Synergy: A Fast and Scalable Feedback-Driven Scheduler for Datacenter Applications

Mayco S. Berghetti
UFMS
Campo Grande, MS, Brazil
mayco.berghetti@ufms.br

Fabrício B. Carvalho
UFMT
Cuiabá, MT, Brazil
fabricio.carvalho@ufmt.br

Ronaldo A. Ferreira
UFMS
Campo Grande, MS, Brazil
raf@facom.ufms.br

***Abstract*—Microsecond-scale datacenter applications demand strict latency guarantees while operating under high load and service time variability. However, existing approaches to mitigate Head-of-Line (HOL) blocking, such as centralized dispatching, fine-grained preemption, and resource reservation, face fundamental scalability limitations. This paper introduces Synergy, a cooperative, application-aware scheduling system that uses direct feedback from applications to prioritize short requests, dynamically adapt scheduling parameters, and avoid 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. Implemented as a user-space library on top of DPDK, Synergy outperforms state-of-the-art systems. In high-dispersion workloads, Synergy achieves up to 43% higher throughput while meeting microsecond-scale service level objectives.**

I. 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) [1]. These applications are composed of complex, latency-sensitive workflows where every microsecond matters [2]. The challenge lies not only in processing large numbers of concurrent requests but also in ensuring that even the slightest delays are minimized [3].

At microsecond timescales, traditional software architectures struggle to keep up with the demands of high-throughput, low-latency workloads. Delays are compounded by factors such as contention for CPU cores [4], memory bandwidth [5], and bursts of requests that introduce queuing and scheduling inefficiencies [3]. To make matters worse, datacenter workloads often exhibit service times with high dispersion, where a mix of extremely short and long requests must coexist [6], [7], [8], [9], [10], [11]. Short requests, taking just a few microseconds, are often delayed behind longer ones leading to higher latencies, especially at the tail, an issue known as *Head-of-Line (HOL) blocking* [3].

To address the HOL blocking problem and bound tail latency, recent research has explored kernel-bypass systems with a variety of scheduling strategies [12], [13], [6], [14], [4], [1], [5], [15], [16], including employing centralized dispatchers for load balancing [8], [9], preempting long requests to prioritize short ones [8], [10], [17], [11], [16], and intra-server resource reservation [9]. Unfortunately, these approaches often scale poorly and force servers to run at low utilization (*e.g.*, below 40%) to meet strict SLOs [1].

Each of these strategies faces fundamental limitations that reduce their effectiveness at scale. Centralized dispatchers [8], [4], [9], [10], despite effectively distributing load, become bottlenecks under high load, which limits throughput and causes servers to remain underutilized. Fine-grained preemptive schedulers [8], [10], [17] introduce substantial overhead due to indiscriminate context switching and interrupt handling, and even optimized variants [10], [11] struggle with workloads exhibiting high service-time variance, leading to poor cache performance and increased CPU costs. Resource reservation strategies [9] rely on external classifiers to distinguish request types, which adds redundant classification overhead and often leads to incorrect predictions when service times depend on dynamic application state, thus calling for more specialized and 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.

This work introduces Synergy, a system that cooperates closely with applications to scale on multicore architectures and address fundamental limitations of existing designs. By incorporating direct application feedback, Synergy differentiates requests by expected service time, prioritizes short ones, and mitigates HOL blocking without sacrificing scalability. Synergy combines distributed scheduling, dynamic load balancing, and job-aware preemption to meet μs-scale SLOs even under high-load and high-variance conditions that are typical of datacenter workloads.

Synergy uses the NIC to distribute requests across per-worker queues, eliminating the need for a dedicated core acting as a centralized dispatcher. To handle load imbalance,

ISBN 978-3-903176-82-9

it employs work stealing [14], which enables underutilized workers to pull tasks from overloaded ones. Combined with efficient lock-free data structures [18], this design preserves the scalability of decentralized dispatch while maintaining high utilization. By eliminating a centralized dispatcher—a known bottleneck in prior systems [8], [9], [4], [10]—Synergy avoids a critical scalability constraint.

In Synergy, applications classify requests and provide feedback to the scheduler, enabling differentiated handling of short and long requests. Short requests run to completion to minimize latency, whereas long requests can be preempted and resumed later to reduce HOL blocking. Preempted requests are placed in a common wait queue rather than returned to their original worker's queues, which allows them to be redistributed more effectively across workers than with queue-length-based strategies, such as work stealing, alone. This strategy prevents long requests from accumulating unevenly across worker queues and enables idle workers to resume deferred computation, thereby enhancing load balance, responsiveness, and overall resource utilization.

Synergy also computes the scheduling quantum dynamically from observed service times and operator-defined parameters, avoiding the drawbacks of fixed large quanta that can penalize short requests [8], [11], [15]. Moreover, it employs job- and load-aware conditional preemption instead of purely time-based preemption, which avoids unnecessary context switches when queues are empty and reduces overhead.

Finally, by delegating request classification to the application, Synergy eliminates the need for a centralized classifier [9] and supports more flexible and accurate classification. This approach removes redundant classification overhead [9] and 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 [19] or the popularity of a search term [20].

We implement Synergy as a libOS using DPDK to bypass the Linux kernel and compare it against Shinjuku [8], Perséphone [9], Concord [10], and Tiny Quanta [11]. Across diverse workloads, Synergy significantly improves throughput while meeting microsecond-scale latency SLOs. In the Extreme workload (§V-A), it increases throughput by 24–43% over prior systems and mitigates HOL blocking up to 81% system load. We further present a comprehensive evaluation, including ablation studies, sensitivity analyses, load-imbalance experiments, and scalability tests across increasing core counts. The source code of Synergy and the scripts for reproducing our results are available at https://github.com/Synergy-repo/synergy.

II. Synergy Design

Figure 1 presents an overview of Synergy and its main components. Synergy receives incoming requests through multiple NIC queues (*e.g.,* using RSS [21], Flow Director [22], or programmable NICs [23]), with each NIC queue assigned to a single worker to avoid concurrent access. Synergy also maintains a global *wait queue* for preempted

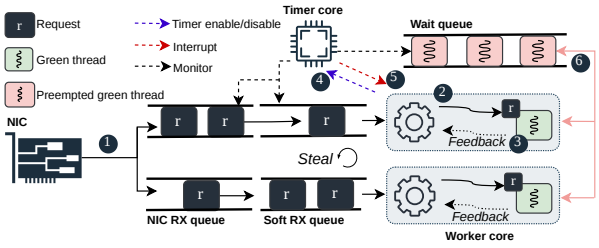

Fig. 1: Synergy overview. Solid lines represent request path, dashed lines indicate timer core queue monitoring, and interactions between timer core and worker cores.

requests. For simplicity, we describe the system using only two request types—short and long—but it can be extended to support additional types using multiple wait queues with 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 [8], [10], [17]. 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 modify a parameter to increase the CPU time allocated to long requests and reduce their total time in the system. Section V-B2 evaluates the impact of this parameter.

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 are waiting in the wait queue for more than a specified threshold.

In summary, as shown in Figure 1, Synergy operates as follows: incoming requests are distributed across multiple queues ❶. Each worker schedules a new or previously-preempted request to the application ❷. If a new request is classified as long, the application notifies Synergy ❸, triggering a timer on the timer core ❹. When needed, the timer core interrupts workers to mitigate HOL blocking ❺, prompting the worker to move the running request to the wait queue ❻.

A. Worker Core

After initializing (*e.g.,* loading a database), the application calls `synergy()` to set up workers, queues, and timers (Algorithm 1, Line 12), passing a callback function to

Algorithm 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)
```

Algorithm 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
```

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 2).

The application invokes `synergy_feedback_start()` at the beginning (Line 7) to notify Synergy that it is processing a long request, and, when it finishes, it invokes `synergy_feedback_finished()` (Line 9).

*1) **Request Selection**:* Algorithm 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 3). This mechanism bounds the delay for long requests that have been waiting to resume.

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, one of the well-established techniques for workload balancing [24], [14], 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()`.

*2) **Quantum Sizing**:* Choosing an appropriate value for the quantum is critical for the performance of a preemptive system [25]. A quantum that is too small increases context-switching overhead, while a large one can delay short requests, leading to HOL blocking. Prior systems [8], [10], [11] 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 by each worker to avoid synchronization. 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. Recall that short requests are never preempted.

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 increase the responsiveness of short requests. Although workloads with extremely short requests (as low as 500 ns [14], [9]) 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.

B. Timer Core

The timer core in Synergy is responsible for two key tasks: *(i)* monitoring the wait queue and *(ii)* interrupting workers when necessary. These tasks, outlined in Algorithm 3, are lightweight, off the critical processing path, and, therefore, do not create a performance bottleneck in the timer core.

*1) **Wait Queue Monitoring**:* The timer core periodically checks the wait queue to ensure timely processing of preempted requests. Instead of tracking per-request timestamps, which would add significant overhead to workers, Synergy adopts a lightweight strategy. At each interval, defined by the user-configurable `THRESH_WQD`, the timer core compares the current consumer index with the producer index saved during the last check. If the consumer index has not

Algorithm 3 Timer Core

```
factor ← QUANTUM_FACTOR
check_wait_queue ← false
last_state ← false
loop
    state ← Q^wait.is_congested(THRESH_WQD)
    if state ≠ last_state then
        if state = true then
            factor ← CONGESTED_QUANTUM_FACTOR
        else
            factor ← QUANTUM_FACTOR
        check_wait_queue ← state
        last_state ← state
    for all w ∈ workers do
        if timer_w expired then
            if Q_w^RX ≠ ∅ then
                interrupt w to process new request
            else
                renew timer_w deadline
```

advanced, this condition 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. It also applies the operator-defined CONGESTED_QUANTUM_FACTOR to increase the quantum, allowing long requests to run longer and helping drain the wait queue faster.

This design ensures progress for preempted requests while giving operators control over the trade-off between short-request responsiveness and long-request throughput. For instance, if long requests violate their SLOs under heavy load, increasing CONGESTED_QUANTUM_FACTOR allows them to complete sooner without affecting short-request performance under lighter load. Consequently, Synergy adapts to dynamic workloads while maintaining low latency and high efficiency.

2) ***Worker Interrupt****:* The timer core in Synergy coordinates worker preemptions to mitigate HOL blocking while avoiding unnecessary interrupts. Rather than blindly preempting when a fixed quantum expires, Synergy takes into account the local state of the worker. If no new requests are pending in the worker's queue, the current request continues running to avoid wasteful context switches. This selective and job-aware strategy improves efficiency over prior systems that rely solely on quantum timers [8], [10], [11].

Unlike hardware timer-based approaches [17], [15], 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 [26], [27], Inter-Processor Interrupts (IPIs) [28], [8], user-level solutions like Intel's User Interrupts (UINTR) [29], [30], [15], and cooperative yielding via compiler instrumentation [10], [11]. This flexibility allows Synergy to operate efficiently across diverse runtime environments and hardware platforms, making it well-suited for latency-sensitive applications.

III. 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. The current implementation consists of 1,688 lines of C code, plus minimal assembly for green-thread context switching and interrupt handling. The optional kernel modules kmod_ipi and kmod_uintr (§V-B3) contain 186 and 209 lines of C code, respectively.

A. 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 attempt to resume long requests from a shared wait queue implemented as a lock-free multi-producer, multi-consumer ring. This queue scales well because: *(i)* it is lock-free; *(ii)* enqueues occur only during preemption, which Synergy minimizes; *(iii)* only idle workers dequeue when uncongested; and *(iv)* it contains only long requests. If no work is found locally or in the wait queue, workers attempt work stealing (§II).

B. Green Threads

Synergy implements green threads using a minimal context that stores callee-saved registers (per the System V AMD64 ABI) plus registers RIP and RSP. Context switches copy only 64 bytes (one cache line), making them lightweight.

Unlike systems that allocate one green thread per request [32], [8], [10], [4], Synergy reuses threads across requests, creating a new one only upon preemption. When resuming from the wait queue, it switches directly to the target green thread, bypassing the main context. Preempted threads are added to a per-worker list for deferred batch deallocation, typically when returning to the main thread after an interrupt. Threads are allocated from DPDK mempools with per-worker caches to minimize locking.

C. Preemption Mechanisms

Synergy supports multiple preemption mechanisms, including Compiler Interrupts and IPI-based approaches.

Compiler Interrupts (CI) CI inserts yield points at compile time, enabling cooperative preemption without arbitrary interrupt delivery. This approach avoids saving full processor state and can outperform asynchronous methods [10], [11]. Synergy's timer core sets the yield flag that is periodically checked by workers. CI incurs low overhead, though tight-loop yield points may add up to 20% overhead [11].

TABLE I: Overhead (ns) for interrupt methods: 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

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

- **signal:** Standard POSIX signals, which incur high overhead due to system calls and generic kernel routines.
- **kmod_ipi:** A custom kernel module that targets CPUs directly, bypasses unnecessary kernel logic, and transfers control quickly to a user-level handler.
- **uintr (User-level Interrupts):** A hardware feature on Intel Sapphire Rapids CPUs that enables fully user-space interrupt delivery. As it is not yet supported in mainline Linux, we implemented a supporting kernel module.

Table I presents microbenchmark results on an Intel Xeon Gold 6438Y+ system, with 32 cores, running Linux 5.15.0. The evaluation measures the latencies in the four phases illustrated in Figure 2: **send** ($st1 - st0$), **propagation** ($st2 - st0$), **receive** ($rt1 - rt0$), and **return** ($rt3 - rt2$).

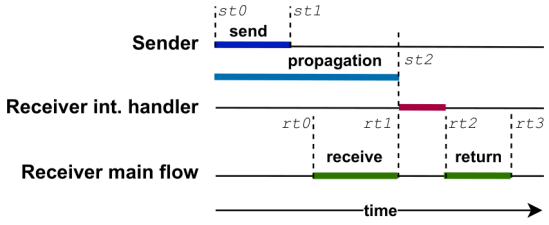

Fig. 2: Interrupt delivery stages corresponding to Table I.

IV. Experimental Methodology and Setup

This section describes our traffic generator, workloads, and testbed configuration for evaluating Synergy against state-of-the-art systems [8], [9], [10], [11].

We evaluate Synergy using both synthetic and real workloads. Table II summarizes the four workloads considered. High, Extreme, and ZippyDB are synthetic workloads built on a configurable application that consumes CPU cycles proportional to each request's target service time. They exhibit high service-time variability and are widely used in prior work [8], [10], [11]. We also include a real workload based on LevelDB [33], a key-value store executing actual `GET` and `SCAN` operations. The service times reported in Table II correspond to averages of 1M requests per operation.

Traffic is generated by an open-loop DPDK-based [31] (v23.11) load generator that bypasses the OS stack to minimize networking overhead. The client sends 128-byte UDP requests over 512 independent flows, with inter-arrival times drawn from an exponential distribution with mean N, corresponding to the target request rate (RPS). This pattern reflects typical datacenter traffic [7]. The client uses three dedicated cores:

TABLE II: Evaluated workloads.

Workload	Request types	Service times (μs)	Ratios (%)
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

one for transmission (TX) and two for reception (RX). The TX core timestamps outgoing requests. The first RX core polls the NIC, timestamps replies, and forwards them to the second RX core, which computes the end-to-end latency. This separation reduces interference and improves measurement accuracy.

Each experiment runs for 60 seconds, with the first 10% of responses discarded as warm-up. We report averages of 10 independent runs with 95% confidence intervals. Since packet loss can affect measured throughput at the client, we report results as a function of offered load rather than throughput. This approach ensures consistent comparisons between systems while tolerating small packet losses. Runs with more than 0.1% packet loss are discarded.

Our primary metric is the 99.9th percentile slowdown, defined as the total request latency divided by its service time. By normalizing for request size, slowdown enables consistent SLO comparisons across systems and is well suited to high-variance workloads. It is particularly sensitive to short requests, which are most affected by queuing delays, and is widely used in prior work [10], [9], [8]. For completeness, we also report the 99.9th percentile tail latency, which captures worst-case behavior but is dominated by long requests.

A. Setup

Unless otherwise stated, experiments run on two CloudLab [34] c6420 nodes connected by a 10 Gbps link. Each node has an Intel Xeon Gold 6142 (16 cores @2.60 GHz), 376 GB RAM, and an Intel X710 10GbE NIC. Turbo Boost is disabled, and all cores are isolated via `isolcpus`. DPDK uses 8,192 2 MB hugepages, and threads are pinned with EAL core masks and `taskset` to preserve NUMA locality and avoid contention. The average RTT is 10 μs. Latency is measured at the client to avoid clock synchronization issues. We use Ubuntu 18.04 with Linux 4.4.185 to ensure compatibility with Dune [28], which is required by Shinjuku.

V. Performance Evaluation

This section compares Synergy with prior systems using synthetic and real workloads. We report tail slowdown and latency under varying loads and configurations, showing the maximum throughput each system sustains without violating a fixed SLO. We also present an ablation study of Synergy's key components, along with analyses of parameter sensitivity, interrupt mechanisms, resilience to load imbalance, and scalability across core counts.

A. Synergy vs. Preemptive Systems

We compare Synergy with leading preemptive systems: Shinjuku [8], Concord [10], and Tiny Quanta (TQ) [11], which

mitigate HOL blocking via interrupts. Shinjuku reduces IPI overhead using hardware virtualization, whereas Concord and TQ rely on Compiler Interrupts (CI). For fairness, we enable CI wherever possible and modify Shinjuku to use Concord's CI mechanism (Shinjuku-CI).

Concord, Shinjuku, and Shinjuku-CI use a fixed 5 μs quantum, as in the original Shinjuku design. Reduing this value degrades performance, as shown in [10]. TQ uses a 2 μs quantum, reported as optimal by its authors. In contrast, Synergy employs a dynamic quantum with `QUANTUM_FACTOR=2` and `THRESH_WQD=0`, which ignores wait-queue congestion and prioritizes short requests while serving long ones on a best-effort basis. All systems run with 14 dedicated worker cores.

*1) **Synthetic Workload**:* **High:** Figure 3 shows 99.9th percentile slowdown (left) and tail latency (right) 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 a 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.

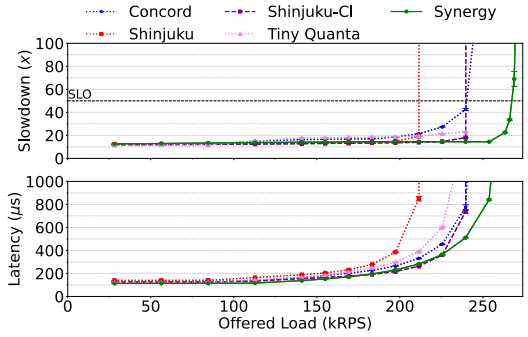

Fig. 3: 99.9th percentile of both slowdown (top) and latency (bottom) in the High workload.

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.

Extreme: Figure 4 shows the results for the Extreme workload, the most skewed workload in Table II. 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.

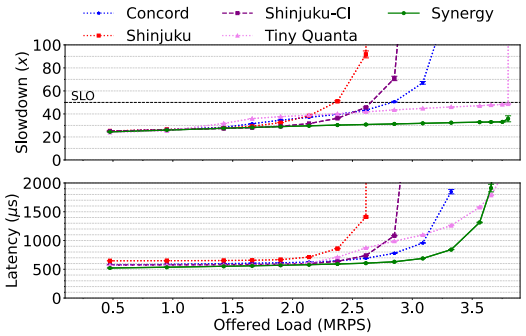

Fig. 4: 99.9th percentile of both slowdown (top) and latency (bottom) in the Extreme workload.

Synergy achieves the best results by combining distributed queues, efficient load balancing, and low-overhead preemption. It reaches 3.79 MRPS before violating the SLO, which is 37%, 31%, and 24% higher throughput than Shinjuku (2.37), Shinjuku-CI (2.61), and Concord (2.84). Compared to TQ, Synergy reduces slowdown from 48.62 to 35.77 (1.35×) at 3.79 MRPS (81% load). It also delivers the lowest latency up to 3.56 MRPS (78% load). By serving long requests best-effort, Synergy consistently prioritizes short ones. Under heavier load, operators can further reduce latency by adjusting input parameters as needed.

ZippyDB: Figure 5 shows the results for the ZippyDB workload, which reflects the request distribution observed in production environments [7]. This workload strikes a balance between the previously evaluated High and Extreme workloads, combining a 1,000× service time dispersion (as in Extreme) with a higher proportion of long requests (as in High). Consequently, it tests the system's ability to sustain high throughput and the effectiveness of its interrupt mechanisms.

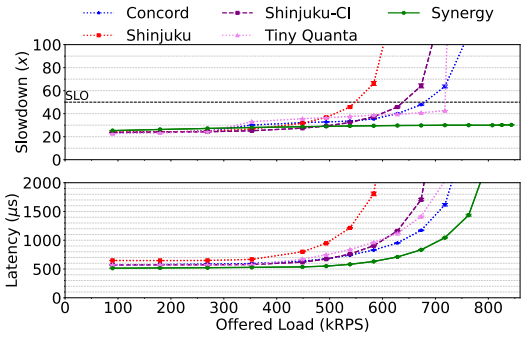

Fig. 5: 99.9th percentile of both slowdown (top) and latency (bottom) in the ZippyDB workload.

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. 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.

*2) **Real Application – LevelDB**:* We compare Synergy with Shinjuku-CI and Concord using LevelDB (v1.23). All systems are compiled with Concord's LLVM pass [35] to ensure identical instruction streams. The database runs on the RAM-backed file system `tmpfs`, preloaded with 1,000 entries; GET selects random keys, and SCAN iterates over all entries (50/50 mix). Figure 6 shows the results. As in High, Synergy outperforms prior systems, but here lock contention further differentiates designs. When long requests fail to acquire a lock outside critical sections, Synergy yields, allowing others to proceed; Concord and Shinjuku-CI block. Synergy achieves 284 kRPS before violating the SLO, which is 26% and 21% higher than Shinjuku-CI (209) and Concord (224). At 224 kRPS (76% load), Synergy reduces slowdown by 2.92× (48.09→16.44) and tail latency by 1.56× (841→539 μs) relative to Concord.

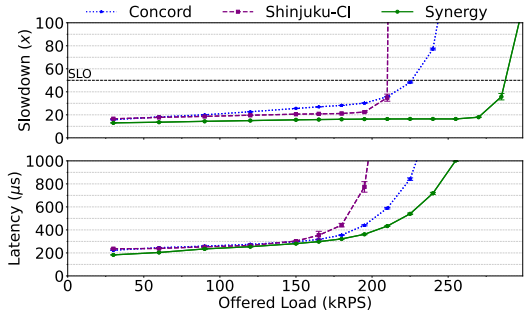

Fig. 6: 99.9th percentile of both slowdown (top) and latency (bottom) in the LevelDB application with 1,000 entries under 50% GET and 50% SCAN workload.

B. Ablation Study

This section evaluates Synergy by isolating component impact (§V-B1), tuning request prioritization (§V-B2), comparing interrupt mechanisms (§V-B3), and testing resilience to load imbalance (§V-B4). We also compare Synergy to Perséphone [9], a non-preemptive, non-work-conserving system used as a baseline.

*1) **Synergy Components Breakdown**:* Figure 7 quantifies the impact of Synergy 's components using ZippyDB. The baseline (rss-ci) employs multi-queue dispatch and CI with a fixed 5 μs quantum but no load balancing. It performs worst, sustaining only 89 kRPS (10% load) with 46.1 slowdown and 1,075 μs latency due to imbalance and lack of prioritization. Adding work stealing (rss-ci+ws) increases throughput to 538 kRPS by redistributing load, yet performance remains limited under high variability since request types are not distinguished. Introducing the wait queue (rss-ci+ws+wq) offloads preempted long requests and prioritizes new arrivals, raising throughput to 763 kRPS (86% load) and significantly lowering slowdown. However, tail latency degrades beyond 61% load as long requests dominate the tail. Conditional preemption (rss-ci+ws+wq+cp) interrupts long requests only when new ones arrive, cutting unnecessary preemptions by over 90% (Figure 7, right axis) and improving both tail latency and throughput (843 kRPS, 95% load). Finally, automatic quantum adjustment (rss-ci+ws+wq+cp+qa) adapts the quantum based on application feedback, reducing slowdown from 40.2 to 30.5 at 95% load without increasing latency.

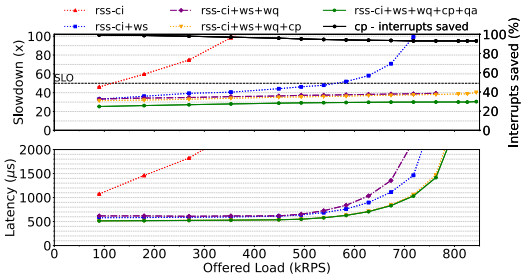

Fig. 7: Performance breakdown by Synergy components. 99.9th percentile of slowdown (top) with percentage of avoided interrupts (top right axis) and latency (bottom).

*2) **Dynamic Request Priority**:* We evaluate Synergy 's priority knobs using the Extreme workload. In Figure 8, `QUANTUM_FACTOR=2` is fixed, while `THRESH_WQD` and `CONGESTED_QUANTUM_FACTOR` vary. The baseline (d0-f0) treats the wait queue as never congested, always prioritizing short requests. Configuration (d15-f20) delays congestion detection, favoring long requests only at higher loads. Larger factors (*e.g.,* d10-f5, d10-f10) let long requests run longer once congestion is detected. These settings trade higher slowdown for lower tail latency. At 78% load (3.66 MRPS), d10-f5 increases slowdown (32.9→56.8) but reduces latency by 530 μs. With a relaxed 80× SLO, d10-f10 lowers latency by 670 μs while raising slowdown to 72.9.

Compared to Perséphone, Synergy (d0-f0) achieves 43% higher throughput before reaching a 50× slowdown (3.8 *vs.* 2.1 MRPS). Perséphone achieves lower latency due to run-to-completion but suffers from HOL blocking at high load. With d10-f10, Synergy incurs only 16% higher latency (904 vs. 765 μs) yet delivers 35% more throughput (3.75 vs. 2.37 MRPS) before both systems hit 80× slowdown.

*3) **Interrupt Methods**:* We evaluate Synergy with the interrupt mechanisms from §III-C. UINTR experiments run on the server described in Section III-C. For reference, we also evaluate Perséphone on the same setup. Figure 9 shows results for the High workload across four variants: S-IPI (`kmod_ipi`), S-Signal (`tgkill`), S-UINTR (user-level interrupts), and S-CI (compiler interrupts). As expected, S-Signal performs worst

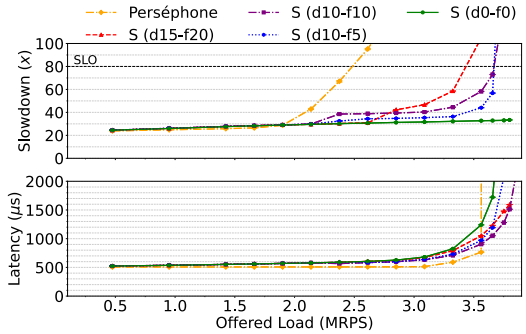

Fig. 8: Synergy knobs for adjusting request processing priorities under the Extreme workload. 99.9th percentile of both slowdown (top) and latency (bottom).

due to high interrupt overhead, though it remains competitive at low load since Synergy limits preemptions (§V-B1). At higher loads, S-IPI, S-UINTR, and S-CI increase throughput by 5%, 10%, and 13%, respectively, over S-Signal, with similar latency trends. At 239 kRPS, S-CI and S-UINTR achieve 7% and 6% higher throughput than Perséphone, with up to 6% and 23% higher tail latency.

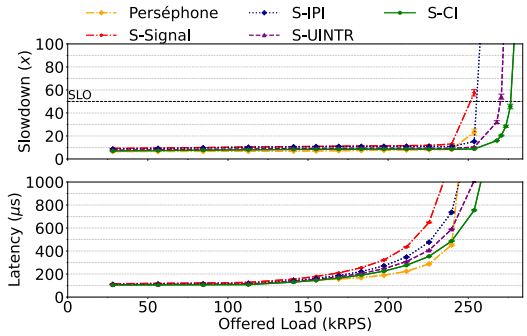

Fig. 9: Synergy performance for the High workload using different interrupt methods. 99.9th percentile of both slowdown (top) and latency (bottom).

*4) **Load Balance**:* Figure 10 evaluates Synergy under queue imbalance by varying the number of active flows. With 512 flows, queues are evenly loaded; with only 8 flows, at least 6 of 14 queues are unused, creating a 42% imbalance due to RSS. Despite this skew, the top graph shows that Synergy mantains stable performance in both cases. The bottom graph shows CPU utilization: under 8 flows, some workers are idle at low load, but as load increases, work stealing redistributes tasks and balances utilization. At 65% load (183 kRPS), the utilization gap across workers falls below 5%.

C. Multicore Scaling

We evaluate scalability by varying the number of worker cores on a 32-core Intel Xeon Gold 6438Y using ZippyDB, comparing Synergy with Perséphone (default and cFCFS). CPU utilization is fixed at 50%, and the request rate scales with the number of workers (*e.g.,* 256 kRPS with 8 workers, 961 kRPS with 30). Figure 11 shows results for 8–30 workers.

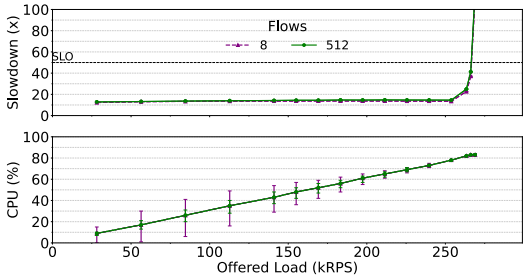

Fig. 10: Synergy performance under the High workload with uneven flow distribution across CPU cores. The upper graph shows the 99.9th percentile of slowdown. In the bottom graph, each data point represents the average CPU utilization across workers, with the bars indicating the maximum and minimum.

With 8 workers, all systems experience high tail latency due to queuing, especially Perséphone. In this setting, Synergy-CI achieves similar slowdown (13× *vs.* 16×) and 1.38× lower tail latency than Perséphone (598 vs. 826 μs), while avoiding cFCFS's extreme slowdown (596×), omitted from the figure for being out of range. As workers increase, Synergy scales consistently up to 30 cores. In contrast, Perséphone and cFCFS exceed the 0.1% packet loss threshold beyond 22 workers, revealing their scalability limits.

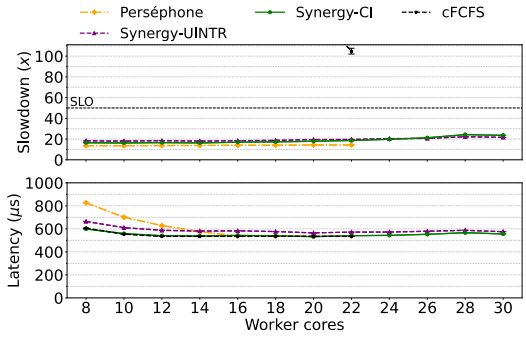

Fig. 11: Synergy multicore scaling for the ZippyDB workload.

VI. Related Work

Optimizing Network Protocol Stacks. Many systems bypass kernel-based stacks to eliminate context switches, mode transitions, and contention on shared data structures. Arrakis [6], mTCP [13], and TAS [36] move protocol processing to user space to improve performance. Arrakis uses SR-IOV to demultiplex NIC traffic directly to applications, which reduces overhead by up to 4× and improves throughput by 1.7×. mTCP achieves 3.2× higher throughput than Linux TCP through core-local, lock-free structures and batching. TAS separates fast and slow paths, improving scalability and reducing tail latency by 2.3×. However, these systems do not address HOL blocking under high service-time dispersion, which is the primary focus of Synergy.

Mitigating Head-of-Line Blocking. Shinjuku [8] and Perséphone [9] mitigate HOL blocking via preemption and core reservation. Shinjuku uses hardware-assisted IPIs to preempt long requests and achieves 6.6× higher throughput

and 88% lower tail latency in bimodal workloads than IX [12] and ZygOS [14]. Perséphone reserves cores for short requests and guarantees strict SLOs with a 5% throughput penalty. Both rely on request-type queues to bind workers to specific classes. In contrast, Synergy delegates classification to applications, uses a shared wait queue for better load redistribution, and relies on preemption to mitigate HOL blocking. AFP [37] incorporates related ideas but focuses on modeling centralized classification overhead. Guo *et al.* [38] show that the effectiveness of user-level versus compiler interrupts depends on workload characteristics, which reinforces the need for Synergy to support multiple interrupt mechanisms.

Lightweight Preemptive User-Level Schedulers. Preemptive user-level scheduling has been explored in [27], [26], but microsecond-scale SLOs require faster interrupt mechanisms. Shinjuku employs APICv and Posted Interrupts for low-latency IPIs. Concord [10] and Tiny Quanta [11] use compiler interrupts to instrument applications at compile time. The UINTR hardware feature [29] further reduces interrupt latency; LibPreemptible [30] exposes it via a flexible API. Systems such as Tardis [17], Skyloft [15], PreemptDB [16], and Vessel [39] build on UINTR for request switching, core redistribution, isolation, and multi-application sharing. Recent extensions [40] narrow the gap between polling and asynchronous interrupts. Synergy uses UINTR to preempt long requests and mitigate HOL blocking and can directly benefit from further advances in interrupt technology.

VII. Conclusion

This paper presents Synergy, a system that combines high throughput and low latency by mitigating HOL blocking in high-dispersion workloads. Synergy uses application feedback to classify requests, enabling an application-aware scheduler that supports several optimizations, including efficient preemption, low-overhead interrupts, and improved load balancing. Experimental results show that Synergy outperforms prior systems in throughput while meeting μs SLOs for short requests and reducing tail latency for long ones.

Acknowledgements

We thank the anonymous reviewers of IFIP Networking for their valuable feedback. This work was supported by CNPq (proc. 308101/2022-7), CAPES (Finance Code 001), FAPESP (procs. 2020/05183-0, 2023/00811-0, and 2023/00812-7), and Sage Networks Telecommunication Services LTDA.

References

[1] I. Zhang *et al.*, "The Demikernel Datapath os Architecture for Microsecond-scale Datacenter Systems," in *ACM SOSP*, 2021.
[2] Z. Zhang *et al.*, "CRISP: Critical Path Analysis of Large-Scale Microservice Architectures," in *USENIX ATC*, 2022.
[3] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, 2013.
[4] A. Ousterhout *et al.*, "Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads," in *USENIX NSDI*, 2019.
[5] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating Interference at Microsecond Timescales," in *OSDI*, 2020.
[6] S. Peter *et al.*, "Arrakis: The Operating System is the Control Plane," *ACM Transactions on Computer Systems*, vol. 33, no. 4, 2015.
[7] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," in *USENIX FAST*, 2020.
[8] K. Kaffes *et al.*, "Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency," in *USENIX NSDI*, 2019.
[9] H. M. Demoulin *et al.*, "When Idling is Ideal: Optimizing Tail-latency for Heavy-tailed Datacenter Workloads with Perséphone," in *ACM SOSP*, 2021.
[10] R. Iyer, M. Unal, M. Kogias, and G. Candea, "Achieving Microsecond-scale Tail Latency Efficiently with Approximate Optimal Scheduling," in *ACM SOSP*, 2023.
[11] Z. Luo *et al.*, "Efficient Microsecond-scale Blind Scheduling with Tiny Quanta," in *ACM ASPLOS*, 2024.
[12] A. Belay *et al.*, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *USENIX OSDI*, 2014.
[13] E. Jeong *et al.*, "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems," in *USENIX NSDI*, 2014.
[14] G. Prekas *et al.*, "Zygos: Achieving Low Tail Latency for Microsecond-scale Networked Tasks," in *ACM SOSP*, 2017.
[15] Y. Jia *et al.*, "Skyloft: A General High-Efficient Scheduling Framework in User Space," in *ACM SOSP*, 2024.
[16] K. Huang *et al.*, "Low-Latency Transaction Scheduling via Userspace Interrupts: Why Wait or Yield When You Can Preempt?" *Proc. of ACM on Management of Data*, vol. 3, no. 3, 2025.
[17] Y. Yang, Z. Huang, A. Kaufmann, and J. Li, "Protected Data Plane OS Using Memory Protection Keys and Lightweight Activation," 2023. [Online]. Available: https://arxiv.org/abs/2302.14417
[18] S. McClure *et al.*, "Efficient Scheduling Policies for Microsecond-Scale Tasks," in *USENIX NSDI*, 2022.
[19] "Redis Pipelining," 2025, https://redis.io/docs/latest/develop/use/pipelining/.
[20] I. Cho *et al.*, "LDB: An Efficient Latency Profiling Tool for Multithreaded Applications," in *USENIX NSDI*, 2024.
[21] RSS, "Introduction to receive side scaling," 2023, https://learn.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.
[22] A. Pesterev *et al.*, "Improving Network Connection Locality on Multicore Systems," in *ACM EuroSys*, 2012.
[23] A. Kaufmann *et al.*, "High Performance Packet Processing with Flexnic," in *ACM ASPLOS*, 2016.
[24] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "A comparison of receiver-initiated and sender-initiated adaptive load sharing," *Performance evaluation*, vol. 6, no. 1, 1986.
[25] B. Teabe *et al.*, "Application-Specific Quantum for Multi-Core Platform Scheduler," in *ACM EuroSys*, 2016.
[26] S. Shiina *et al.*, "Lightweight Preemptive User-level Threads," in *ACM PPoPP*, 2021.
[27] S. Boucher *et al.*, "Lightweight Preemptible Functions," in *USENIX ATC*, 2020.
[28] A. Belay *et al.*, "Dune: Safe User-level Access to Privileged {CPU} Features," in *USENIX OSDI*, 2012.
[29] Sohil Mehta, "x86 User Interrupts Support," 2023, https://lwn.net/Articles/869140/.
[30] Y. Li *et al.*, "Libpreemptible: Enabling Fast, Adaptive, and Hardware-assisted User-space Scheduling," in *IEEE HPCA*, 2024.
[31] Intel, "Data plane development kit," 2024, https://www.dpdk.org/.
[32] H. Qin *et al.*, "Arachne: Core-Aware Thread Management," in *USENIX OSDI*, 2018.
[33] LevelDB, "LevelDB," 2024, https://github.com/google/leveldb.
[34] D. Duplyakin *et al.*, "The Design and Operation of CloudLab," in *USENIX ATC*, 2019.
[35] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *IEEE CGO*, 2004.
[36] A. Kaufmann *et al.*, "TAS: TCP Acceleration as an OS Service," in *ACM EuroSys*, 2019.
[37] M. Berghetti *et al.*, "AFP: A Feedback-Driven Microservices Request Scheduler," in *SBRC 2024*, 2024.
[38] L. Guo *et al.*, "The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling," in *USENIX NSDI*, 2025.
[39] J. Lin *et al.*, "Fast Core Scheduling with Userspace Process Abstraction," in *ACM SOSP*, 2024.
[40] B. Aydogmus *et al.*, "Extended User Interrupts (xUI): Fast and Flexible Notification without Polling," in *ACM ASPLOS*, 2025.