access. The system also preserves key properties such as application-level isolation and scalability. Experimental evaluations show that TAS handles many concurrent TCP connections with minimal throughput degradation. Furthermore, TAS improves the tail latency of a key-value store application by 2.3× compared to IX [7], a prior high-performance TCP stack.

3.2 System Interference

Interference in a computer system occurs when some element of the system limits the performance of an application. This section covers systems that address different types of interference.

3.2.1 Arachne

Arachne [56] is a general-purpose user-space thread manager designed to deliver low latency and high throughput for applications that handle short-lived threads.

Many datacenter applications, such as Memcached [47], process requests with service times under 10 μs. In such scenarios, the high cost of thread creation in traditional systems, for example, C++ `std::thread` incurs a creation overhead of approximately 13.3 μs [56], makes per-request thread creation infeasible.

To mitigate this, applications typically use thread pools that initialize a fixed number of threads at startup to handle incoming requests. However, when the application spawns more threads than available cores to it, thread multiplexing causes contention and increases latency.

Arachne addresses these challenges by assigning exclusive core usage to applications over longer periods (tens of milliseconds) and dynamically allocating cores based on workload demands. Applications scale their internal parallelism according to the number of assigned cores. In addition, Arachne implements an efficient threading model that supports very short thread lifetimes on the order of microseconds while delivering performance comparable to hand-written event-driven code and preserving the simplicity of thread-based programming.

Arachne consists of three core components:

- **Core Arbiter**: A dedicated process that runs independently from Arachne applications. It manages core allocation among competing applications based on requests made through the Arachne runtime. Importantly, it does not forcibly revoke cores but uses a priority-based scheme to distribute resources fairly.

- **Arachne Runtime**: A lightweight thread library used by applications. It efficiently manages threads and handles load balancing by assigning new threads to underutilized cores at creation time.

- **Core Policy**: Allows each application to define how it will use its allocated cores. For example, an application may reserve a dedicated core for a specific thread (*e.g.,* a dispatcher), while distributing other threads across remaining

cores.

Experimental results show that Arachne improves the 99th percentile latency of Memcached by up to 40× compared to Linux. These gains stem from exclusive core usage, which reduces interference from other applications, and Arachne's lightweight thread management with minimal load-balancing overhead. When co-deployed with a compute-intensive workload (*e.g.,* the x264 video encoder), Arachne maintains Memcached's low latency.

Similar to systems like Arrakis [53], IX [7], and Shenango [50], Arachne seeks to combine low latency with efficient resource usage. However, unlike those systems, it does not rely on kernel-bypass mechanisms or direct NIC access. This design choice makes Arachne suitable for networked applications and general-purpose workloads requiring responsive, high-throughput threading on multicore systems.

3.2.2 Shenango

Shenango [50] is a system designed to reconcile efficient CPU utilization with the stringent latency requirements of network applications with tail latency Service-Level Objectives (SLOs) in the microsecond range. Unlike traditional systems that adjust core allocations at millisecond intervals [56], Shenango operates at a much finer granularity, making decisions every 5 μs. Shenango also uses 5 μs of queuing delay as an early congestion signal to determine when to allocate additional CPU cores to an application.

Shenango is composed of two main components: the IOKernel and the runtime. The IOKernel runs on a dedicated core and manages network I/O for applications. Shenango also executes a congestion detection algorithm determining when applications require additional computational resources. The runtime and application code communicate with the IOKernel via shared memory. Shenango provides a user-space protocol stack and thread management system, including

user-level thread scheduling and work-stealing for load balancing across cores.

Using Shenango, applications configure two CPU core allocations: guaranteed and burst cores. The system exclusively reserves guaranteed cores, ensuring a minimum level of performance that it cannot revoke. In contrast, the IOKernel dynamically allocates burst cores based on detected congestion and can reclaim them anytime. The runtime may also voluntarily return idle guaranteed cores to the IOKernel to improve overall CPU utilization.

Shenango's architecture allows it to efficiently serve a mix of workloads, including both batch-processing applications and latency-sensitive services, without sacrificing performance. Its responsiveness and flexible resource management enable better utilization of modern multicore systems, particularly in datacenter environments.

3.2.3 Caladan

Caladan [25] is a system designed to mitigate application interference while improving resource utilization. It builds upon Shenango [50] but introduces several key enhancements. In addition to queuing delay, which Shenango already uses, Caladan incorporates additional signals to detect load changes and interference.

One of Caladan's core innovations is its direct path mode, which bypasses the IOKernel's packet dispatching. This allows application runtimes to communicate directly with the NIC using the `libibverbs` [43] library, reducing processing overhead. Caladan also includes a custom kernel module called KSCHED, which implements more efficient scheduling mechanisms than the Linux kernel API (*e.g.,* `sched_setaffinity`). This module also collects performance counters from remote cores, including cache miss rates.

At deployment, applications receive two categories of CPU cores: guaranteed cores, which the application always has access to, and burst cores, which the system temporarily assigns based on load. Applications fall into Latency-Critical (LC)

and Best-Effort (BE). BE applications operate at a lower priority and only use burst cores. The system can revoke burst cores from BE applications to reduce interference or meet the demands of LC applications. Although LC applications avoid preemption by BE tasks, they can voluntarily release idle cores when load-balancing mechanisms, based on work stealing, fail to find new work.

Caladan uses queuing delay to identify when an application needs more cores. Once a need is detected, it considers both the application's classification (LC or BE) and additional interference signals to make allocation decisions. These signals include request processing time, DRAM bandwidth usage (DRAM BW), and last-level cache (LLC) miss rate.

The first metric helps manage hyper-thread interference by tracking how long a request runs on a core. If the runtime exceeds a defined threshold, the system asks the sibling hyper-thread to yield, idling the core. We use DRAM BW and LLC miss rate in combination to avoid DRAM channel saturation, which can increase memory access latency. When DRAM usage exceeds a set threshold, the system identifies the core with the highest LLC miss rate and revokes that core from the BE task, causing interference. This process continues until DRAM bandwidth usage drops below the limit.

Compared to Parties [17], a previous system designed to mitigate interference, Caladan shows similar performance under constant interference. However, it significantly outperforms Parties when interference is bursty (*e.g.,* due to garbage collection in BE applications). Caladan also achieves comparable latency to other kernel-bypass systems [50, 55], and delivers a throughput of 10 million requests per second. Due to its single-queue dispatch design, this is double that of its predecessor, Shenango, which is limited to 5 million requests per second.

3.3 Head-of-line Blocking

Head-of-line (HOL) blocking occurs when long-lived requests delay short-lived ones because the latter gets queued behind the former. This section discusses systems that specifically target this issue.

3.3.1 Shinjuku

Shinjuku [36] leverages hardware virtualization features to implement a user-space scheduling system capable of preemption at the microsecond scale. Its goal is to enable fair sharing of CPU resources across requests, preventing long-running operations from blocking short ones. It is crucial in applications with diverse service time distributions. For instance, get/set requests in key-value stores like Memcached [47] typically follow an exponential distribution with low variance, but background tasks (*e.g.,* garbage collection) can introduce long tails. Search engines often exhibit heavy-tailed request distributions such as log-normal, Zipf, or Pareto, while key-value databases like RocksDB [60] experience bimodal distributions, mixing fast operations (*e.g.,* get/put) with slower ones (*e.g.,* range scans).

Shinjuku extends Dune [6], a kernel module that exposes Intel Virtualization Technology (VT-x) features, to support Inter-Processor Interrupts (IPIs) used in preemption. The IPI handling path is heavily optimized, reducing overhead to 1993 cycles, an improvement of 2.1× over the unoptimized 4219-cycle path. Context switching is also enhanced by modifying the Linux `ucontext` library, reducing the cost from 2290 to just 109 cycles.

Shinjuku implements two scheduling policies. The first is a single-queue policy, where all requests go into a unified queue, and the dispatcher assigns the head-of-queue request to an available worker. If a request exceeds a predefined time quantum (typically 5–15 μs), the system preempts it and returns it to the queue. The

second policy uses multiple queues, where a network subsystem classifies requests by type (*e.g.,* get/scan). It assigns each type its queue and a user-defined 99th percentile latency SLO. The dispatcher selects the queue with the longest waiting time relative to its SLO. Preempted requests return to their respective queues and are placed at the head or tail depending on the distribution of their service times: short-tailed workloads go to the head, while long-tailed or multimodal workloads go to the tail.

In evaluation, researchers compare Shinjuku with IX [7] and ZygOS [55], both of which also use Dune. However, only Shinjuku supports preemption to mitigate HOL blocking. Under synthetic workloads, including fixed-latency (1 μs), exponential (mean 1 μs), and bimodal (99.5% at 0.5 μs, 0.5% at 500 μs), Shinjuku consistently delivers the best or near-best performance. In the bimodal case, Shinjuku achieves 5× higher throughput and 50% lower latency, while IX and ZygOS suffer from head-of-line blocking.

Further experiments using RocksDB [60] evaluate get requests (6 μs) and scans (240 μs for 1000 entries, 1200 μs for 5000 entries). In the bimodal workload (99.5% GET, 0.5% SCAN(1000)), Shinjuku again outperforms the others, achieving 6.6× higher throughput and 88% lower latency. In a more balanced bimodal workload (50% GET, 50% SCAN(5000)), Shinjuku demonstrates the importance of combining preemption with multiple-queue scheduling to maintain performance under mixed loads.

3.3.2 Perséphone

Perséphone [20] is designed to reduce tail latency in datacenter applications that operate at a microsecond scale and exhibit high service time dispersion (*i.e.,* requests have widely varying processing times). Key-value database applications, for instance, handle multiple request types, each with distinct service time

characteristics. In Redis [57], simple get/put requests typically have a service time of 2 μs [53], whereas more complex scan operations can require hundreds of microseconds [36].

Such variation in service times can lead to Head-of-Line (HOL) blocking, where short requests are delayed by long ones, thereby inflating latency. Perséphone uses the slowdown metric to quantify this effect, which is defined as the ratio of the total time a request spends in the system to its service time. For example, if a short request (1 μs) is queued behind a long one (500 μs), its slowdown is $(1+500)/1 = 501$, whereas the long request's slowdown is $(500 + 1)/500 = 1.002$. This asymmetry highlights how long requests can severely delay short ones, but not vice versa.

To address this issue, Perséphone introduces a scheduling policy called DARC (Dynamic Application-aware Reserved Cores), which avoids HOL blocking by dedicating exclusive CPU cores to short requests. The system profiles each request type's service time and uses the average CPU demand under high load conditions to determine how many cores to reserve. It also monitors queue delays and CPU demand variation to adjust these reservations dynamically in response to workload changes.

Requests are classified by type using an external classifier and placed into corresponding queues. The dispatcher selects requests from these queues and assigns them to available application cores. Short requests, with minimal impact on long ones, can execute on any free core, including those reserved. However, long requests are restricted to non-reserved cores to prevent interference with latency-sensitive workloads. This cycle stealing mechanism allows short requests to use reserved resources during bursts temporarily.

Perséphone [20] evaluated workloads with a dispersion of 100× to 1000× between short and long requests under four policies: DARC, d-FCFS, c-FCFS, and PS. DARC consistently delivered lower overall slowdown and latency across all scenarios while reducing peak throughput slightly (by 5%). Under a more stringent SLO,

however, Perséphone achieves higher throughput than the other approaches.

3.3.3 Concord

Concord [33] leverages LLVM passes [39] to automatically instrument applications at compile time by inserting yield points that help mitigate Head-of-Line (HOL) blocking. Concord's design is centered around three key components:

- **LLVM Passes**: Concord instruments application code using LLVM to insert checks periodically evaluating a preemption flag during request processing. When the flag is set, the application voluntarily yields the current request. This mechanism, known as compiler interrupts, offers a low-overhead alternative to hardware interrupts such as IPIs (Inter-Processor Interrupts).

- **JBSQ**: Concord employs a dispatcher thread to efficiently distribute requests between workers using the JBSQ (Join-Bounded-Shortest-Queue) load-balancing policy. This strategy minimizes contention introduced by the dispatcher and reduces worker idle time by assigning requests to the least loaded queues within bounds.

- **Dispatcher**: Beyond request distribution, the dispatcher can optionally participate in processing requests, though this is primarily beneficial under low-load conditions. More importantly, it is responsible for setting the preemption flags that signal workers to yield, enabling HOL blocking mitigation through cooperative multitasking.

Experimental results demonstrate that Concord achieves higher throughput than Shinjuku due to its lightweight preemption mechanism. However, this comes at the cost of slightly higher tail latency, which is attributable to the characteristics of the JBSQ policy. For instance, under a High workload, where 50% of requests have a 1 μs service time and the remaining 50% take 100 μs, Concord delivers 20% more

throughput than Shinjuku but incurs a slowdown increase of approximately 3× for the short requests.

3.3.4 Tiny Quanta

Tiny Quanta builds on a design similar to Concord's, using LLVM passes for instrumentation. However, unlike Concord, Tiny Quanta inserts probes that use the timestamp counter register (*e.g.*,, using `rdtsc` instruction) to track how long each request has been executing locally. If the execution time exceeds a predefined threshold, the worker voluntarily preempts the request. This self-managed approach removes the need for external coordination to trigger preemption.

Tiny Quanta implements a two-level scheduling mechanism. In the first level, a dispatcher assigns requests to worker queues using the Join-Shortest-Queue (JSQ) policy, which reduces contention and improves load balancing. Each worker applies a processor-sharing policy in the second level, enabling fine-grained control over request execution time and reducing head-of-line blocking.

Experimental results show that Tiny Quanta outperforms Shinjuku in both throughput and scalability. Specifically, it achieves 1.33× and 2.6× higher throughput than Shinjuku under High and Extreme workloads, respectively. Additionally, thanks to low dispatcher contention and efficient local scheduling, Tiny Quanta scales up to 12 million requests per second (MRPS) using 16 worker cores on workloads with low service time dispersion, significantly outperforming Shinjuku, which scales only up to 2 MRPS under similar conditions.

Chapter 4

Design Space

This chapter discusses the design space adopted by various systems targeting datacenter application processing and highlights unexplored optimization opportunities. By analyzing the common architectural choices, we seek to identify the trade-offs hindering performance and scalability and set the stage for new design directions.

4.1 Motivation

Recent advances in networking have led to transmission rates reaching hundreds of gigabits per second, with terabit links coming soon [11]. Meanwhile, processor speeds have remained relatively stagnant, making it necessary to scale network applications across multiple CPU cores. However, building systems that scale efficiently with core count while maintaining low tail latency remains challenging. Key issues include ensuring balanced load distribution across cores and preventing long-running requests from delaying shorter, latency-sensitive ones.

4.2 Request Dispatching and Load Balancing

To address the challenges of scaling datacenter applications on multicore architectures, both industry systems [24, 49] and academic work [3, 7] rely on sharding—partitioning application or protocol state (*e.g.,* TCP control blocks) and assigning each shard to a dedicated core. This strategy, known as *share-nothing* [55], is typically combined with Receive-Side Scaling (RSS) [61], where the NIC assigns incoming packets to queues based on a hash of the packet's contents (*e.g.,* five-tuple). Each queue maps to a specific core, ensuring that packets from the same flow are processed by the same core. The result is a decentralized scheduling policy known as d-FCFS (Decentralized First-Come, First-Served), where each core independently processes requests from its own queue to completion.

While d-FCFS supports scalable designs, it suffers from two major limitations. First, RSS does not always distribute requests evenly, which leads to load imbalances—some cores become overloaded while others sit idle—resulting in a non-work-conserving system and increased latencies [3]. Second, long requests can block short ones even when there are idle workers, a classic HOL blocking issue [20].

To balance the load across workers, several systems adopt variants of Centralized FCFS (c-FCFS), where all requests are placed in a shared queue and processed by multiple cores in arrival order [20, 36]. While this approach improves load distribution, it introduces its own challenges: bursts of long requests can monopolize all cores, delaying short ones and still causing HOL blocking. Systems like Shinjuku [36] and Perséphone [20] approximate c-FCFS by using a dedicated dispatching core that distributes requests to workers via a JIQ (Join Idle Queue) policy. Tardis [69], though operating with multiple NIC queues, also approximates c-FCFS by relying on a globally shared queue accessible to all workers.

Although c-FCFS provides theoretically optimal tail latency [67], it suffers from a scalability bottleneck, as the dispatcher or shared queue becomes a point of

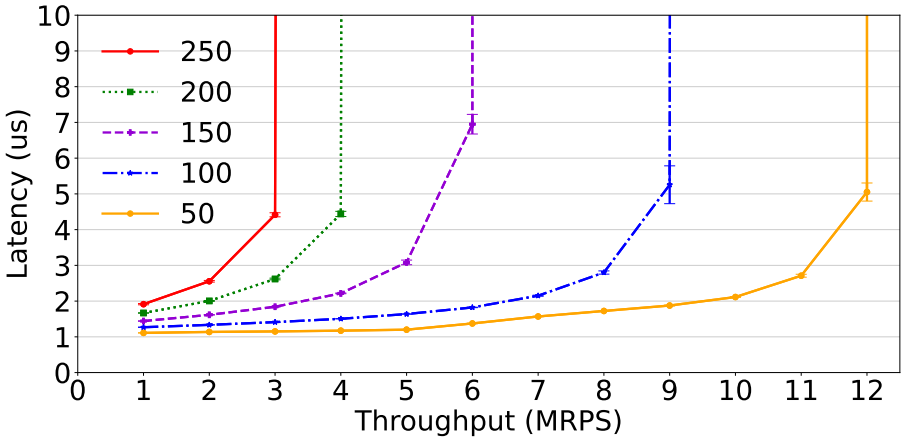


Figure 4.1: Simulation of different overheads in nanoseconds to centralized dispatching. The results show 99.9th latency on a function of load using 1 dispatcher, 14 workers and requests with service time of 1 μs.

contention [25, 46].

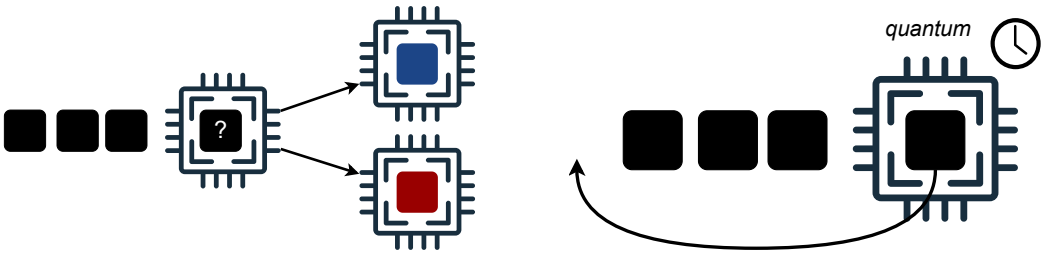
We extend the simulator provided by [46] to evaluate the scalability limitations of the c-FCFS policy by modeling a system with one dispatcher and 14 workers. This configuration reflects real-world deployments that employ c-FCFS [20, 36]. To simulate dispatching overheads, the dispatcher waits for x nanoseconds after receiving a new request before forwarding it. Figure 4.1 shows the 99.9th percentile tail latency as a throughput function, using requests with a service time of 1 μs. The results demonstrate that even a few nanoseconds of dispatching overhead can significantly impact system throughput and request tail latency.

To reduce this contention, Concord and Tiny Quanta adopt JBSQ (Join-Bounded-Shortest-Queue) and JSQ (Join-Shortest-Queue) policies, which reduce dispatcher contention at the cost of increased tail latency. Tardis takes a different approach: each worker dequeues N requests at a time from the global queue into a local queue. Smaller values of N improve load balance but increase contention; larger values reduce contention but worsen balance, making N a tunable parameter that controls how closely Tardis approximates c-FCFS.

Synergy takes a different approach. It adopts a decentralized dispatching

strategy where each worker receives requests directly from its NIC queue, thus avoiding the scalability issues of c-FCFS. To compensate for the inherent imbalance of multiple queues, SYNERGY employs two complementary techniques: work stealing [46, 55] and a global *wait queue* for preempted long requests. In workloads with high service time dispersion, equalizing queue lengths with work stealing is insufficient. For example, a worker handling a single long request may use more CPU cycles to process it than one handling many short requests. The global wait queue enables long requests to be redistributed among workers, improving balance beyond what static queue assignment with work stealing allows.

4.3 HOL-Blocking Mitigation



(a) Core reservation. (b) Preemption.

Figure 4.2: HOL Blocking mitigation strategies.

As shown in Figure 4.2, existing systems that address HOL blocking generally fall into two main categories: core-reservation and preemptive approaches.

Perséphone [20] is a recent system that dedicates cores exclusively to processing short requests. These reserved cores remain idle when there are no short requests to process. While this approach reduces overhead for the workers and prevents long requests from delaying short ones, it shifts complexity to the dispatcher, which must classify incoming requests and place them in separate queues, turning it into a potential bottleneck of the system.

Preemptive approaches, such as Shinjuku, Concord, Tardis, and Tiny Quanta,

enforce a fixed-time quantum for each request, limiting how long any request can run uninterrupted. This processor-sharing model is theoretically better for workloads with high dispersion [67]. However, enforcing it adds overhead. Shinjuku and Concord rely on dispatchers to interrupt workers—Shinjuku uses virtualization features to reduce IPI (Inter-Processor Interrupt) overhead. Concord, on the other hand, introduces Compiler Interrupts (CI), enabling workers to yield voluntarily based on a shared flag. Tiny Quanta uses CI alone, removing the dispatcher's role in interrupting. Tardis employs hardware timers and recent CPU features like user-level interrupts [64] to preempt workers periodically.

Synergy also uses preemption but improves upon prior designs by using job-aware conditional preemption. It interrupts long requests only when other jobs are waiting in the worker's queue, which avoids unnecessary context switches and reduces the system overhead.

4.4 Application Awareness

Most existing approaches are application-agnostic and treat all requests uniformly, ignoring valuable information that the application can provide [35, 55, 69]. This lack of awareness forfeits opportunities to make more informed scheduling and resource allocation decisions. By incorporating application-level information—such as request type, expected processing time, or priority—systems can adapt more effectively to workload characteristics, leading to more efficient resource usage and improved performance.

Some of the recent systems like Perséphone [20] and Shinjuku [36] distinguish request types. Perséphone classifies requests at dispatch time and assigns each worker to a specific class, allowing workers to process requests without interruption. However, classification is done externally by the dispatcher, duplicating effort the application later repeats. Shinjuku supports multiple queues per worker but relies

on the network subsystem—also running on the dispatcher—for classification.

Unlike prior work, SYNERGY uses direct application feedback to classify requests, eliminating the need for centralized classification. This approach eliminates redundancy and enables lightweight cooperation between the application and the scheduler. It also broadens the range of request types that can be treated differently, including those whose service times depend on internal state or application-specific factors (*e.g.,* search term popularity [18]).

4.5 Extra Core and Optimizations

Many designs rely on a dedicated core for auxiliary tasks such as dispatching, request classification, or triggering preemptions. However, since this core handles every request, it often becomes a scalability bottleneck. In Perséphone, for example, the dispatching core is also responsible for periodically adjusting resource allocation (*e.g.,* resizing the set of reserved cores for short requests), which temporarily pauses dispatching and increases queueing delays [20].

SYNERGY, like some prior systems, reserves an extra core for background tasks. Importantly, this core remains off the request-processing path, ensuring it does not limit scalability. It performs two key functions: *(i)* interrupting workers processing long requests when needed and *(ii)* monitoring the wait queue to ensure long requests make progress. These functionalities provide the following benefits:

Reduced Interrupt Frequency: SYNERGY minimizes preemptions by interrupting a long request only when there is a new request waiting on the worker's queue. This approach contrasts with prior systems that interrupt any request exceeding a fixed quantum, imposing constant overhead even when unnecessary. Fewer interrupts not only reduce the latency for long requests but also improve system throughput. Furthermore, SYNERGY ensures that short requests execute to completion without requiring a large quantum. This not only avoids the impact of

Table 4.1: Comparison of SYNERGY and prior systems across key scheduling design dimensions.

	SYNERGY	**Shinjuku** [36]	**Perséphone** [20]	**Concord** [33]	**Tardis** [69]	**Tiny Quanta** [45]
Request Dispatching	Hardware/Decentralized	Software/Centralized	Software/Centralized	Software/Centralized	Hardware/Decentralized	Software/Centralized
Load Balancing	Work Stealing and Wait Queue	JIQ	Reserved JIQ	JBSQ	Global Queue and Work Stealing	JSQ
HOL Blocking Mitigation	Preemptive	Preemptive	Core reservation	Preemptive	Preemptive	Preemptive
Application Aware	Yes	Yes	Yes	No	No	No
Extra-Core Function	Worker Interrupt and Monitoring	Request Dispatching and Worker Interrupt	Request Classification and Dispatching, and Core Reservation Updates	Request Dispatching and Worker Interrupt	-	Request Dispatching
Interrupt Frequency	Job-aware	Timed	-	Timed	Timed	Timed
Quantum Size	Dynamic	Static	-	Static	Static	Static

premature interruptions on short requests but also enables the use of small quantum.

Dynamic Quantum Sizing: Thanks to the application feedback, SYNERGY dynamically adjusts the quantum size based on workload conditions. By identifying the type of request each worker is processing and measuring the service time of short requests, SYNERGY can adjust the quantum size in real time. This adaptive strategy avoids the limitations of a fixed quantum: if the quantum is too large, short requests experience unnecessary delays when the worker is processing a long request; if too small, the system incurs excessive preemptions and added overhead. Also, operators can fine-tune quantum sizing parameters to prioritize specific request types according to application needs.

Together, the design choices in SYNERGY allow it to combine the scalability of decentralized dispatching with the latency benefits of intelligent, application-aware preemption, effectively mitigating HOL blocking without sacrificing performance. Table 4.1 summarizes the design space and compares SYNERGY with the existing systems.

Chapter 5

SYNERGY

This chapter introduces SYNERGY, a cooperative, application-aware scheduling system that uses direct feedback from applications to prioritize short requests, dynamically adjusts scheduling parameters, and avoids unnecessary preemptions. SYNERGY delivers fast, low-overhead, and scalable scheduling optimized explicitly for datacenter applications with microsecond-scale workloads, as we outline in Chapter 4. By combining efficient scheduling mechanisms with lightweight coordination, SYNERGY addresses the performance demands of modern low-latency and high-throughput environments.

5.1 Design

Figure 5.1 presents an overview of SYNERGY and illustrates the interaction between its main components. SYNERGY receives incoming requests through multiple NIC queues (*e.g.,* using RSS [61], Flow Director [52], or programmable NICs [37]). Each NIC queue is assigned to a single worker to prevent concurrent access to the hardware queues. SYNERGY also maintains a global *wait queue* to store preempted requests. For clarity, we describe the system using only two request types—short and long—but it can be extended to more types by using multiple wait queues with

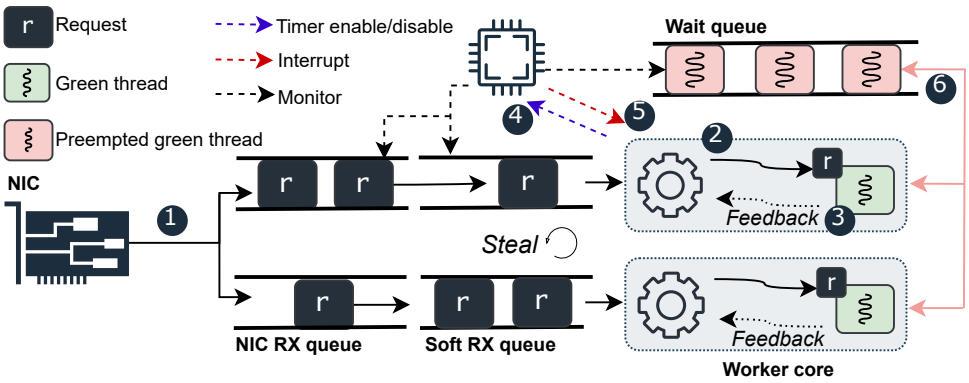


Figure 5.1: SYNERGY overview. Solid lines represent request path and dashed lines indicate timer core queue monitoring and interactions between timer core and worker cores.

priority levels based on request type.

To enable efficient work stealing, each worker transfers batches of requests from its NIC queue to a local software queue. It then processes each request using a reusable *green thread*, unlike prior systems that allocate a new green thread per request [33, 36, 69]. Short requests always run to completion and are never preempted. In contrast, when a long request is interrupted, SYNERGY moves its associated green thread to the wait queue and switches to a new green thread to process new requests.

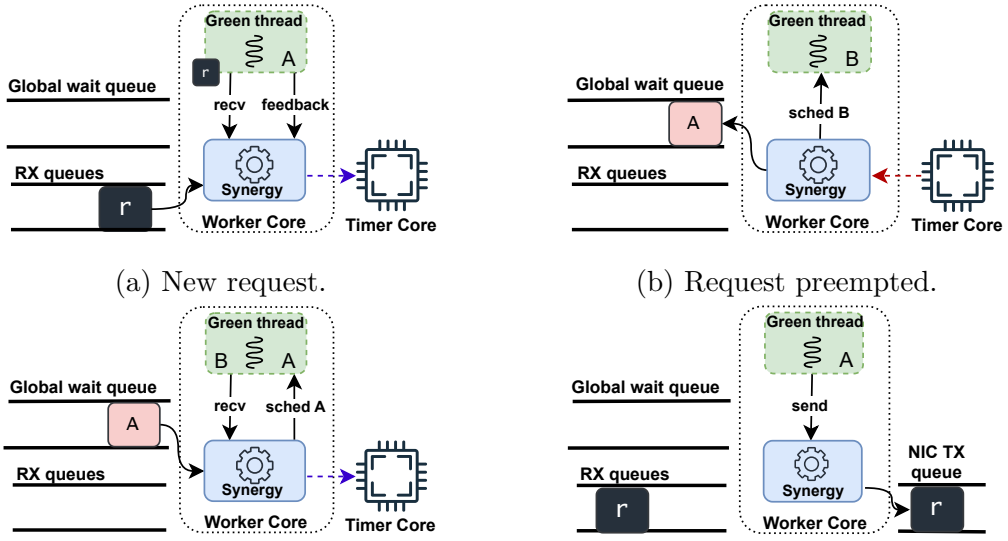
Any worker can access the wait queue to resume preempted requests to improve load distribution across cores. Workers also compute the average service time of short requests so that SYNERGY can dynamically calculate the quantum for long requests. The goal is to set the quantum so that a short request queued behind a long one is not delayed for more than the typical service time of a short request. Operators can adjust this behavior using a configurable multiplier to increase the CPU time allocated to long requests and reduce their total time in the system. Section 6.3.2 evaluates the impact of this multiplier.

SYNERGY also reserves a dedicated CPU core—referred to as the *timer core*—to coordinate time-sensitive scheduling tasks. It monitors both the workers' queues

and the wait queue, preempts long requests when needed, and signals workers to resume preempted requests that have been waiting in the wait queue for more than a specified threshold.

In summary, SYNERGY operates as follows: first, incoming requests are distributed across multiple queues ❶. Within each worker, a request is scheduled to the application ❷, which may correspond to either a new request or a previously preempted one. When a new request is scheduled, the application sends feedback to SYNERGY if the request is classified as long ❸, which triggers SYNERGY to activate a timer in the timer core ❹. When necessary, the timer core interrupts workers to mitigate head-of-line (HOL) blocking ❺, causing the worker to place the currently executing request into the wait queue ❻.

5.1.1 Worker Core



(a) New request.

(b) Request preempted.

(c) Request resumed.

(d) Request done.

Figure 5.2: Request life cycle. The components inside the dotted rectangle represent a single worker. Solid lines mean request path and SYNERGY operations. Blue dashed lines indicate worker actions to enable/disable the timer, signaling the timer core. Red dashed line represents an interrupt from the timer core to the worker core.

After initializing (*e.g.,* loading a database), the application calls `synergy()` to set up workers, queues, and timers (Algorithm 5.1, Line 12), passing a callback function to handle incoming requests. The SYNERGY initialization function launches a green thread on each worker to run the callback function (`server_loop()`), which in turn calls `synergy_recv()` (Line 3) to select a request for the worker to process (Algorithm 5.2).

Algorithm 5.1 Aplication Pseudocode

```
 1: function SERVER_LOOP                          ▷ Instantiated per worker.
 2:     loop
 3:         req ← synergy_recv()
 4:         if classify_request(req) == short then
 5:             process request req to completion
 6:         else
 7:             synergy_feedback_start()
 8:             process request req
 9:             synergy_feedback_finished()
10:         synergy_send(req reply)
11: ...                                           ▷ Application initialization.
12: synergy(server_loop)
```

To inform SYNERGY that it is processing a long request, the application invokes `synergy_feedback_start()` at the beginning (Line 7) and `synergy_feedback_finished()` at the end of processing (Line 9). Figure 5.2 illustrates the request life cycle. In the initial stage (Figure 5.2a), the application receives a request via `synergy_recv()`, classifies it as short or long, and, for long requests, provides feedback to SYNERGY, which then activates a timer managed by the timer core.

Later (Figure 5.2b), if the request exceeds its quantum and the worker's local queue is not empty, the timer core interrupts the worker. The corresponding green thread (*e.g.,* A) is moved to the wait queue, and a new thread (*e.g.,* B) is scheduled to process a new request. At a later point (Figure 5.2c), any worker may resume preempted requests from the wait queue to guarantee that they make progress. Finally, once processing is completed, the application sends the reply over the

network, concluding the request life cycle (Figure 5.2d and Line 10 of Algorithm 5.1). Short requests, by contrast, run to completion without interruption to minimize overhead and latency (Line 5).

5.1.1.1 Request Selection

Algorithm 5.2 shows how each worker selects the next request to execute. The worker first checks the `check_wait_queue` flag (Line 3) to determine whether it should prioritize a preempted request from the global wait queue ($\mathcal{Q}^{\text{wait}}$). This flag is set by the timer core when a green thread is waiting in the wait queue for a time longer than a specified threshold (Algorithm 5.3). This mechanism bounds the delay for long requests that have been waiting to resume.

Algorithm 5.2 SYNERGY Request Selection

```
 1: function SYNERGY_RECV
 2:     loop
 3:         if check_wait_queue then
 4:             if th ← Q^wait.dequeue() then
 5:                 resume green thread th
 6:         if req ← Q^RX.dequeue() then
 7:             return req
 8:         if th ← Q^wait.dequeue() then
 9:             resume green thread th
10:         for i = 1, ..., tot_workers-1 do
11:             j ← (worker_id + i) mod tot_workers
12:             if Q^tmp ← Q_j^RX.steal(STEAL_THRESHOLD) then
13:                 req ← Q^tmp.dequeue()
14:                 Q^RX ← Q^tmp
15:                 return req
```

If the flag is unset or the wait queue is empty, the worker fetches a request from its local software queue ($\mathcal{Q}^{\text{RX}}$) and begins processing (Lines 6 and 7). If the local queue is also empty, the worker rechecks the wait queue (Line 8) regardless of the `check_wait_queue` flag. If the wait queue is still empty, the worker attempts to steal requests from another worker's receive queue $\mathcal{Q}_j^{\text{RX}}$ (Line 12), where j is the chosen target. Work stealing occurs only if $\mathcal{Q}_j^{\text{RX}}$ has more requests than the

operator-defined `STEAL_THRESHOLD`, which helps avoid stealing too few requests and ensures the cost of stealing is amortized.

When resuming a preempted long request from the wait queue, its execution restarts in SYNERGY's interrupt handler. Before returning to the application, the interrupt handler marks the current green thread as processing a long request and starts a timer associated with the worker where the thread was resumed by calling `synergy_feedback_start()`.

5.1.1.2 Quantum Sizing

Choosing an appropriate value for the quantum is critical for the performance of a preemptive system: a quantum that is too small increases context-switching overhead, while a large one can delay short requests, leading to HOL blocking. Prior systems [33, 36, 45] use fixed quanta, typically between 2–15 μs, tuned offline based on workload characteristics. While simple, this approach is sensitive to runtime variations such as cache behavior and may result in short requests being preempted, incurring unnecessary overhead in the system.

SYNERGY, instead, adjusts the quantum dynamically at runtime. Each worker maintains an Exponential Moving Average (EMA) of short request service times, computed locally to avoid synchronization. To track short request service times, the worker computes the difference between a request's termination and scheduling times and updates the EMA with this value. When the application identifies a long request, the worker uses its EMA to set the preemption quantum. This per-worker approach ensures that quantum sizing adapts to workload conditions in real time with minimal overhead.

To provide additional flexibility, SYNERGY introduces a tunable quantum factor (QUANTUM_FACTOR) that scales the computed quantum. Operators can use this factor to allocate more or less CPU time to reduce the total latency of long requests or to increase the responsiveness of short requests. Although workloads

with extremely short requests (as low as 500 ns [20, 55]) make a small quantum more costly, SYNERGY mitigates this overhead through two strategies: *(i)* allowing operators to scale the quantum using the configurable factor and *(ii)* avoiding preemptions when no new requests are waiting to be processed. Together, these mechanisms balance the responsiveness required for short requests with the efficiency needed to handle long ones.

5.1.2 Request Classification

Prior systems often perform this classification externally [20, 36], requiring prior knowledge of the application's protocol. This approach introduces redundancy—since the application reclassifies the same request—and lacks generality, especially for complex applications like search engines [44].

SYNERGY takes a different approach by delegating classification to the application, which signals SYNERGY through lightweight feedback. When the application identifies a long request, it notifies SYNERGY to activate the timer interrupt on the current worker to bound the request's runtime and protect short requests from interference. This mechanism incurs minimal overhead, as notifications occur only for long requests, and short requests run to completion and are never interrupted.

By relying on application-level feedback, SYNERGY avoids hardcoding application-specific logic into the dispatcher. This separation of concerns allows applications to implement classification using criteria that are best aligned with their internal semantics, while preserving a decentralized scheduling architecture. As a result, SYNERGY eliminates the need for centralized classification and supports more complex classification schemes.

5.1.3 Timer Core

The timer core in SYNERGY is responsible for two key tasks: *(i)* monitoring the wait queue and *(ii)* interrupting workers when necessary. Algorithm 5.3 outlines these tasks.

Algorithm 5.3 Timer Core

```
 1: factor ← QUANTUM_FACTOR
 2: check_wait_queue ← false
 3: last_state ← false
 4: loop
 5:     state ← Q^wait.is_congested(THRESH_WQD)
 6:     if state ≠ last_state then
 7:         if state is true then
 8:             factor ← CONGESTED_QUANTUM_FACTOR
 9:         else
10:             factor ← QUANTUM_FACTOR
11:         check_wait_queue ← state
12:         last_state ← state
13:     for each w in workers do
14:         if timer_w expired then
15:             if Q_w^RX is not empty then
16:                 interrupt w to process new request
17:             else
18:                 renew timer_w deadline
```

5.1.3.1 Wait Queue Monitoring

The timer core periodically checks the wait queue to ensure the timely processing of preempted requests. Instead of tracking per-request timestamps, which would add overhead to workers, SYNERGY adopts a lightweight strategy inspired by Shenango's congestion control [50]. At each interval, defined by the user-configurable parameter `THRESH_WQD`, the timer core compares the wait queue's current consumer index with the producer index recorded during the previous check. If the consumer index has not advanced, this indicates that some requests have been waiting for at least `THRESH_WQD` microseconds.

When this condition is met, the timer core sets the `check_wait_queue` flag to true, prompting workers to prioritize the wait queue over local queues and work stealing, and ensuring that preempted requests are eventually resumed. The `THRESH_WQD` parameter must be carefully tuned to the workload. If it is set too low, the wait queue will consistently be marked as congested, causing workers to check it before their local queues. This behavior can reduce overall system efficiency. Conversely, if `THRESH_WQD` is set to the special value `0`, requests in the wait queue are processed in a best-effort manner, since the wait queue is never considered congested.

The timer core also applies the operator-defined parameter `CONGESTED_QUANTUM_FACTOR` to increase the quantum when the wait queue is congested, allowing long requests to run longer and helping them complete more quickly.

This design guarantees progress for all preempted requests while giving operators control over the trade-off between responsiveness for short requests and throughput for long ones. For example, if long requests violate their SLOs under heavy load, increasing `CONGESTED_QUANTUM_FACTOR` helps them complete sooner without sacrificing short-request performance under lighter load. As a result, SYNERGY adapts to dynamic workloads while maintaining low latency and high efficiency.

5.1.3.2 Worker Interrupt

The timer core in SYNERGY coordinates worker preemptions to mitigate HOL blocking while avoiding unnecessary interrupts. Rather than preempting blindly when a fixed quantum expires, SYNERGY takes the worker's local state into account. If no new requests are pending in the worker's queue, the current request continues running, avoiding wasteful context switches. Furthermore, we choose no preempt to process requests waiting in wait queue because the priority between requests is same, and the current request can finish faster. This selective, job-aware strategy improves efficiency over prior systems that rely solely on quantum timers [33,36,45].

Unlike hardware timer-based approaches [35, 69], which eliminate the need for a dedicated core but impose higher user-space overhead and reduced flexibility, SYNERGY's software-managed preemption offers finer control at lower cost. The timer core supports a wide range of interrupt delivery mechanisms, including signals [9, 63], Inter-Processor Interrupts (IPIs) [6, 36], user-level solutions like Intel's User Interrupts (UINTR) [35, 42, 64], and cooperative yielding via compiler instrumentation [33,45]. This flexibility allows SYNERGY to operate efficiently across diverse runtime environments and hardware platforms, making it well-suited for latency-sensitive applications.

5.2 Implementation

We implement SYNERGY as a user-space library linked directly with the application. Since each SYNERGY instance serves a single application, workers and the timer core run as threads within the same process, which simplifies variable sharing and coordination. While the current design targets single-application deployments, SYNERGY can be extended to support multiple applications by running the timer core in a separate process and using inter-process shared memory. The current implementation consists of 1,688 lines of C code, along with minimal assembly for green-thread context switching and interrupt handling. The optional kernel modules kmod_ipi and kmod_uintr (§6.3.3) contain 186 and 209 lines of C code, respectively.

5.2.1 Data Plane

SYNERGY uses DPDK [31] (v23.11) to bypass the kernel and access the NIC directly. Each worker owns a dedicated RX/TX queue pair and continuously polls its RX queue. Upon dequeuing a batch, the first request is processed immediately, while the remainder are placed into a per-worker software queue implemented as a lockless DPDK ring to support efficient work stealing without synchronization overhead.

Idle workers first try to resume long requests from the wait queue, implemented as a lockless multi-producer, multi-consumer ring. Despite being shared among all workers, this queue scales well due to three reasons: *(i)* it is lock-free; *(ii)* enqueue operations only occur during preemption events, which SYNERGY minimizes; and *(iii)* dequeue operations are limited to idle workers when the wait queue is not congested. If no tasks are found locally or in the wait queue, idle workers attempt work stealing, as discussed in Section 5.1.1.1.

5.2.2 Green Threads

SYNERGY implements green threads (*i.e.,* light-weight, user-level threads) from scratch using a custom data structure that stores the callee-saved (per the System V AMD64 ABI [66]), instruction pointer (RIP), and stack pointer (RSP) registers. This minimal context enables switching between green threads by copying only 64 bytes—*i.e.,* one cache line—making it more efficient than general-purpose alternatives.

Unlike prior systems that spawn one green thread for each new request [33, 36, 50, 56], SYNERGY reuses green threads for processing different requests, which reduces allocation and context-switch overhead. A new green thread is created only when the current one is preempted. SYNERGY further minimizes overhead when resuming a green thread from the wait queue: instead of returning to the main thread context, it switches directly to the resumed green thread. The preempted green thread is placed in a per-worker list for deferred deallocation. This design avoids unnecessary context switches, enables batch deallocation, and defers cleanup to opportune moments—such as when returning to the main thread due to an interrupt. Green threads are allocated using DPDK mempools with per-worker caches to reduce locking in (de)allocation operations.

Similar to prior work [33, 36, 50], SYNERGY disables preemption while the

application executes a critical section. Since SYNERGY only preempts long requests by design, this restriction applies exclusively to them. Critical sections include code protected by locks and functions that rely on thread-local storage (TLS) [22].

To handle locks, SYNERGY uses a shim layer that intercepts functions like `pthread_mutex_(un)lock` and forces the green thread to yield if it fails to acquire a lock and is not already holding one. If a thread holds a lock and fails to acquire a second, it will yield after exiting the critical section if the timer core requests preemption during the non-preemptible period. As SYNERGY targets low-latency applications, these critical sections are typically short, and nested locking is discouraged in scalable multicore designs.

For TLS, which is used internally by functions like `malloc` to avoid locking, SYNERGY disables preemption while such functions execute since green-thread-level preemption could lead to inconsistencies due to shared system-thread TLS contexts. Copying and restoring TLS state per green thread is impractical, as there are often more green threads than system threads, making concurrent reuse unsafe. Consequently, SYNERGY prevents green-thread preemption during TLS-dependent operations and disallows application-level TLS use in its current implementation. Future work may extend SYNERGY with green-thread-aware global variables to support TLS-like behavior safely.

5.2.3 Preemption Mechanisms

SYNERGY supports multiple preemption mechanisms to interrupt long requests and mitigate HOL blocking, including Compiler Interrupts (CI) and several IPI-based methods.

Compiler Interrupts (CI). CI relies on compiler-inserted yield points, allowing voluntary preemption without interrupting execution at arbitrary points. This approach avoids saving the full processor state and has been shown to outperform

asynchronous methods [33,45]. SYNERGY adopts a CI design similar to Concord [33], where the timer core sets a flag that workers periodically check. Because CI avoids interrupt delivery altogether, it incurs lower overhead—unless yield points fall inside tight loops, where it can add up to 20% overhead [45].

IPI-based Methods. For applications where CI is unsuitable, SYNERGY supports the following IPI mechanisms:

- **signal:** The default POSIX IPI mechanism suffers from high overhead due to system call transitions and general-purpose kernel routines (*e.g.,* TID-to-CPU mapping).
- **kmod_ipi:** A custom kernel module that we developed to reduce signal overhead. It supports direct CPU ID targeting and bypasses kernel features unrelated to SYNERGY. In the receiver, it saves RIP and transitions to a user-level handler to handle interrupts much faster.
- **uintr (User-level Interrupts):** A hardware feature on Intel Sapphire Rapids CPUs that allows IPIs to be sent and handled entirely in user space. Since it is not yet supported in mainline Linux, we implemented a kernel module to enable the **uintr** functionality on the processor.

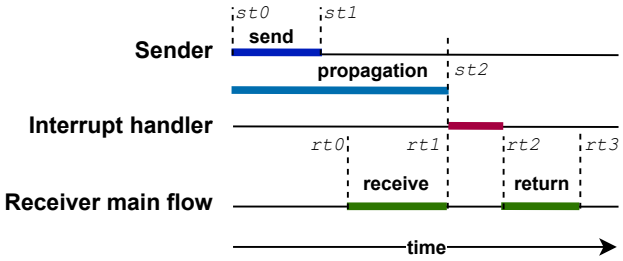


Figure 5.3: Interrupt delivery path stages measured in Table 5.1.

Table 5.1 presents microbenchmark results on an Intel Xeon Gold 6438Y+ system running Linux 5.15.0. The evaluation measures latency across the four phases

Table 5.1: Overhead (ns) for interrupt methods, showing 50th and 99.9th percentiles over 500k runs.

Method	send		propagation		receive		return	
	p50	**p99.9**	**p50**	**p99.9**	**p50**	**p99.9**	**p50**	**p99.9**
signal	746	929	2216	2533	1434	3185	641	677
kmod_ipi	490	549	1225	1659	728	2516	100	138
uintr	171	236	671	1354	464	1534	45	105

illustrated in Figure 5.3: **send** ($st1 - st0$), **propagation** ($st2 - st0$), **receive** ($rt1 - rt0$), and **return** ($rt3 - rt2$). Each method was tested over 500k runs with Turbo Boost and frequency scaling disabled. **uintr** achieves the lowest overhead among IPI-based methods because it avoids the kernel overhead.

Chapter 6

Evaluation

This chapter presents a comparative evaluation of SYNERGY against prior systems using both synthetic and real-world workloads. Our analysis focuses on key performance metrics, including tail latency, slowdown, and throughput across diverse load conditions and system configurations. To gain more in-depth insight into SYNERGY's design, we conduct an ablation study that isolates the impact of its main components. Additionally, we evaluate the SYNERGY's sensitivity to configuration parameters, the effectiveness of different interrupt mechanisms, and its robustness in the presence of load imbalance.

6.1 Methodology and Setup

This section describes our traffic generator, workloads, and testbed configuration for evaluating SYNERGY against state-of-the-art systems [20, 33, 36, 45].

We evaluate performance using synthetic and real workloads. Table 6.1 summarizes the four workloads used in our experiments. The High, Extreme, and ZippyDB workloads are synthetic and use a configurable application that consumes CPU cycles proportional to each request's target service time. These workloads exhibit high service time variability and are widely used in prior work [33, 36, 45].

Table 6.1: Evaluated workloads.

Workload	**Request Type(s)**	**Service time(s) (μs)**	**Ratio(s) (%)**
High	Short, Long	1, 100	50, 50
Extreme	Short, Long	0.5, 500	99.5, 0.5
ZippyDB	Short1, Short2, Long	0.5, 2.5, 500	78, 19, 3
LevelDB	GET, SCAN	0.92, 94	50, 50

We also include a real-world workload based on LevelDB [41], a key-value store that performs real `GET` and `SCAN` operations. The service times in Table 6.1 reflect averages from experiments with one million requests of each type.

Traffic is generated using an open-loop load generator built on DPDK [31] (v23.11), which bypasses the OS networking stack to minimize NIC and protocol overhead. The client sends 128-byte UDP requests over 512 independent flows. Request inter-arrival times follow an exponential distribution with mean N, where N corresponds to the mean time interval for a given request rate in Requests Per Second (RPS). This traffic pattern is commonly observed in datacenter workloads [12]. The client runs on three dedicated CPU cores: one for transmision (TX), and two for reception (RX). The first RX core polls the NIC and forwards replies to the second RX core, which records latencies. This separation minimizes interference and improves measurement accuracy.

Each experiment runs for 60 seconds, with the first 10% of responses discarded as warm-up. We report averages across 10 independent runs with 95% confidence intervals. We show individual-run results when relevant. Any run with more than 0.1% loss is discarded.

Our primary performance metric is the 99.9th percentile tail latency, critical for latency-sensitive services. Since long requests often dominate tail latency in high-variance workloads and to capture queuing effects independent of service time, we also report the 99.9th percentile of slowdown, defined as total request latency divided by its service time. This metric, often dominated by short requests, enables

consistent SLO comparisons across workloads and is widely used in related work [20, 33, 36].

6.1.1 Setup

Unless otherwise stated, experiments are conducted on two CloudLab [23] c6420 nodes connected via a 10 Gbps link. Each node features an Intel Xeon Gold 6142 CPU (16 cores @2.60 GHz), 376 GB RAM, and an Intel X710 10GbE NIC. Turbo Boost is disabled, and all cores are isolated from the Linux scheduler using the isolcpus parameter. DPDK is configured with 8,192 2MB hugepages, and threads are pinned to specific cores using EAL core masks and taskset to ensure NUMA locality and avoid contention. The average round-trip time (RTT) is 10 μs. All latency measurements are collected at the client side to avoid clock synchronization issues. We run Ubuntu 18.04 with Linux kernel 4.4.185 to ensure a fair comparison between Shinjuku and SYNERGY, as this is the latest version compatible with Dune [6], a dependency of Shinjuku.

6.2 SYNERGY *vs.* Preemptive Systems

We begin by comparing SYNERGY with leading preemptive systems: Shinjuku [36], Concord [33], and Tiny Quanta (TQ) [45], all of which use interrupt-based techniques to mitigate HOL blocking. Shinjuku uses hardware virtualization to reduce IPI overhead, while Concord and Tiny Quanta rely on Compiler Interrupts (CI). To ensure a fair comparison, we use CI in all systems where applicable and modify Shinjuku to use Concord's CI mechanism, referring to it as Shinjuku-CI.

Concord, Shinjuku, and Shinjuku-CI all use a fixed 5 μs quantum, as in the original Shinjuku paper. Reducing this value hurts performance, as shown in prior work [33]. Tiny Quanta uses a 2 μs quantum, which its authors report as optimal. In contrast, SYNERGY uses a dynamic quantum with parameter `QUANTUM_FACTOR`=2

and `THRESH_WQD`=0, a reserved value that instructs SYNERGY to ignore congestion in the wait queue and prioritize short requests while processing long ones from the wait queue in a best-effort manner. Finally, all systems use 14 dedicated CPU cores for the workers.

6.2.1 Synthetic Workload

6.2.1.1 High

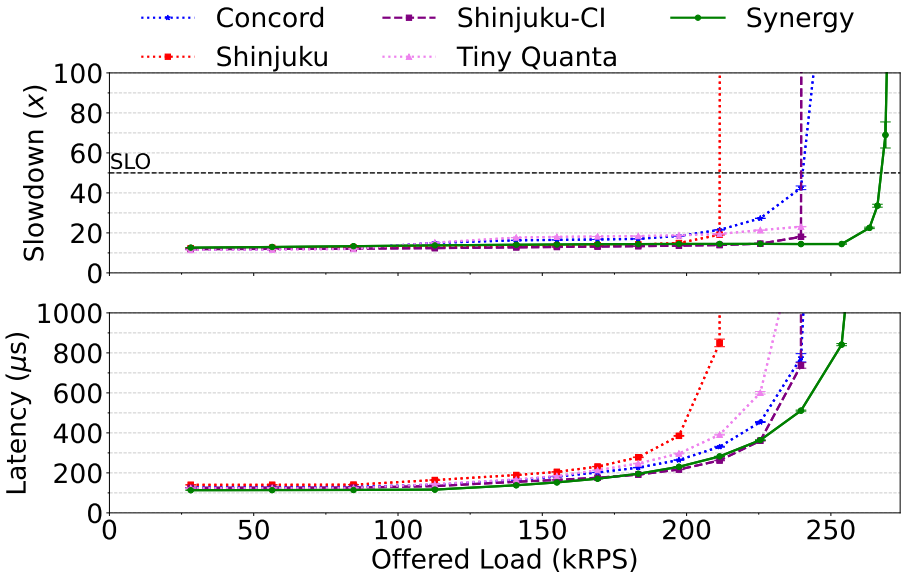


Figure 6.1: 99.9th percentile of both slowdown (top) and latency (bottom) in the High workload. At 239 kRPS, which represents 86% of load, SYNERGY reduces slowdown by 2.9×, 1.5×, and 1.2× compared to Concord, Tiny Quanta, and Shinjuku-CI, respectively, while maintaining 20% more throughput that Shinjuku to 50× slowdown SLO.

Figure 6.1 shows 99.9th percentile slowdown (top) and tail latency (bottom) for the High workload, respectively. Shinjuku suffers from high interrupt overhead, which limits its throughput. Replacing its IPI mechanism with CI (Shinjuku-CI) improves throughput by 11%, from 211 kRPS to 239 kRPS. Concord and Tiny Quanta achieve similar throughput of 239 kRPS before violating the SLO (in this case, a slowdown of at most 50), but Tiny Quanta has a smaller slowdown due

to its smaller quantum, while Concord offers better tail latency because its larger quantum favors long requests.

SYNERGY outperforms all systems, achieving 20% higher throughput than Shinjuku (211 *vs.* 266 kRPS) and 9% more than Shinjuku-CI, Concord, and Tiny Quanta (239 *vs.* 266 kRPS), for the target SLO. At 239 kRPS, which represents 86% of CPU load, SYNERGY reduces slowdown by 2.9×, 1.5×, and 1.2× (SYNERGY=14.42, Concord=42.55, TQ=23.12, Shinjuku-CI=18.02) compared to Concord, Tiny Quanta, and Shinjuku-CI, respectively.

At 211 kRPS (76% of CPU load), which corresponds to the request rate where all the systems still meet the SLO, SYNERGY reduces the tail latency (282 μs) by 3×, 1.38×, and 1.17× compared to Shinjuku=849 μs, TQ=390 μs, and Concord=330 μs, respectively, while remaining competitive with Shinjuku-CI=262 μs.

6.2.1.2 Extreme

Figure 6.2 shows the results for the Extreme workload. This workload has the highest dispersion among those in Table 6.1, and its extremely high proportion of short requests stresses the systems with a higher request rate.

Shinjuku performs worst due to dispatcher contention and constant interrupt overhead. Concord and Tiny Quanta scale better than Shinjuku by using the Join-Bounded Shortest Queue (JBSQ) and Join-the-Shortest-Queue (JSQ) load-balancing policies, respectively, which reduce contention at the dispatcher.

SYNERGY achives the best performance by combining the scalability of distributed queues with efficient load balancing and low-overhead optimizations. It achieves the highest throughput (3.79 MRPS) and lowest latency, delivering 37%, 31%, and 24% more throughput than Shinjuku (2.37 MRPS), Shinjuku-CI (2.61 MRPS), and Concord (2.84 MRPS), respectively, before each system violates the slowdown SLO. Compared to Tiny Quanta (48.62), SYNERGY reduces slowdown (35.77) by 1.35× before the system drops requests at 3.79 MRPS, which corresponds

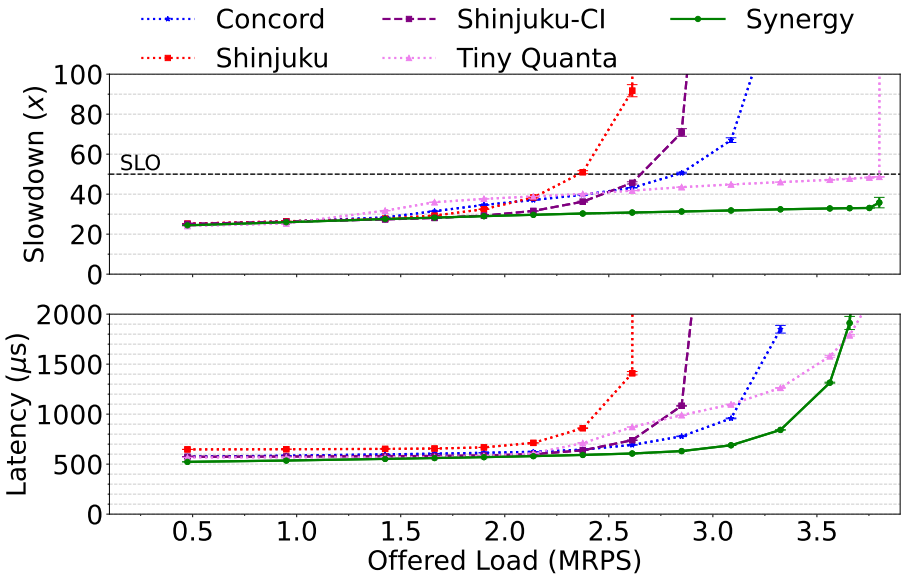


Figure 6.2: 99.9th percentile slowdown (top) and latency (bottom) for the Extreme workload. SYNERGY delivers up to 37% more throughput before SLO violation.

to 81% of CPU load. In terms of latency, SYNERGY outperforms all systems up to 3.56 MRPS (78% load). Because SYNERGY handles long requests on a best-effort basis, it consistently prioritizes short ones. Under heavier load, operators can further reduce latency by adjusting input parameters as needed.

6.2.1.3 ZippyDB

Figure 6.3 shows the results for the ZippyDB workload, which reflects the request distribution observed in production environments [12]. This workload balances the previously evaluated High and Extreme workloads, featuring a 1,000× (like Extreme) dispersion and a higher proportion of long requests. As a result, it stresses both system throughput and interrupt mechanisms.

Consistent with the previous experiments, Shinjuku delivers the lowest throughput, followed by Shinjuku-CI, Concord, and Tiny Quanta. SYNERGY achieves 843 kRPS, which is 36%, 25%, 20%, and 14% higher than Shinjuku (538 kRPS), Shinjuku-CI (628 kRPS), Concord (673 kRPS), and Tiny Quanta (717 kRPS), respectively, before violating the SLO.

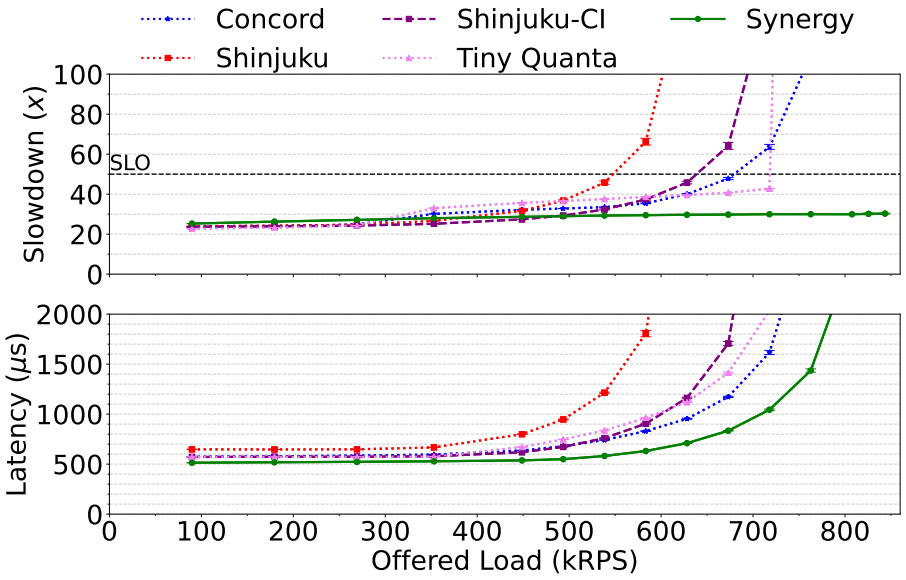


Figure 6.3: 99.9th percentile slowdown (top) and latency (bottom) for the ZippyDB workload. SYNERGY delivers up to 36% more throughput before SLO violation.

At 717 kRPS, 81% of CPU load, SYNERGY reduces slowdown by 1.47× compared to Tiny Quanta, from 42.7 to 29.97, and lowers tail latency by 1.54× compared to Concord, from 1,616 μs to 1,044 μs. Even when processing long requests in a best-effort manner, SYNERGY consistently achieves lower tail latency across all load levels.

6.2.2 Real Application - LevelDB

This section compares SYNERGY with Shinjuku-CI and Concord using a real application. All systems run the same LevelDB version (v1.23), compiled with Concord's LLVM compiler pass [39] to ensure identical instruction streams and eliminate variance from instrumentation or compiler differences. The database runs on a RAM-backed file system (*i.e.,* tmpfs) to eliminate disk I/O overhead and is preloaded with 1,000 unique entries. SCAN requests iterate over all entries, while GET requests target randomly selected keys.

Figure 6.4 shows the results. Since the workload resembles High, SYNERGY

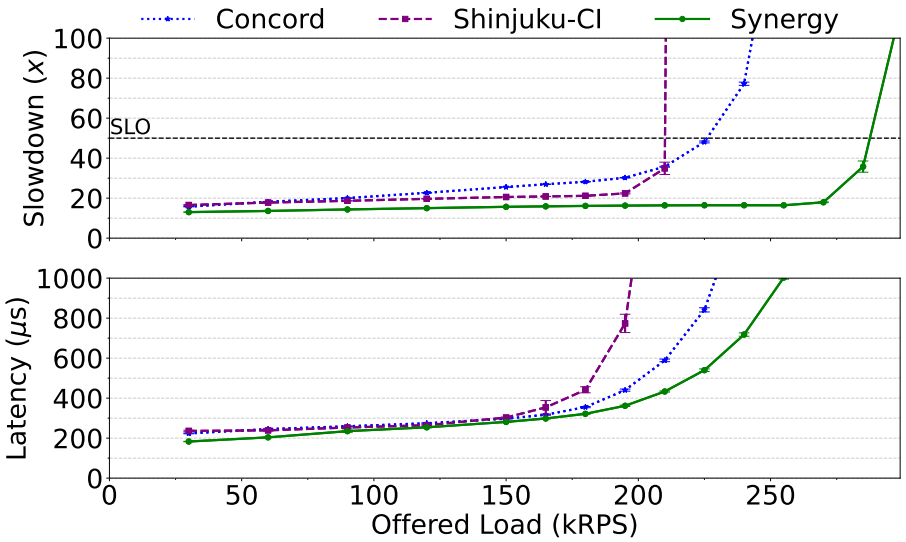


Figure 6.4: 99.9th percentile slowdown (top) and latency (bottom) for LevelDB with 1,000 entries under 50% GET and 50% SCAN workload.

behaves similarly to Figure 6.1. However, unlike the synthetic workload, LevelDB introduces lock contention in GET and SCAN operations. Synergy mitigates this contention by making long requests yield when failing to acquire a lock outside critical sections, allowing other requests to progress. In contrast, Concord and Shinjuku-CI block until the lock is acquired. Synergy achieves a throughput of 284 kRPS, which is 26% and 21% more throughput than Shinjuku-CI (209 kRPS) and Concord (224 kRPS), respectively, before violating the slowdown SLO. At 224 kRPS, 76% load, Synergy reduces slowdown by 2.92×, from 48.09 to 16.44 and tail latency by 1.56×, from 841 μs to 539 μs compared to Concord.

6.3 Ablation Study

This section presents a detailed analysis of Synergy. We begin by isolating the impact of each component on performance (§6.3.1), then examine how input parameters influence its ability to balance short and long requests (§6.3.2). We also evaluate the supported interrupt mechanisms (§6.3.3) and stress the system

under severe load imbalance (§6.3.4). Finally, we also compare SYNERGY to Perséphone [20], a non-preemptive system that reserves cores for processing short requests exclusively.

6.3.1 SYNERGY Components Breakdown

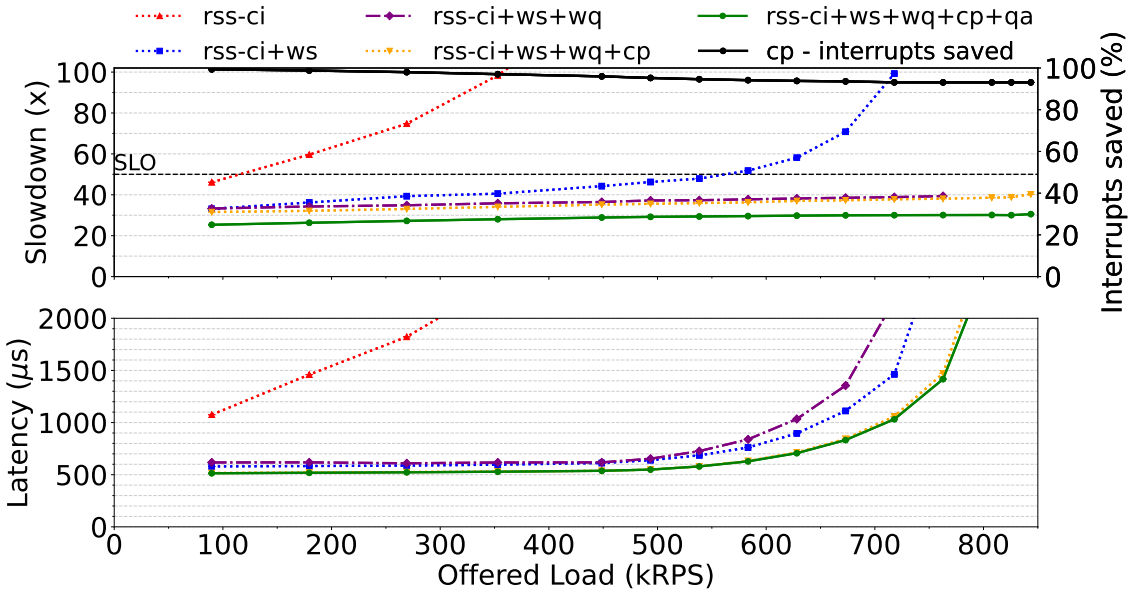


Figure 6.5: Performance improvement breakdown by individual SYNERGY components. The upper graph shows the 99.9th percentile of slowdown (left y-axis) and the percentage of avoided interruptions (right y-axis). The lower graph shows the 99.9th percentile of latency.

Figure 6.5 shows the individual contribution of each SYNERGY component using the ZippyDB workload. The first evaluated configuration, **rss-ci**, depends solely on multi-queue and interrupt-based processing. It uses the CI technique with a 5 μs quantum, where preempted requests are placed at the end of the worker's queue. This setup serves as the baseline, with additional components incrementally enabled. As expected, **rss-ci** delivers the worst performance, with slowdown and latency of 46.14 and 1075 μs, respectively, at 10% of system capacity with 89,739 RPS, since it lacks any mechanism to balance requests across queues. To address this limitation, the second configuration, **rss-ci+ws**, incorporates work-stealing to balance the load

among the queues.

Although work-stealing improves the system's processing capacity, sustain throughput of 538 kRPS before SLO violation, it falls short under workloads with high variability for two main reasons: *(i)* it cannot balance the load with sufficient granularity since equalizing the number of requests per queue alone is not enough, and *(ii)* it lacks visibility into which request should be processed next, causing many short requests to wait behind long ones for at least one quantum. The following configuration, **rss-ci+ws+wq**, introduces the wait queue to address these limitations. The wait queue enables more efficient distribution of long requests across idle workers and prioritizes processing newly arrived requests.

Since the wait queue prioritizes the processing of newly arrived requests, the slowdown drops significantly, sustaining throughput of 762 kRPS, which represents 86% of system capacity, before the system starts dropping requests. The top graph of Figure 6.5 highlights the queue's effectiveness in mitigating HOL blocking. However, the wait queue also increases tail latency, especially after 61% of system capacity with 538 kRPS, as shown in the bottom graph, as long requests dominate the tail latency. It reflects a trade-off between mitigating HOL blocking and handling long requests. Note that it is only necessary to preempt the processing of a long request when that request is likely causing HOL blocking, motivating the following configuration, **rss-ci+ws+wq+cp**, which introduces conditional preemption.

Unlike prior preemptive systems that interrupt requests based on fixed timers, Synergy only preempts a long request after it has executed for at least one quantum and when the worker's queue contains newly arrived requests, an indication of potential HOL blocking (§5.1.3). The right axis of the top graph of Figure 6.5 shows the percentage of preemptions avoided compared to the configuration without conditional preemption. As expected, the higher offered load increases the need for preemptions to mitigate HOL blocking. Synergy avoids over 90% of preemptions, which increases system throughput to 95% of system capacity with 843 kRPS and

significantly improves tail latency, even under high utilization. As the results show, conditional preemption is the key optimization that enables SYNERGY to consistently achieve the best tail latency compared to the preemptive systems evaluated in §6.2.

Finally, a fixed quantum may not provide optimal performance for short requests. For example, a 5 μs quantum can cause up to a 10× slowdown for requests with a service time of 0.5 μs. To address this limitation, we evaluate the final configuration, **rss-ci+ws+wq+cp+qa**, which introduces automatic quantum adjustment. In addition to using application-level feedback to minimize timer management overhead, we use this feedback to compute the quantum based on the workload characteristics (§5.1.1). In this experiment, the automatic quantum computation reduces the slowdown compared to the fixed quantum configuration without impacting latency, reducing the slowdown from 40.23 to 30.54 with 95% of system utilization.

6.3.2 Dynamic Request Priority

This section evaluates SYNERGY under different configurations to prioritize short or long requests using the Extreme workload. Figure 6.6 shows results with `QUANTUM_FACTOR`=2 fixed, while `THRESH_WQD` and `CONGESTED_QUANTUM_FACTOR` are varied. For example, the configuration (d15-f20) sets the *THRESH_WQD*=15 μs and the *CONGESTED_QUANTUM_FACTOR*=20. In particular, configuration **(d0-f0)** indicates that the wait queue is never considered congested, and the *CONGESTED_QUANTUM_FACTOR* is never used. As a result, requests in the wait queue are processed in a best-effort manner, giving maximum priority to short requests. We use this configuration as the baseline in the subsequent evaluations.

The (d15-f20) configuration is the most tolerant to queueing delay before considering the wait queue congested. As a result, the dynamic priority

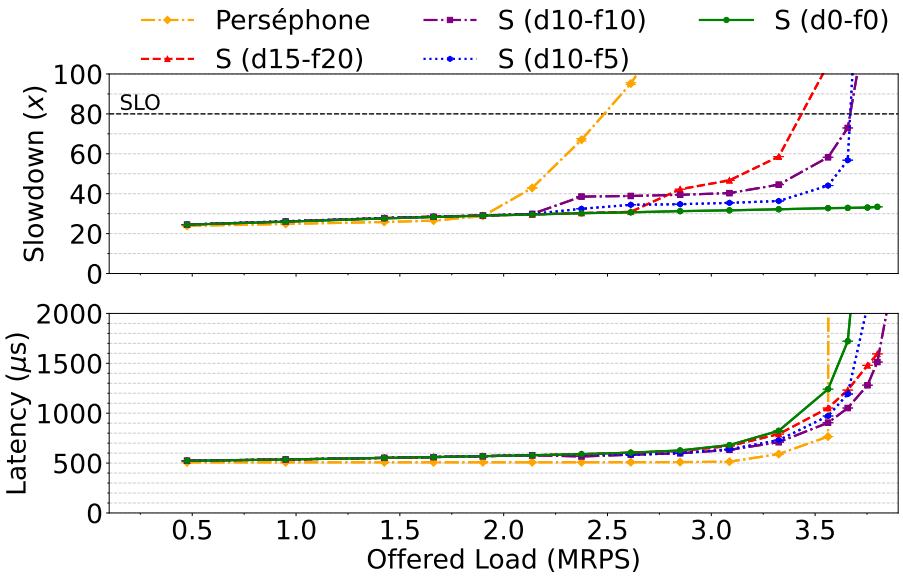


Figure 6.6: SYNERGY knobs for adjusting request processing priorities under the Extreme workload. Top: 99.9th percentile slowdown. Bottom: 99.9th percentile latency. By relaxing the slowdown SLO to 80×, configuration (d10-f10) improves tail latency by 336 μs compared to (d0-f0) at 78% of the offered load (3.65 MRPS). Furthermore, SYNERGY maintains 43% more throughput than Perséphone before violating the 50× slowdown SLO.

adjustment mechanism is triggered only under a higher offered load than the other configurations. Regarding the second parameter, a higher factor allows long requests to execute without interruption for more time, such as in configurations (d10-f5) and (d10-f10). Consequently, the resulting behavior negatively impacts slowdown, which is dominated by short requests, but improves latency, which long ones dominate. For example, suppose the SLO can tolerate a slowdown of up to 60×. In that case, the operator may choose the configuration **(d10-f5)**, which increases slowdown by 23.9, from 32.88 to 56.8, but reduces latency by 530 μs, from 1,721 μs to 1,191 μs, compared to the baseline at 3.65 MRPS, 78% of system capacity. Similarly, if the SLO can be relaxed to 80×, configuration (d10-f10) becomes a viable option, increasing slowdown by 32.64, from 32.88 to 72.94, while reducing latency by 670 μs, from 1,721 μs to 1,051 μs, under the same load conditions.

Compared to Perséphone, SYNERGY, under the (d0-f0) configuration, achieves

43% higher throughput (3.8 MRPS *vs.* 2.1 MRPS) before hitting a 50× slowdown. Perséphone has lower latency due to its run-to-completion model but suffers from HOL blocking at high load. When using the (d10-f10) configuration, SYNERGY incurs at most a 16% higher tail latency (904 μs *vs.* 765 μs) than Perséphone while still sustaining 35% more throughput (3.75 MRPS *vs.* 2.37 MRPS) before both systems reach the 80× slowdown.

6.3.3 Interrupt Methods

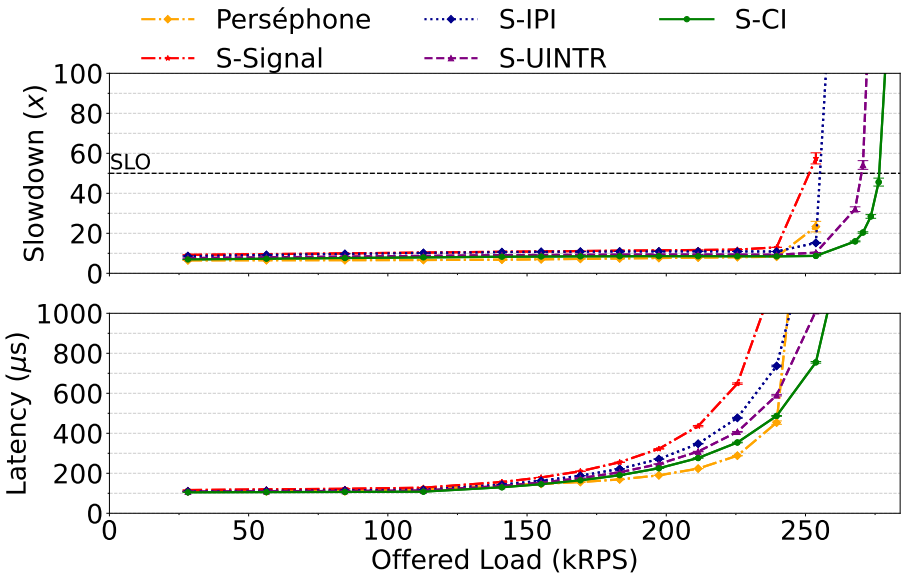


Figure 6.7: SYNERGY performance for the High workload using different interrupt methods. Top: 99.9th percentile slowdown. Bottom: 99.9th percentile latency.

This section evaluates SYNERGY using the different interrupt mechanisms presented in §5.2.3. To assess the user-level interrupt (UINTR) technology in SYNERGY, we conducted the experiments in this section on a server equipped with an Intel(R) Xeon(R) Gold 6438Y+ processor, 125 GiB of RAM, Ubuntu 22.04.05 operating system with Linux Kernel 5.15.0, and a Mellanox ConnectX-4 100 Gb/s NIC. For comparison, we also run Perséphone on the same setup.

Figure 6.7 show the results for the High workload, where SYNERGY-IPI (S-IPI)

uses the kmod_ipi module, S-Signal relies on the standard signal mechanism (*i.e.,* `tgkill`), S-UINTR uses user-level interrupts, and S-CI employs the Compiler Interrupts technique used in all previous experiments. As expected, S-Signal delivers the worst performance due to its high interrupt overhead (§5.2.3). However, because Synergy minimizes the number of preemptions to mitigate HOL blocking (§6.3.1), S-Signal performs similarly to the other mechanisms under low load. In contrast, S-IPI, S-UINTR, and S-CI improve throughput by 5%, 10%, and 13%, respectively, compared to S-Signal before violating the slowdown SLO. We observe the same pattern for the latency. Compared to Perséphone at 239 kRPS, S-CI and S-UINTR deliver 7% and 6% higher throughput, respectively, while incurring up to 6% and 23% higher tail latency.

6.3.4 Load Balance

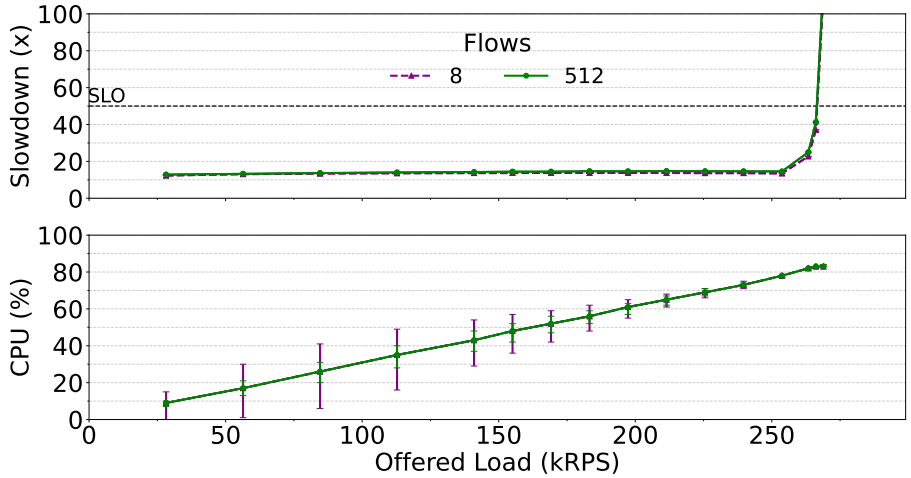


Figure 6.8: Synergy performance under the High workload with uneven flow distribution across CPU cores. The upper graph shows the 99.9th percentile of slowdown. In the bottom graph, each data point represents the average, with the bars indicating the maximum and minimum, CPU utilization across workers.

Synergy relies on multiple NIC queues to scale, with one queue assigned per worker. Most NICs use RSS [61] to distribute packets by hashing the five-tuple and mapping the result through an indirection table. However, RSS can cause significant

imbalance [3], especially under high-dispersion workloads where equalizing request counts per queue is insufficient (§6.3.1). SYNERGY addresses this load imbalance by combining a global wait queue with work stealing, allowing idle workers to process requests from overloaded peers regardless of their original queue. Figure 6.8 evaluates SYNERGY under queue imbalance by varying the number of active flows. With 512 flows, worker queues are evenly loaded (*i.e.,* receive the same number of flows); with only 8 flows, at least 6 of the 14 queues remain unused, resulting in a 42% imbalance. Despite this load imbalance caused by RSS, the top graph shows that SYNERGY maintains stable performance in both scenarios. The bottom graph shows CPU utilization. At low load, some workers remain idle under 8-flow configurations, but as the load increases, underutilized workers engage in work stealing, which increases their utilization. At 65% load (183 kRPS), the CPU usage gap across workers drops below 5%.

6.4 Multicore Scaling

We evaluate the scalability of SYNERGY by varying the number of available worker cores, aiming to assess how well it maintains performance as parallelism increases. The evaluation includes multiple worker configurations and compares SYNERGY against Perséphone, tested under both its default scheduling policy and a centralized First-Come, First-Served (cFCFS) baseline. All experiments are conducted on the same server described in Section 5.2.3, equipped with a 32-core Intel Xeon Gold 6438Y, using the ZippyDB workload. To ensure a fair comparison, we fix the effective CPU utilization at 50% for all configurations, meaning workers spend 50% of their time processing requests. The request rate (RPS) is adjusted proportionally to the number of workers: for example, 256kRPS with 8 workers and 961kRPS with 30 workers.

Figure 6.9 shows results for configurations with 8 to 30 workers. With only

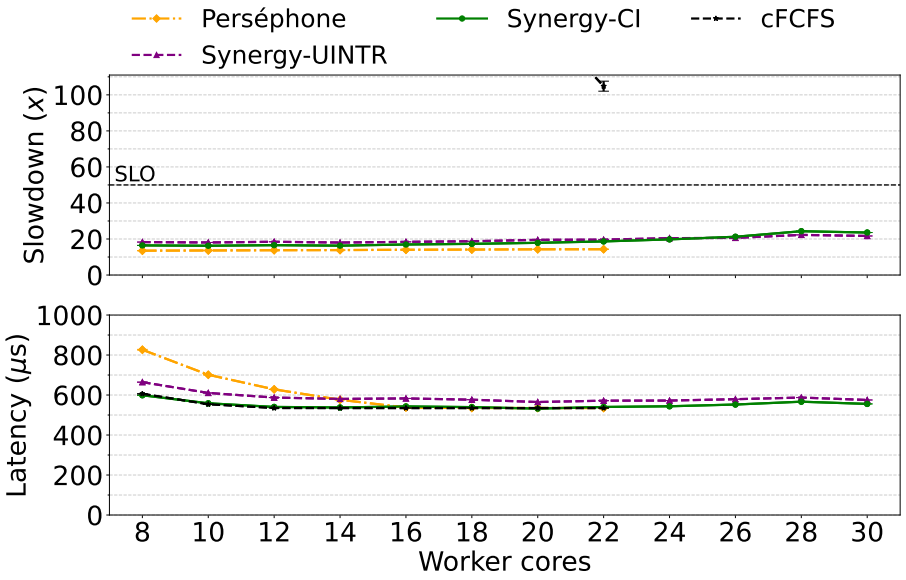


Figure 6.9: Synergy multicore scaling capacity for the ZippyDB workload. Top: 99.9th percentile slowdown. Bottom: 99.9th percentile latency. Perséphone and cFCFS dropp requests when using more than 22 workers due to dispatcher contention. In contrast, Synergy scales up to 30 workers, the maximum available on the server used for the experiment.

8 workers, all systems suffer higher tail latencies due to longer queues, especially during traffic bursts. The effect is strongest in Perséphone, which reserves some cores for short requests, leaving fewer workers for long requests, the main drivers of tail latency.

In the 8-worker setup, Synergy-CI achieves competitive slowdown (13× vs. 16× for Perséphone) while reducing tail latency by 1.38× (598 μs vs. 826 μs). Synergy-CI also outperforms cFCFS in terms of tail latency (598 μs vs. 605 μs), while cFCFS suffers from an extreme slowdown of 596×, omitted from Figure 6.9 for being out of range.

As the number of workers increases, Synergy scales more effectively than the alternatives. It maintains stable performance up to 30 workers—the maximum supported by the server—while both Perséphone and cFCFS exceed the tolerated 0.1% packet loss threshold beyond 22 workers, exposing their scalability limitations.

Chapter 7

Discussion

This chapter discusses the advantages of application-driven feedback in Synergy and how it can be effectively implemented across various workloads.

7.1 Delegating Classification to the Application

Request classification is inherently application-specific. For example, Minos [21] uses payload size as a proxy for service time. Key-value stores like Redis [57], RocksDB [60], and LevelDB [41] can distinguish request types based on operations (*e.g.,* `GET` vs. `SCAN`). In contrast, systems like Apache Lucene [44] may only estimate request cost after partial processing, since ranking popular terms is significantly more expensive [18].

Prior systems [20,36] rely on external classification based on protocol knowledge. This design introduces three key drawbacks: *(i)* redundancy, as classification still occurs internally within the application; *(ii)* limited scalability, as classification is centralized and serial; *(iii)* reduced accuracy, as external logic cannot account for runtime state or encrypted content. Preemptive schedulers [33, 36, 45], on the other hand, rely on fixed-time quanta to implicitly classify requests: any request exceeding the quantum is implicitly treated as long. While this approach enables simple

prioritization, it suffers from unnecessary preemptions and cannot avoid delaying short requests when a large quantum is used.

In contrast, **proactive, application-level classification** offers two main advantages: *(i)* it reduces timer management overhead by enabling preemption only when needed; and *(ii)* it enables job-aware preemption, where only long-running requests are interrupted, preserving the responsiveness of short ones. SYNERGY uses lightweight feedback from the application to enable this proactive classification. This inline mechanism avoids protocol-specific logic, supports parallelism across worker threads, and generalizes across applications.

7.2 Timeliness and Practical Implementation

Feedback is only useful if delivered early enough to affect scheduling decisions. The sooner a request is identified as long, the more effectively the system can mitigate HOL blocking. In many systems, early classification is feasible. For example: *(i)* in key-value stores, request types are known immediately after parsing; *(ii)* in pipelined systems like Redis [58], feedback can reflect the number of pipelined operations; *(iii)* in search engines [18], lightweight cost estimation (*e.g.,* based on query popularity) can be used.

In modular applications, tools like LDB [18] and CoverUP [54] can help automate feedback insertion. LDB identifies latency-critical functions, while CoverUP uses LLMs to improve test coverage. These tools could be extended to detect expensive operations and automatically instrument them with feedback logic. While this is a promising research direction, we leave this automation for future work.

7.3 Benefits of Application Feedback

Application-level feedback provides fine-grained, low-latency control over scheduling decisions. Below, we summarize the key benefits:

Low Overhead. Feedback incurs minimal overhead for two reasons: *(i)* classification is already performed by the application, so the feedback simply communicates the result; and *(ii)* only long requests require feedback, which are typically a small fraction of total requests. Even when a timer is activated, SYNERGY 's timer implementation is lightweight (< 40 ns in our tests). Without feedback, timers would need to be configured for every request, increasing overhead.

Scheduler Generalization. Delegating classification to the application decouples the scheduler from protocol-specific logic or communication patterns. This design improves maintainability, eliminates the need for per-application tuning, and enables reuse across diverse workloads.

Job-Aware Preemption. By default, the scheduler assumes all requests are short unless told otherwise. This lets short requests run to completion without preemption. Consequently, small quanta can be used to improve responsiveness without hurting short-request latency.

Dynamic Quantum Sizing. Feedback allows the quantum to adapt dynamically to workload characteristics. This removes the need for manual tuning and enables real-time responsiveness to changing conditions.

Resilience to Workload Variability. Application feedback provides timely signals that help the scheduler adapt to workload changes, such as bursts or shifts in request mix, improving robustness under diverse and unpredictable conditions.

7.4 Dealing with Multiple Request Types

Priority queues are a well-established technique for handling multiple request types, as used in systems like Perséphone [20] and Shinjuku [36]. Extending Synergy to support multiple queues with explicit priorities is straightforward and orthogonal to our core contributions. Our focus is on showing that even without centralized queues, decentralized scheduling with lightweight feedback can achieve high performance and low tail latency.

Chapter 8

Conclusion

This work explores the design space of systems for processing datacenter applications and introduces novel, previously unexplored optimizations. We consolidate these ideas into the implementation of a system called Synergy, which achieves both high scalability and low latency while effectively mitigating Head-of-Line (HOL) blocking in workloads with significant service time variability.

The main contributions of this work are: *(i)* a novel technique that enables direct cooperation between the application and the scheduler, significantly reducing HOL blocking and associated overhead. *(ii)* a new strategy to minimize interrupt frequency, improving throughput while still addressing HOL blocking. *(iii)* a comprehensive evaluation of several recent state-of-the-art systems that use interrupts and core reservation to mitigate HOL blocking. *(iv)* a system, called Synergy, that integrates these optimizations into a cohesive design, resulting in a more scalable architecture than prior solutions without sacrificing latency.

A key innovation introduced in this work is the use of application-level feedback for request classification. This approach allows the scheduler to adapt across a broad range of applications without requiring application-specific logic. The feedback mechanism enables precise scheduling decisions and unlocks several performance optimizations, such as automatic quantum adjustment and job-aware preemptions.

Experimental results show that SYNERGY can process significantly more requests than prior state-of-the-art systems while consistently meeting strict service-level objectives (SLOs) for short requests—on the order of a few microseconds. It also reduces tail latency for long-running requests compared to existing preemptive scheduling techniques, highlighting its effectiveness across a variety of conditions.

In conclusion, this work shows that tight cooperation between applications and the scheduler can significantly improve performance in latency-critical environments. The techniques introduced here pave the way for more adaptive, efficient, and scalable systems in modern datacenters.

8.1 Future Work

A broader evaluation of the feedback technique across a wider range of real-world applications could offer deeper insights into its effectiveness under diverse workload characteristics. While this work demonstrates the benefits of application-driven feedback using LevelDB [41], many widely adopted systems, such as Redis [57], RocksDB [60], Apache Lucene [44], and Memcached [47], present distinct execution patterns and resource profiles that could serve as valuable benchmarks for further validation.

Another promising avenue involves reducing the manual effort required to adopt feedback techniques and employ automatic instrumentation. As discussed in Section 7.2, automating the integration of application feedback into scheduling logic could broaden the applicability of the approach and lower the barrier to adoption.

Finally, extending SYNERGY using emerging hardware features to meet the increasing demands of high-throughput environments. As network interface cards (NICs) continue to scale in bandwidth, sustaining line-rate performance requires highly efficient request processing. Hardware-assisted technologies such as Intel's Dynamic Load Balancer (DLB) [32], which provides efficient queue management,

could be leveraged to implement SYNERGY 's wait queue with lower overhead, potentially boosting both performance and scalability. Moreover, because SYNERGY employs preemptive scheduling, advances in interrupt delivery and handling, whether through hardware or OS-level enhancements, can directly improve its responsiveness. These future directions can build upon the codebase developed in this work to explore enhanced implementations and broader deployment scenarios.

Bibliography

[1] D. Ardelean, A. Diwan, and C. Erdman. Performance Analysis of Cloud Applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, 2018.

[2] B. Aydogmus, L. Guo, D. Zuberi, T. Garfinkel, D. Tullsen, A. Ousterhout, and K. Taram. Extended User Interrupts (xUI): Fast and Flexible Notification without Polling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 373–389, 2025.

[3] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić. RSS++ Load and State-aware Receive Side Scaling. In *Proceedings of the 15th international conference on emerging networking experiments and technologies*, pages 318–333, 2019.

[4] The Barrelfish OS. `https://barrelfish.org/`.

[5] L. A. Barroso, U. Hölzle, and P. Ranganathan. *The Datacenter as a Computer: Designing Warehouse-scale Machines*. Springer Nature, 2019.

[6] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged {CPU} Features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, 2012.

[7] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. {IX}: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, 2014.

[8] M. S. Berghetti, F. B. Carvalho, and R. A. Ferreira. AFP: A Feedback-Driven Microservices Request Scheduler. In *Proc. of SBRC 2024*, pages 1148–1161, 2024.

[9] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Lightweight Preemptible Functions. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 465–477, 2020.

[10] Q. Cai et al. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.

[11] Q. Cai et al. Towards μs Tail Latency and Terabit Ethernet: Disaggregating the Host Network Stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 767–779, 2022.

[12] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, Modeling, and Benchmarking {RocksDB}{Key-Value} Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[13] F. B. Carvalho and R. A. Ferreira. Scaling Stateful Network Services on Multicore Architectures. In *Proceedings of the IEEE Network Operations and Management Symposium (NOMS 2025)*, pages 01–08, 2025.

[14] F. B. Carvalho, R. A. Ferreira, Í. Cunha, M. A. Vieira, and M. K. Ramanathan. Dyssect: Dynamic Scaling of Stateful Network Functions. In *IEEE INFOCOM*

2022-IEEE Conference on Computer Communications, pages 1529–1538. IEEE, 2022.

[15] F. B. Carvalho, R. A. Ferreira, Í. Cunha, M. A. Vieira, and M. K. Ramanathan. State Disaggregation for Dynamic Scaling of Network Functions. *IEEE/ACM Transactions on Networking*, 32(1):81–95, 2023.

[16] Coroutines, 2025. `https://en.cppreference.com/w/cpp/language/coroutines`.

[17] S. Chen et al. Parties: Qos-aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[18] I. Cho, S. J. Park, A. Saeed, M. Alizadeh, and A. Belay. {LDB}: An Efficient Latency Profiling Tool for Multithreaded Applications. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1497–1510, 2024.

[19] J. Dean and L. A. Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[20] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When Idling is Ideal: Optimizing Tail-latency for Heavy-tailed Datacenter Workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 621–637, 2021.

[21] D. Didona and W. Zwaenepoel. Size-Aware Sharding for Improving Tail Latencies in In-Memory Key-Value Stores. In *USENIX NSDI'19*, 2019.

[22] U. Drepper. Elf Handling for Thread-local Storage. Technical report, Technical report, Red Hat, Inc., 2003. URL http://people.redhat.com, 2005.

[23] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, et al. The Design and Operation of {CloudLab}. In *2019 USENIX annual technical conference (USENIX ATC 19)*, pages 1–14, 2019.

[24] L. Foundation. The Linux Kernel, 2024. `https://kernel.org/`.

[25] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[26] GNU. GNU Pth - The GNU Portable Threads, 2025. `https://www.gnu.org/software/pth/`.

[27] Proposal: Non-cooperative Goroutine Preemption, 2025. `https://go.googlesource.com/proposal/+/master/design/24543-non-cooperative-preemption.md`.

[28] S. Han et al. PacketShader: A GPU-Accelerated Software Router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[29] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. *ACM Sigplan Notices*, 50(4):161–175, 2015.

[30] K. Huang, J. Zhou, Z. Zhao, D. Xie, and T. Wang. Low-Latency Transaction Scheduling via Userspace Interrupts: Why Wait or Yield When You Can Preempt? *Proceedings of the ACM on Management of Data*, 3(3):1–25, 2025.

[31] Intel. Data Plane Development Kit, 2024. `https://www.dpdk.org/`.

[32] Intel Dynamic Load Balancer, 2025. `https://www.intel.com/content/www/us/en/download/686372/intel-dynamic-load-balancer.html`.

[33] R. Iyer, M. Unal, M. Kogias, and G. Candea. Achieving Microsecond-scale Tail Latency Efficiently with Approximate Optimal Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 466–481, 2023.

[34] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. {mTCP}: A Highly Scalable User-level {TCP} Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.

[35] Y. Jia, K. Tian, Y. You, Y. Chen, and K. Chen. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 265–279, 2024.

[36] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive Scheduling for {μsecond-scale} Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[37] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High Performance Packet Processing with Flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–81, 2016.

[38] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[39] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[40] B. H. Leitao. Tuning 10Gb Network Cards on Linux. In *Proceedings of the 2009 Linux Symposium*, pages 169–185. Citeseer, 2009.

[41] LevelDB. LevelDB, 2024. `https://github.com/google/leveldb`.

[42] Y. Li, N. Lazarev, D. Koufaty, T. Yin, A. Anderson, Z. Zhang, G. E. Suh, K. Kaffes, and C. Delimitrou. Libpreemptible: Enabling Fast, Adaptive, and Hardware-assisted User-space Scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 922–936. IEEE, 2024.

[43] Libibverbs Library. `https://www.ibm.com/docs/en/aix/7.2?topic=ofed-libibverbs-library`.

[44] Apache Lucene, 2024. `https://lucene.apache.org/`.

[45] Z. Luo, S. Son, D. Bali, E. Amaro, A. Ousterhout, S. Ratnasamy, and S. Shenker. Efficient Microsecond-scale Blind Scheduling with Tiny Quanta. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 305–319, 2024.

[46] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy. Efficient Scheduling Policies for {Microsecond-Scale} Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, 2022.

[47] Memcached. Memcached - A Distributed Memory Object Caching System, 2023. `https://memcached.org/`.

[48] Microsoft. Fibers, 2025. `https://learn.microsoft.com/en-us/windows/win32/procthread/fibers`.

[49] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling Memcache at Facebook. In *10th*

USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pages 385–398, 2013.

[50] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High {CPU} Efficiency for Latency-Sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[51] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The RAMCloud Storage System. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[52] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 337–350, 2012.

[53] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[54] J. A. Pizzorno and E. D. Berger. CoverUp: Coverage-Guided LLM-Based Test Generation. *arXiv preprint arXiv:2403.16218*, 2024.

[55] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[56] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne:{Core-Aware} Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[57] Redis Ltd. Redis, 2023. `https://redis.io/`.

[58] Redis Pipelining, 2025. https://redis.io/docs/latest/develop/use/pipelining/.

[59] L. Rizzo. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 101–112, USA, 2012. USENIX Association.

[60] RocksDB. RocksDB, 2023. http://rocksdb.org/.

[61] RSS. Introduction to Receive Side Scaling, 2023. https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[62] R. Schöne, D. Molka, and M. Werner. Wake-up Latencies for Processor Idle States on Current x86 Processors. *Computer Science-Research and Development*, 30:219–227, 2015.

[63] S. Shiina, S. Iwasaki, K. Taura, and P. Balaji. Lightweight Preemptive User-level Threads. In *Proceedings of the 26th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 374–388, 2021.

[64] Sohil Mehta. x86 User Interrupts Support, 2023. https://lwn.net/Articles/869140/.

[65] Overview of Single Root I/O Virtualization (SR-IOV). https://learn.microsoft.com/en-us/windows-hardware/drivers/network/overview-of-single-root-i-o-virtualization--sr-iov-/.

[66] System V Application Binary Interface, 2025. https://gitlab.com/x86-psABIs/x86-64-ABI.

[67] A. Wierman and B. Zwart. Is Tail-Optimal Scheduling Possible? *Operations Research*, 2012.

[68] W. Wu et al. The Performance Analysis of Linux Networking–Packet Receiving. *Computer Communications*, 30(5):1044–1057, 2007.

[69] Y. Yang, Z. Huang, A. Kaufmann, and J. Li. Protected Data Plane OS Using Memory Protection Keys and Lightweight Activation, 2023.

[70] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, et al. The Demikernel Datapath os Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.

[71] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi. {CRISP}: Critical Path Analysis of {Large-Scale} Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

Appendix A

Publications

We published the initial performance results of SYNERGY, based on simulations, in the Brazilian Symposium on Computer Networks and Distributed Systems (SBRC) 2024. After implementing SYNERGY with several new features, such as different interrupt mechanisms, we submitted its full design, implementation, and performance evaluation to a top systems conference, as listed below.

- **BERGHETTI, M. S.**; CARVALHO, F. B.; FERREIRA, R. A. **AFP: Um Escalonador de Requisições de Microsserviços Guiado por Feedback**. In Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), Niterói, RJ, Brasil, p. 1134-1147, 2024 [8] (In Portuguese).

- **BERGHETTI, M. S.**; CARVALHO, F. B.; FERREIRA, R. A. **Achieving High Throughput and Low Tail Latency in Microsecond-Scale Datacenter Applications with Synergy**. Submitted for publication.