REEF: Energy-Efficient, Application-QoS-Aware Thread Processing in Oversubscribed Server Environments

Ning Li
Dept. of CSE, University of Texas at Arlington
Arlington, Texas, USA
ningli.windbell@gmail.com

Hong Jiang
Dept. of CSE, University of Texas at Arlington
Arlington, Texas, USA
hong.jiang@uta.edu

Hao Che
Dept. of CSE, University of Texas at Arlington
Arlington, Texas, USA
hche@cse.uta.edu

Zhijun Wang
Dept. of CSE, University of Texas at Arlington
Arlington, Texas, USA
zhijun.wang@uta.edu

Abstract

Modern storage-intensive server applications such as key-value stores and relational databases often rely on thread oversubscription to sustain high throughput in cloud environments. While effective at hiding I/O stalls, this practice introduces serious challenges, including unpredictable query behaviors, instruction-per-query (IPQ) inflation, and instability in dynamic power management (DPM). Existing QoS- or energy-centric techniques, developed in isolation at one of the service layers, fail to holistically optimize resource and energy efficiency under connection-level QoS constraints. This paper presents REEF (*R*esource- and *E*nergy-*E*fficient user-space scheduling *F*ramework), a non-intrusive, cross-layer approach that coordinates query processing across the network stack, application layer, and OS resource manager. REEF transforms self-serving threads into on-call threads activated by near-optimal, proactively batched scheduling, enabling deep CPU C-state residency and mitigating CPU and I/O contention. Our extensive evaluation on real server applications (MongoDB and MySQL) demonstrates that REEF can substantially improve the energy efficiency of server applications under different connection-level QoS schemes, by up to 53.45% for throughput per power, up to 2.18× and 5.33× for coefficient of P99 and P99.9 tail-latency per power respectively, and significantly reduce resource consumption, by up to 73.65% in CPU frequency.

CCS Concepts

• **Computer systems organization → Multicore architectures**; • **Software and its engineering → Power management**; **Operating systems**; • **Information systems → Database management system engines**; • **Networks → Network performance modeling**.

Keywords

Power and Energy Management, Connection-Level Quality of Service (QoS), Resource and Energy Efficiency, Oversubscribed Server Environments, Workload Consolidation and Thread Coordination

ACM Reference Format:
Ning Li, Hong Jiang, Hao Che, and Zhijun Wang. 2025. REEF: Energy-Efficient, Application-QoS-Aware Thread Processing in Oversubscribed Server Environments. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772269

1 Introduction

Nowadays, increasingly more enterprises move their businesses to the cloud for agile and cost-effective data services based on various storage-intensive *server applications* (e.g., key-value stores and relational database systems). This is because modern datacenters provide elastic compute and storage resources with multi-threaded models to serve hundreds or even thousands of clients or components (e.g., microservices or serverless functions) by network connections. These server applications often have *worker threads* that scale with client connections and become idle during disk/network I/Os, necessitating sufficient thread concurrency beyond hardware limits to maintain sustainably high performance. This has led to a practice called *thread oversubscription* [33], in which more threads than hardware execution units (e.g., CPU cores or hardware threads) are created by service providers to exploit elasticity and make full use of the given resources to maximize total throughput. However, as a double-edged sword, thread oversubscription also introduces several critical challenges, as illustrated in Fig. 1.

1) *Arbitrary Query Behaviors*: Thread oversubscription can amplify the impact of arbitrary and highly dynamic connection-level query patterns, leading to increasingly randomized and unpredictable thread processing behavior. This unpredictability becomes particularly problematic when operating systems use CPU schedulers such as Linux's Completely Fair Scheduler (CFS)[12], which allocate CPU time evenly across threads without considering their underlying connection-level quality-of-service (QoS) requirements. *Connection-level QoS* refers to a server's ability to meet performance guarantees, such as query or transaction latency targets, on a per-connection basis. In environments with hundreds or thousands of concurrent client connections that are each typically served by a dedicated or shared worker thread, the mismatch between

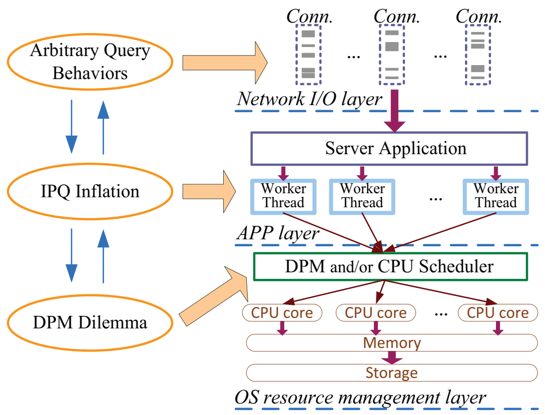

Figure 1: An illustration of the challenges posed by thread oversubscription.

fair scheduling and heterogeneous QoS needs can result in two major problems. First, CPU core-level idle periods are frequently and unpredictably interrupted due to merging of diverse query patterns from multiple threads scheduled on the same core. This reduces the likelihood idle durations exceed the threshold required for entering deep CPU C-states (such as C6), which are essential for energy savings. Second, oversubscription increases contention and queuing delays, raising the risk of violating service-level objectives (SLOs), particularly for tail latency metrics that are critical to latency-sensitive applications. As a result, although thread oversubscription may improve total throughput, it can degrade both energy efficiency and QoS under bursty and scalable workloads.

2) *Instruction Per Query (IPQ) Inflation*: Server applications such as MongoDB [5] and MySQL [6] heavily rely on concurrency control mechanisms and resource partitioning strategies [28, 41, 48, 58, 62, 67] to optimize multi-core execution and coordinate concurrent accesses to shared resources like buffer caches, all the while preserving correctness. However, thread oversubscription can overwhelm these mechanisms, leading to increased *unproductive processing*, e.g., excessive lock contention or repeated wait cycles, which produce little or no useful work from the perspective of end users (e.g., successfully completed queries). In parallel, oversubscription-induced cache conflicts and misses trigger frequent slow-path operations (e.g., fetching data from disk rather than cache), resulting in *inefficient processing*. These factors increase the CPU instructions required per query, a phenomenon known as IPQ inflation, reducing resource efficiency by consuming more CPU cycles without commensurate performance gains.

3) *Dilemma of Dynamic Power Management (DPM)*: Thread oversubscription inherently causes multiple threads or processes to share underlying hardware execution units. DPM techniques aim to balance energy efficiency and QoS by using hardware features such as dynamic voltage and frequency scaling (DVFS) [15, 17, 19, 35, 39, 47, 50, 54, 57] and CPU C-state management [13, 22, 52, 53, 55, 71, 72]. These mechanisms depend on the stability and predictability of the relationship between QoS-relevant metrics and CPU resource allocation. Metrics such as throughput and mean latency are typically assumed to respond predictably to changes in CPU frequency or core assignment. However, under thread oversubscription, this relationship becomes unstable and fractured, particularly for metrics sensitive to latency variability. Resource contention and I/O bottlenecks from over-provisioning compute resources relative to backend capabilities can distort this relationship, as demonstrated in Section 2.1. Consequently, DPM may introduce large and unpredictable control-induced errors when regulating latency-sensitive QoS metrics such as tail latency SLOs. This problem becomes even more severe in cloud-scale systems where many client connections, each with distinct and potentially conflicting QoS needs, are multiplexed onto oversubscribed threads. In such scenarios, the ability of DPM to maintain precise QoS control while minimizing energy use becomes significantly weakened.

Despite growing awareness of these challenges, existing solutions remain insufficient. Most QoS-centric scheduling policies [8, 34, 42, 43, 60] and resource- or energy-centric DPM schemes [15, 17, 19, 35, 39, 47, 48, 50, 54, 57, 58, 62] are not well-suited to address the compounded impact of thread oversubscription on connection-level QoS, resource utilization, and energy efficiency. These solutions are typically developed for distinct objectives and are implemented at different service layers, such as the network I/O layer, the application layer, or the OS resource management layer, largely oblivious of other layers. While interactions across these layers are possible, they are often loosely coordinated and largely reactive, serving upstream requests in a passive manner. This separation in design and execution significantly hinders holistic optimization of QoS enforcement and overall system efficiency in terms of both resource and energy consumption.

In this paper, we present the *R*esource- and *E*nergy-*E*fficient user-space scheduling *F*ramework (REEF), a non-intrusive, cross-layer solution that coordinates the end-to-end query processing pipeline across multiple service layers for server applications such as MongoDB[5] and MySQL[6]. REEF improves overall resource and energy efficiency while maintaining connection-level QoS guarantees. At its core, REEF establishes a resource- and energy-aware connection-to-thread control path by transforming traditional self-serving worker threads into on-call threads that remain idle until explicitly scheduled. This architecture synchronizes query access patterns with the execution states (running or idle) of worker threads through proactive, near-optimal query batching. As a result, REEF extends thread-level idle durations within QoS constraints, increasing opportunities for deeper CPU C-state residency and enhancing energy efficiency. Additionally, REEF mitigates inefficient thread activity by reducing CPU and I/O contention, improving both resource- and energy-efficiency while alleviating IPQ inflation.

We implement a REEF prototype and evaluate its effectiveness in enhancing the resource and energy efficiency of server applications such as MongoDB and MySQL. Our evaluation spans seven representative kernel- and user-space QoS approaches across ten diverse test scenarios. Results show that REEF consistently improves both resource- and energy-efficiency without compromising connection-level QoS. The REEF source code is available at https://github.com/NingBellWind/REEF_core.

2 Background and Motivation

2.1 Background and Case Studies

In this section, we dive deeper into the multi-layer software stack (i.e., the network I/O layer, application layer, and the OS resource management layer) that must be established to support running of

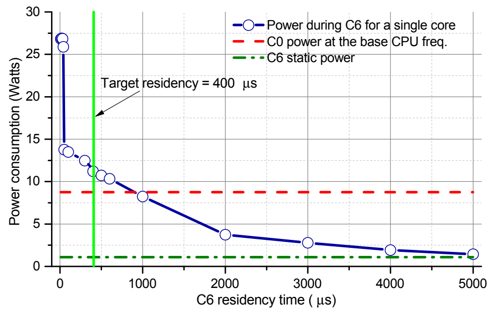

Figure 2: **An illustration of the relationship between C6 power consumption and residency time for one CPU core.**

a server application, as shown in Fig. 1, and further elaborate on the three challenges imposed by thread oversubscription.

Arbitrary Query Behaviors: Arbitrary and dynamic query behaviors at the *network I/O layer* propagate downward, causing irregular sleep-wake cycles of threads at the *application layer.* Under CPU oversubscription, these diverse execution patterns interleave across threads sharing the same core, fragmenting idle periods and reducing chance of entering deep CPU C-states for power savings.

C-states, managed at the *OS resource management layer* (e.g., via the Intel CPU idle driver in Linux [68]), allow parts of the CPU (e.g., PLL, caches, core clock) to be powered down during idle times. Deeper states like C6 offer greater savings but incur higher transition latency and require a minimum *target residency*[1]. In practice, cores must remain idle significantly beyond this threshold, often 2 to 3×, to achieve net energy benefits [10, 71].

To study power-residency tradeoffs, we run a controlled experiment on an Intel® Xeon® E5-2650 v4 CPU. A synthetic workload repeatedly enters sleep states from 10 μs to 5000 μs, running one minute per configuration on an isolated core with turbo boost disabled and all C-states except C6 off. This forces core into C6 upon sleeping. Using RAPL interface [47, 59], we measure power consumption against C6 residency. As shown in Fig. 2, short C6 residencies (<300 μs) result in power usage exceeding C0 base frequency due to entry overheads like cache flushes, context saves, and PLL shutdowns. Meaningful savings only emerge after 1000 μs, when power drops below C0 levels, despite the 400 μs target residency [3]. These findings align with Intel's management code [71].

As a result, it remains highly challenging for dynamic and bursty query patterns, especially under thread oversubscription, to effectively leverage deep C-states such as C6 for energy savings. The irregular sleep-wake transitions at the application layer, induced by unpredictable I/O behaviors, tend to fragment core idle periods. This fragmentation prevents the processor from remaining idle long enough to surpass the target residency thresholds required to offset C6's transition costs. Consequently, while deep C-states offer significant power-saving potential, their utilization is often undermined in oversubscribed server environments. Shallower C-states, though more readily accessible, suffer from fragmented idle periods caused by arbitrary query behaviors, limiting their residency and resulting in modest energy savings.

IPQ Inflation: To support large-scale connection-driven services, server applications often use numerous self-serving worker

[1]Target residency refers to the minimum idle time needed for energy savings to outweigh transition costs.

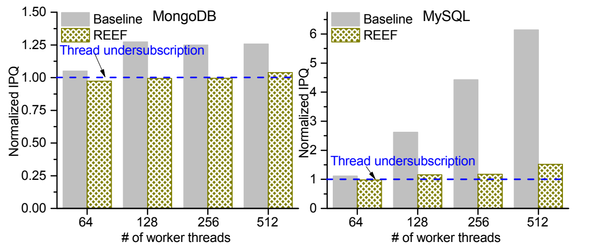

Figure 3: **The comparisons of IPQ for MongoDB and MySQL under different levels of thread oversubscription.**

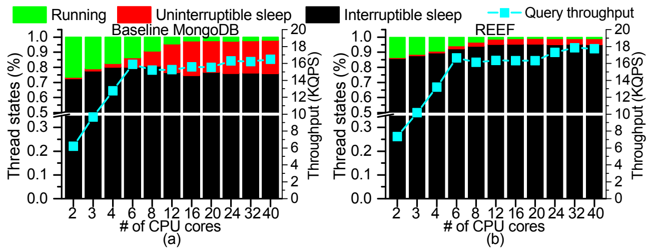

Figure 4: **Breakdown of thread states and query throughput under server application thread packing across different numbers of CPU cores.**

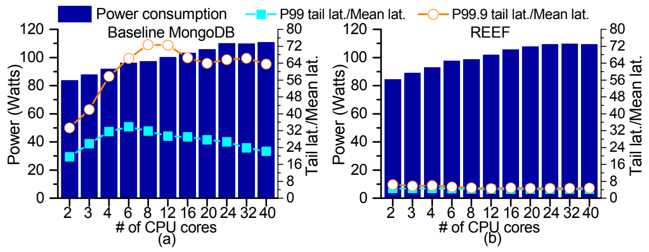

Figure 5: **Comparisons of power consumption and query latency variability (measured by tail latencies normalized by mean) under server application thread packing across different numbers of CPU cores.**

threads, each managing one or more connections. These threads continuously poll state (e.g., I/O events) and immediately process incoming queries to minimize latency. However, such behavior can amplify resource contention, leading to inefficiencies like repeated lock contention, cache conflicts, and excessive disk I/O. These factors collectively inflate the number of instructions executed per query (IPQ), particularly under thread oversubscription.

Figure 3 shows normalized IPQ trends for two systems: MongoDB [5] using a YCSB workload [24], and MySQL [6] running a sysbench-based OLTP workload [40]. Experimental configurations are detailed in Section 5. For MongoDB, IPQ grows moderately from 1.05× to 1.27× as threads scale from 64 to 128, plateauing up to 512 threads. In contrast, MySQL shows a continual IPQ increase from 1.11× to 6.14× driven by compounded lock and cache contention [42]. These results underscore the challenge of IPQ inflation under thread oversubscription, which escalates CPU cycle usage.

DPM Dilemma: As discussed in Section 1, dynamic power management (DPM) aims to reduce energy usage while meeting quality-of-service (QoS) constraints. A common assumption in DPM is that QoS metrics, such as throughput or tail latency, exhibit predictable, often monotonic/linear relationships with power consumption. To evaluate this assumption for metrics like query throughput and P99/P99.9 latencies, we apply thread packing [23, 36, 61], running MongoDB with 512 worker threads under a YCSB workload [24].

Threads are consolidated onto 2 to 40 CPU cores, with unused cores entering deep C-states (e.g., C6) via Intel CPU idle driver [68].

This setup enables us to observe QoS and power trends under varying core allocations. As shown in Fig. 4(a) and 5(a), throughput increases linearly and mean latency declines (no shown) as core count grows from 2 to 6. Beyond that, both metrics plateau, while power consumption continues to rise. Thus, DPM can reliably trade energy for performance in this low-core range (2-6 in this example).

However, tail latencies diverge from mean latency as core count increases, with P99.9 latency rising steeply from 2 to 8 cores. This indicates higher percentile latency variability despite more cores. As Fig. 4(a) shows, this behavior results from increased I/O contention. While more cores alleviate CPU contention (reflected by fewer threads in the *running state (R)*), they increase uncoordinated I/O activity, leading higher *uninterruptible sleep state (D)*, where threads await disk/network I/O and cannot be interrupted. In contrast, the *interruptible sleep state (S)* represents idle threads that can be awakened by signals/events, offering genuine energy-saving opportunities. As D-state residency increases and encroaches on S-state time, the system loses chances for voluntary CPU yielding, thereby reducing the likelihood of achieving energy savings without affecting QoS. Since DPM schemes typically manage only CPU resources, they fail to jointly address CPU and I/O contention. Consequently, the presumed linear QoS-power relationship breaks down, limiting DPM's ability to balance energy efficiency and QoS.

Motivation: To address the compounded challenges of thread oversubscription, we propose REEF, a cross-layer, non-intrusive framework for efficient scheduling and energy management. REEF mitigates uncoordinated query arrivals via proactive batching, aligning inter-batch gaps with optimal CPU C-state residency for improved energy efficiency. To reduce contention and IPQ inflation from oversubscribed self-serving threads, REEF adopts scheduler-driven on-call threads that activate only when assigned work. It further decouples QoS enforcement from power control through a programmable, connection-aware scheduling interface. Finally, REEF applies a unified latency-based model to classify and consolidate threads, ensuring predictable QoS and enabling fine-grained coordination between QoS requirements and energy-aware scheduling across diverse workloads.

2.2 Related Work

Table 1 summarizes limitations of existing solutions in addressing core challenges introduced by thread oversubscription: (1) arbitrary query behaviors, (2) IPQ inflation, and (3) the DPM dilemma.

QoS-centric scheduling frameworks, such as AppleS [42], Rein [60], and UTSLO [43], target QoS metrics like tail latency and fairness, but neglect energy efficiency. Kernel-space schedulers like ghOSt [34] and Cgroups [8] provide thread-level control but lack coordination with application-layer query behaviors.

System-wide DPM techniques, including Pegasus [47], TimeTrader [66], and PowerNap [52], optimize energy efficiency under aggregated metrics but fail to capture connection-level variance and thread dynamics. While C-state-based methods improve efficiency globally, they are not tailored to multi-threaded server workloads.

Application-oriented DPM solutions, such as Rubik [39] and μDPM [21], offer sub-millisecond control but assume low request rates and overlook thread oversubscription. Peafowl [13] adopts intrusive energy-aware scheduling, requiring tight integration and fixed thread-core mappings, unsuitable for dynamic environments.

Approach	Arbitrary Qry. Behavior	IPQ Inflation	DPM Dilemma
QoS-Centric Scheduling Frameworks			
AppleS [42]	Median	Median	✗
UTSLO [43]	Median	Median	✗
ghOSt [34]	✗	Low	✗
Cgroups [8]	✗	Low	✗
Fine-grained Schedulers [37, 49]	Median	Low	✗
System-wide DPM Techniques			
Pegasus [47]	Low	✗	✗
TimeTrader [66]	Median	✗	✗
Carb [73]	Low	✗	✗
Dreamweaver [55]	Median	✗	✗
PowerNap [52]	Low	✗	✗
Application-Oriented DPM Solutions			
Rubik [39]	Low	✗	✗
μDPM [21]	Median	✗	✗
Peafowl [13]	Low	✗	✗
Demeter [65]	Median	Low	✗
This Work			
REEF	High	High	High

Table 1: Comparison of prior solutions and REEF in addressing key challenges under thread oversubscription. For *Arbitrary Query Behavior*, "Low" indicates reactive handling, while "Median" denotes proactive handling that does not span the full query processing pipeline. For *IPQ Inflation*, "Low" reflects unstable optimization under thread oversubscription, and "Median" indicates partial mitigation of I/O contention. "High" represents comprehensive support across the entire query processing pipeline (covering both CPU and I/O contention), whereas ✗ denotes no effective support.

Demeter [65] presents a QoS-aware CPU scheduler that reduces power consumption for co-located black-box workloads by adjusting core allocations based on latency SLOs. Although conceptually similar to REEF, Demeter oversimplifies thread behaviors and overlooks the impact of dynamic query behaviors and IPQ inflation, limiting its effectiveness in balancing QoS and energy efficiency under oversubscription, as shown in Section 5.4.

Recent DPM efforts have shifted toward microservices [16], serverless platforms [46, 63], and large language models (LLMs) [64], but they fail to address the fundamental challenges of thread oversubscription. Unlike these approaches, REEF is a cross-layer, energy-aware scheduling framework that orchestrates the entire query pipeline, from the network I/O layer to the OS layer, while mitigating both CPU and I/O contention. By decoupling QoS enforcement from power management, REEF overcomes the DPM dilemma and ensures predictable performance with minimal IPQ inflation.

3 The REEF Design

As illustrated in Fig. 6, REEF provides an application-aware scheduling framework that coordinates query processing across the network I/O, application, and OS resource management layers using the Ideal Batch optimization strategy. It achieves both resource and

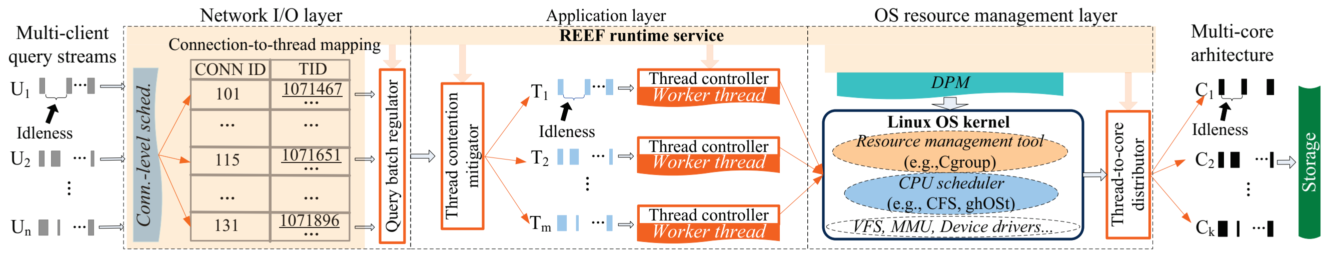

Figure 6: **The REEF architecture.**

energy efficiency while satisfying QoS targets. Unlike conventional approaches that focus solely on performance or energy, REEF maximizes CPU idle time within QoS bounds and mitigates contention from thread oversubscription, improving both system efficiency and connection-level QoS. To guide and evaluate REEF's design, we define and quantify efficiency metrics aligned with these goals.

3.1 Comprehensive Efficiency Assessment

REEF provides a comprehensive framework for evaluating resource and energy efficiency along with diverse QoS targets, including throughput and tail latency metrics such as the 99^{th} ($P99$) and 99.9^{th} ($P99.9$) percentiles.

Resource efficiency is typically assessed via throughput and tail latency for different application types (e.g., I/O- or CPU-intensive) [14, 27, 38, 42, 62, 70]. Accordingly, we define resource efficiency using normalized performance in throughput, $P99$, and $P99.9$ tail latency, measured as the geometric mean of the speedups of co-located applications relative to their isolated baselines, with harmonic mean as an alternative [20, 26, 69]. To further reflect CPU resource use, we track the average frequency of cores serving worker threads under high I/O load (typically exceeding 95%).

Symbols	Definition
$EE(TPP)$	Throughput per power (TPP).
$EE(Px)$	$\frac{TPP}{Px\_tail/mean\_latency}$.

Table 2: **The energy efficiency (EE) measures in terms of throughput and tail latency, where TPP measures throughput per power unit and $EE(Px)$ reflects the tail-latency performance per power unit in terms of the x^{th} percentile tail latency normalized to the mean latency.**

Energy efficiency is evaluated using multiple metrics aligned with different performance goals: $EE(TPP)$, $EE(P99)$, and $EE(P99.9)$, as shown in Table 2. These metrics are computed using the total power consumed by the CPU, memory, and storage components necessary for running the applications.

3.2 Ideal Batch Strategy

As illustrated in Fig. 2, deep C-state (C6) residency inversely correlates with power consumption until reaching static C6 power. However, meaningful power savings occur only if the residency exceeds a break-even threshold T (e.g., 1000 μs for Intel® Xeon® E5-2650 v4). This underscores the need to maximize uninterrupted CPU idle time (e.g., Δ_j^{idle} on core j) and ensure that it consistently exceeds T; otherwise, C6 entry yields no energy benefit.

For example, a single I/O-intensive connection served by a worker thread on core j may exhibit widely varying inter-arrival times Δ_j due to arbitrary query behaviors. This unpredictability introduces randomness in Δ_j^{idle}, potentially reducing core idle durations below T. We model Δ_j^{idle} as:

$$\Delta_j^{\text{idle}} = \max(\Delta_j - \delta, 0), \tag{1}$$

where δ denotes the per-query overhead, comprising query processing time δ_{proc}, CPU scheduling latency δ_{sched}, and C-state wake-up delay δ_{wakeup}. For an architecture with M cores, the potential for power savings through deep C-state (C6), termed *C6 Opportunity*, is expressed as:

$$\text{C6 Opportunity}(T) = \frac{\sum_{j=0}^{M-1} \Delta_j^{\text{idle}} \cdot \mathbf{1}(\Delta_j^{\text{idle}} > T)}{T_{\text{total}}} \tag{2}$$

where $T_{\text{total}} = \sum_j \Delta_j^{\text{idle}} + \frac{T_{\text{proc}}}{M}$, T_{proc} is the accumulated query processing time, and $\mathbf{1}(\cdot)$ is the indicator function.

To address this, we explore the optimal query batching strategy, termed *Ideal Batch*, for a core j handling a query stream with mean inter-arrival time μ_j. Since only idle periods longer than the break-even threshold T yield real energy savings, the effective idle time is modeled as $\Delta_j^{\text{idle}} \cdot \mathbf{1}(\Delta_j^{\text{idle}} > T)$.

We begin by defining the function:

$$f(\Delta_j) = \max(\Delta_j - \delta, 0) \cdot \mathbf{1}(\Delta_j > T + \delta),$$

which captures the effective idle time after accounting for per-query overhead δ. This function is convex for $\Delta_j > \delta$, making it suitable for optimization in determining the interarrival time that maximizes C-state residency opportunities. By Jensen's inequality:

$$\mathbb{E}[f(\Delta_j)] \le f(\mathbb{E}[\Delta_j]) = f(\mu_j)$$

Maximum occurs when interarrival times are fixed: $\Delta_j = \mu_j$. Thus, we can achieve the core's optimal idle time $\Delta_j^{\text{batch-idle}}$ under N-query batching:

$$\Delta_j^{\text{batch-idle}} = N \cdot \mu_j - N \cdot \delta_{proc} - \delta_{sched} - \delta_{wakeup}, \tag{3}$$

Based on Eq. 3, the batch size N can be adjusted to optimize the effective idle time $\Delta_j^{\text{batch-idle}}$. This is because δ_{sched}, representing context switch overhead, is typically a few microseconds [33], and δ_{wakeup} occurs once per batch, amortized across N queries. As N increases, $\Delta_j^{\text{batch-idle}}$ grows, improving *C6 Opportunity*. However, excessive inter-batch idleness can hurt tail-latency-related efficiency (e.g., $EE(P99)$ and $EE(P99.9)$), making batch size a constraint to a practical solution, as discussed in Section 3.4.

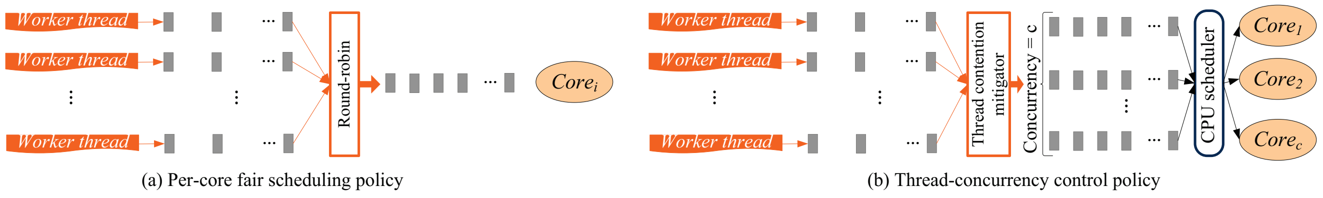

Figure 7: **The illustration of on-core Ideal Batching.**

3.3 On-Core Ideal Batching

Enforcing the Ideal Batch strategy in oversubscribed servers is challenging due to shared CPU cores among multiple worker threads and the need to consolidate heterogeneous client query streams. Each client connection is typically handled by a dedicated worker thread, a best practice promoting fine-grained concurrency, deterministic execution, easier debugging, and improved cache locality [32]. However, this model strains OS-level CPU and I/O management, particularly under arbitrary query behaviors, due to irregular multi-threaded contention.

To address this, REEF implements thread-level Ideal Batching by converting traditional self-serving threads into on-call threads that remain idle until assigned a query batch (see Section 4). Upon completing its batch, a thread returns to idle, enabling per-thread Ideal Batch execution.

Query batches must be fairly distributed across threads sharing a core. A basic approach (Fig. 7(a)) groups threads by core and uses fair-sharing (e.g., weighted round-robin [7]). However, this method has two limitations: 1) It addresses only CPU contention. As shown in Fig. 4 and Fig. 5, I/O contention in oversubscribed settings severely limits DPM's ability to balance QoS and energy efficiency, a problem CPU-only policies cannot solve. 2) Strict per-core scheduling interferes with OS-level CPU schedulers, constraining optimizations like NUMA balancing and cache locality.

REEF introduces a thread contention mitigator that dynamically reduces both CPU and I/O contention under the Ideal Batch strategy while remaining cooperative with kernel schedulers. It hides excessive thread activity from OS resource managers and tunes the thread concurrency limit C to match available CPU and I/O resources, as shown in Fig. 7(b). To preserve system responsiveness, C is set below the total number of physical cores, leaving headroom for the OS and co-located workloads. In multi-tenant settings, performance often degrades due to variable I/O loads [29, 31], amplifying the impact of CPU over-provisioning on QoS, especially for compute- and I/O-bound applications, as shown in Fig. 5.

Determining the number of CPU cores dedicated to a server application, C_{avail}, is straightforward by excluding allocations for the OS and co-located services. However, estimating the optimal number of cores to fully utilize storage I/O under heavy threading, C_{optimal}, is more challenging. Inspired by AppleS [42], which identifies the minimum parallelism to saturate I/O, we define C_{optimal} as the fewest concurrent worker threads, $\text{Min}N_{\text{worker}}$, that maximize throughput. Since worker threads dominate in thread oversubscription, we approximate $C_{\text{optimal}} \approx \text{Min}N_{\text{worker}}$, and enforce the concurrency limit as $C \approx \min(C_{\text{avail}}, \text{Min}N_{\text{worker}})$. This hides many on-call threads from the OS, enabling scalable performance. As shown in Fig. 3(b), REEF mitigates IPQ inflation to levels comparable with thread undersubscription for both MongoDB and MySQL. By easing I/O contention, REEF ensures stable latency variability even with varied CPU core counts, as shown in Fig. 5(b), enabling a balanced trade-off among QoS, resource use, and energy efficiency.

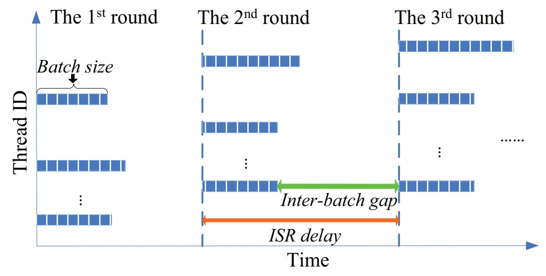

Figure 8: **An illustration of the scheduling policy implemented by REEF.**

To enforce thread concurrency in on-core Ideal Batching, the thread contention mitigator adopts a customized token-bucket scheduler to regulate batch-level admission across worker threads. Unlike traditional QoS-driven token-bucket schedulers [74–76] that regulate query rates solely to meet QoS targets, our approach optimizes query patterns at the batch level to achieve near-optimal C6 residency, maximizing energy efficiency without sacrificing QoS. Under the concurrency cap C, execution proceeds in discrete scheduling rounds, during which only C worker threads may dispatch query batches. In each round, a thread T_i can submit a batch of N_i queries, determined by its token allocation, into the system stack, including both the application and OS layers, provided that $N_i > 0$, as shown in Fig. 6. The value of N_i is dictated by QoS policies. If $N_i = 0$, the thread yields its turn.

3.4 QoS Policies under Ideal Batching

As shown in Fig. 6, REEF implements connection-level QoS scheduling through a query batch regulator, which assigns a batch size to each server application's worker thread. To maximize inter-batch idle time and fully utilize CPU and I/O resources, the regulator adaptively adjusts the interval between scheduling rounds, called the *inter-scheduling-round (ISR)* delay, according to a preset load threshold. This delay translates to core-level idle time, improving both resource and energy efficiency.

A connection-level QoS policy typically implements a feedback control loop that dynamically determines "who" (i.e., which thread) can send "how many queries" (i.e., the query batch size) at any given moment according to connection-level QoS objectives [30, 31, 42–44]. To integrate such a policy into the REEF runtime, as illustrated in Fig. 8, the query batch regulator partitions the system runtime into multiple *scheduling rounds*, which serve as the minimum granularity for enforcing QoS targets under the Ideal Batch strategy. At the beginning of each scheduling round, the QoS scheduling policy determines the batch size for each worker thread based on connection-specific performance metrics and the scheduling algorithm. The REEF runtime, acting as an intermediary between

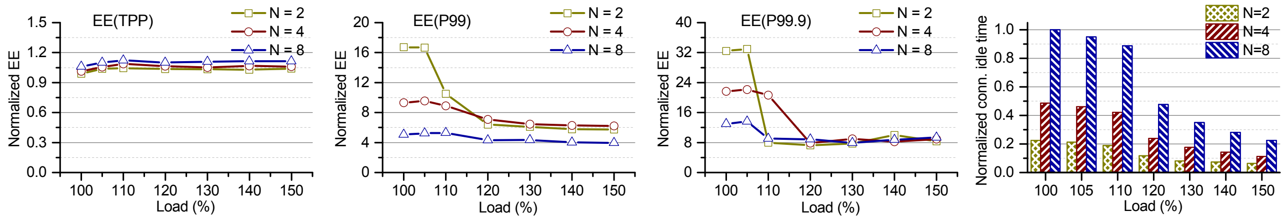

Figure 9: A real case of the throughput target optimization constrained by energy-efficient(EE) metrics.

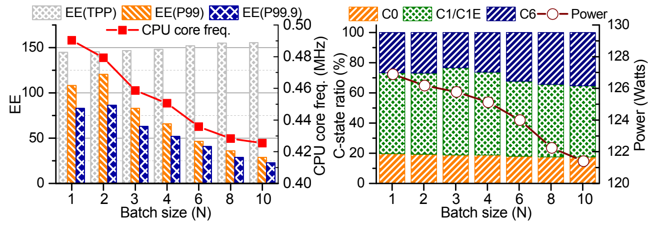

Figure 10: Comparisons in CPU frequency, the energy-efficiency metrics, as well as the power consumption and its C-state breakdowns under different query batch size.

the server application and the policy engine, provides the necessary connection-to-thread mappings and access to connection-level performance metrics such as throughput and tail latencies.

To support this, REEF extracts connection metadata (socket ID, IP, port) from system calls and links it with the serving thread ID (TID). It also monitors system call traces to compute runtime statistics, including throughput, latency, and inter-arrival times.

ISR delay optimization: To maximize the ISR delay D while fully utilizing server capacity, the query batch regulator dynamically adjusts D based on a load limit λ, aiming for high energy efficiency. The limit λ is set according to the average query throughput $\bar{\lambda}$ observed under high I/O utilization (e.g., above 95%), which can be estimated from system configurations. Setting $\lambda > \bar{\lambda}$ may leverage transient over-provisioning for higher throughput but risks I/O congestion during resource contention, degrading tail latency and energy-efficiency metrics such as $EE(P99)$ and $EE(P99.9)$. As expected, increasing λ or reducing batch size N shortens the ISR delay. By default, the minimum delay is based on the C6 break-even threshold (see Fig. 2), ensuring even the core with the shortest idle window can benefit from deep C-states for energy savings.

Figure 9 shows a case study on tuning the load limit $\lambda^p = \frac{\lambda}{\bar{\lambda}} \times$ 100% for a MongoDB key-value store with 512 worker threads (see Section 5.1 for details). As λ^p increases from 100% to 150%, $EE(TPP)$ initially improves but plateaus around $\lambda^p = 105\%$, where throughput saturates. Beyond this point, both $EE(P99)$ and $EE(P99.9)$ decline, making $\lambda^p = 105\%$ an optimal trade-off. A smaller batch size N improves $EE(P99)$ and $EE(P99.9)$, albeit with a slight reduction in $EE(TPP)$. In practice, REEF can automatically optimize λ for a given server application before deployment.

Batch size optimization: The batch size N can be tuned to optimize energy-efficiency metrics, particularly $EE(P99)$ and $EE(P99.9)$. As shown in Fig. 10, using a small batch size (e.g., $N = 2$) significantly improves tail-latency efficiency, by 3.75× for $EE(P99)$ and 3.64× for $EE(P99.9)$, at the cost of a modest 7% drop in $EE(TPP)$. This gain results from shorter inter-batch intervals, which increase thread activation frequency and reduce delay between batches. However, smaller N limits connection-level locality (e.g., cache reuse) [42], slightly lowering throughput. As shown in the rightmost subfigure of Fig. 9, it also increases power usage by up to 5.6 watts (4.61%). To balance these trade-offs, REEF dynamically caps batch size per thread based on optimization goals. For throughput-centric use, it allows larger batches (e.g., $N \leq 8$) to exploit locality; for tail-latency-sensitive workloads, it enforces smaller N (e.g., $N = 2$) when ISR delays exceed the C6 break-even threshold.

3.5 DPM under Ideal Batching

REEF decouples QoS enforcement from dynamic power management (DPM) at the OS resource management layer, addressing the DPM dilemma of balancing QoS and energy efficiency. By proactively reshaping connection-level query patterns via the Ideal Batch strategy and mitigating CPU and disk I/O contention, REEF ensures more stable and predictable performance. This design allows REEF-based DPM to avoid reacting to transient SLO violations or sudden load spikes. Meanwhile, connection-level QoS policies are enforced independently through REEF, enabling precise per-connection targeting. As a result, the REEF-based DPM scheme focuses on providing near-isolated execution environments for connections with stringent SLOs (e.g., high-percentile tail latency targets), enabling stable, efficient QoS enforcement with minimal resource and energy overhead through *QoS-centric workload consolidation.*

To do this, we begin by unifying various QoS metrics, including query throughput, P99 and P99.9 tail latency, and user disengagement ratios (UDRs) [43], into a standardized target: mean query latency SLO. Throughput SLOs are translated using Little's Law [45], while tail latencies and UDRs are mapped to mean latency based on observed latency variance [43, 56]. As shown in Fig. 5(b), REEF stabilizes latency variability under thread packing, enabling accurate conversion of high-percentile or UDR constraints into mean latency using short-term variance measurements.

Each worker thread, typically bound to a client connection, is thus associated with a unified mean latency target. Assuming CPU cores are divided into R core areas, the REEF-based DPM scheme manages groups of threads with similar QoS targets per area. Clustering methods like K-means [51] group threads by QoS similarity. Once clustered, REEF's thread-to-core distributor assigns threads to core areas, dynamically resizing each area based on real-time QoS status. It expands areas experiencing violations and reallocates cores from regions with relaxed targets when all constraints are met (see Section 4). Benefiting from near-optimal C6 residency through on-core Ideal Batching (Section 3.3), REEF supports QoS-centric workload consolidation for enhanced energy efficiency.

4 Implementation and Overhead Analysis

REEF is implemented with 10K LOC in user-space without modifying server applications or the OS kernel. It uses lightweight techniques like syscall interception and Linux signal handling to non-intrusively enforce Ideal Batching and QoS-centric workload consolidation within the application runtime. The following overhead analysis highlights REEF's practicality and efficiency.

On-call worker threads: To support connection-level QoS enforcement under the Ideal Batching strategy, REEF delegates thread-level control to the query batch regulator by injecting its runtime service thread into the application using the *syscall_intercept* library [2]. This enables REEF to operate as an internal thread within the application process and manage worker-thread execution directly. REEF then hotpatches two key control routines in each worker thread's runtime context: *on_call_init()* and *on_call_handler()*. The former, invoked once per thread, registers a Linux signal handler [9] and sets up syscall interception for inter-thread communication. The latter transitions a thread to an idle state after it completes a query batch.

All worker threads are initially placed into the idle state. REEF uses the connection-to-thread mapping to emulate a QoS policy, directing threads to run or sleep via Linux signals and the on-call handler. This enables Ideal Batch enforcement without modifying server application code. Crucially, REEF only intercepts syscalls and uses standard Linux signals; it does not alter application logic or internal data, ensuring compatibility and preserving the functional and security integrity of the server application.

QoS-Centric Workload Consolidation: To prioritize CPU allocation for threads with stricter service-level objectives (SLOs) under the Ideal Batch strategy, REEF supports a QoS-aware consolidation scheme, outlined in Algorithm 1. Worker threads are partitioned into R core areas, A_0 through A_{R-1}, ranked from strictest to most relaxed SLOs. Each area's SLO is defined by the total throughput budget, computed as the sum of per-thread throughput targets.

REEF first transforms each connection-level SLO, expressed in mean latency, into a per-thread throughput budget using Little's Law [45]. It then applies K-means clustering [51] to group threads by their mean latency targets and assigns them to corresponding core areas. Once grouped, the total throughput budget for each core area is determined. To validate feasibility, REEF ensures the sum of area budgets does not exceed the application's sustainable throughput. If it does, a warning is issued, indicating the current system cannot meet the specified QoS requirements.

Next, we empirically profile application throughput across varying core counts. For each area (except the strictest), REEF allocates the minimal cores needed to satisfy its throughput budget. The strictest area receives remaining cores to prioritize its SLOs.

REEF continuously monitors per-thread SLOs at 100 ms intervals. Upon detecting a violation in area A_r, it triggers reactive reallocation by scanning areas with looser SLOs (from A_{R-1} to A_{r+1}) to identify a donor core that can be reassigned without compromising that area's QoS. Specifically, for a candidate donor area A_k, if all its threads currently meet their SLOs with Cores$[k]$ allocated cores, we reference the pre-measured throughput $TP[\text{Cores}[k]-1]$, corresponding to one fewer core. We then compare this value against the throughput budget B_k for A_k, adjusted by an empirical correction factor α, defined as the ratio of $TP[\text{Cores}[k]]$ to the current measured total throughput in A_k. If the condition

$$TP[\text{Cores}[k]-1] > \alpha \cdot B_k$$

Algorithm 1: QoS-Centric Workload Consolidation

```
Input: Total cores $C$, core areas $A_0, \ldots, A_{R-1}$ (ordered from
       strictest to loosest SLO), threads $\mathcal{T}$ with
       connection-level SLOs
Output: Core allocation Cores$[0 \ldots R-1]$
// Initialization based on throughput budget
1  Convert each thread's SLO to its throughput budget;
2  Cluster threads into $R$ areas via K-means on their mean
     latency SLOs;
3  foreach area $A_r$ do
4      Compute total throughput budget $B_r$ from threads in $A_r$;
5  Measure throughput TP$[1 \ldots C]$ under varying core counts;
6  for $r \leftarrow R-1$ down to 1 do
       // Assign minimum cores to meet $B_r$
7      for $c \leftarrow 1$ to $C$ do
8          if $TP[c] \geq B_r$ then
9              Cores$[r] \leftarrow c$;
10             $C \leftarrow C - c$;
11             break;
// Assign remaining cores to the strictest SLO area
12 Cores$[0] \leftarrow C$;
13 while application is running do
       // Reactive core reallocation on SLO violations
14     for $r \leftarrow 0$ to $R-1$ do
15         Monitor SLO compliance in $A_r$;
16         if any thread in $A_r$ violates its SLO then
17             for $k \leftarrow R-1$ down to $r+1$ do
18                 if threads in $A_k$ meet SLOs and a core is
                     transferable then
19                     Cores$[r] \leftarrow$ Cores$[r] + 1$;
20                     Cores$[k] \leftarrow$ Cores$[k] - 1$;
21                     break;
       // Proactive core consolidation if all SLOs are met
22     if all threads meet their SLOs then
23         for $r \leftarrow 0$ to $R-2$ do
24             for $k \leftarrow R-1$ down to $r+1$ do
25                 if a core in $A_k$ can be moved while keeping
                     $A_k$ compliant then
26                     Cores$[r] \leftarrow$ Cores$[r] + 1$;
27                     Cores$[k] \leftarrow$ Cores$[k] - 1$;
28     Wait for next monitoring interval (e.g., 1s);
```

holds, then one core from A_k is considered transferable, and reallocation proceeds accordingly.

To further improve resource efficiency when all SLOs are satisfied, REEF initiates a proactive core consolidation phase. If no SLO violations are detected, it will reallocate CPU cores from lower-priority areas to stricter ones. This is done by scanning core areas from A_{R-1} to A_{r+1} and reassigning cores to higher-priority areas (from A_0 to A_{R-2}), as long as the donor area A_k can still meet

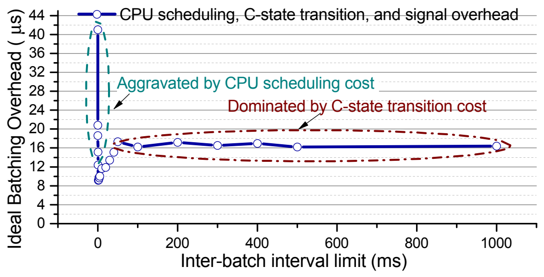

Figure 11: Ideal Batching overhead observed as a function of the inter-batch interval limit.

its SLOs. This rebalancing enhances latency-sensitive workload isolation and stability without compromising QoS.

Overhead Analysis: We evaluate REEF's overhead from two perspectives: (1) the Ideal Batching cost and (2) syscall interception overhead. For the former, we examine cumulative costs from CPU scheduling, C-state transitions, and Linux signals, which vary with inter-batch idle duration. To assess these factors, we conduct experiments on REEF-assisted MongoDB [5] using 512 worker threads serving a YCSB workload [24] (see Section 5.1). To control batching, REEF uses an inter-batch interval limit, manually setting idle durations between 0.2 ms and 1000 ms, with a fixed small batch size (2 queries). This setup allows us to observe the overhead induced by varying degrees of inter-batch idleness. As shown in Fig. 11, when the interval is below 500 μs, most threads remain active, and CPU scheduling dominates latency, reaching up to 40 μs. For intervals exceeding 40 ms, deep C-state residency increases, and scheduling latency becomes significantly lower than the C-state transition overhead. Overall, batching overhead converges to approximately 16 μs, which is negligible compared to the millisecond-scale latencies typical of storage-bound queries. Linux signal handling adds just 2.35 μs of overhead and imposes virtually no energy cost, as signals are only used to wake threads via empty handlers.

Syscall interception adds about 100 ns overhead, acceptable even for soft real-time systems [25]. When MongoDB runs with 512 I/O-bound threads, REEF's syscall interception increases power by just 0.87 W (0.68% of total consumption), confirming its efficiency under thread oversubscription.

5 Performance Evaluation

Test Environment: All REEF evaluation experiments are conducted on a dedicated rack of PowerEdge R630 servers. The application server, which hosts server and co-located applications, features 2×12-core Intel Xeon E5-2650 CPUs, 64 GB RAM, a Broadcom NetXtreme II BCM57810 10 Gb NIC, and 4×1 TB SATA HDDs. Workloads are generated from another server. The server applications are supported by a RAID-0 SSD array comprising five 800 GB SATA MLC SSDs, consolidating all logical volumes. Servers are interconnected via a Dell N4032F switch with 10 Gb peak bandwidth. To evaluate application power usage, we adopt a two-step method. First, CPU and memory power are measured using Model-Specific Registers (MSRs) via the RAPL interface [47, 59]. Second, since the storage I/O capacity is saturated (typically above 95% utilization), we estimate SSD array power using the sum of individual SSD nameplate values [11]. This setup enables accurate assessment of resource and energy efficiency under multi-client, query-intensive workloads, as outlined in Section 3.1.

5.1 Baselines and Methodology

We adopt two representative multi-threaded server applications: the key-value store MongoDB 4.4.3 [5] and the relational database MySQL 8.0.15 [6], driven by a multi-client cloud service workload and an online transaction processing (OLTP) workload, respectively. These applications operate under different user- and kernel-space scheduling policies across the three service layers.

The multi-client cloud workload is generated by Yahoo Cloud Serving Benchmark (YCSB) [24], with each client issuing a Zipf-distributed key-value query stream composed of GET and SET operations (default 50% each), targeting a 150 GB MongoDB dataset. The OLTP workload uses Sysbench [40] with 512 clients accessing a MySQL database of 48 tables, each with 10 million rows (480 million rows total), simulating large-scale mixed read/write OLTP access. To assess resource and energy efficiency under workload consolidation, we also run three compute-intensive PARSEC benchmarks [18] alongside MongoDB or MySQL: *frecmine* (frequent itemset mining), *fluidanimate* (fluid dynamics simulation), and *streamcluster* (streaming clustering).

We evaluate seven widely used or state-of-the-art scheduling strategies spanning QoS-aware, resource-efficient, and DPM-driven approaches, along with some of their reasonable combinations, using MongoDB and MySQL as test platforms.

AppleS [42] is a connection-level QoS scheduler that mitigates excessive client-side I/O parallelism to ensure fair and stable I/O sharing. It improves high-percentile tail latency performance (e.g., P99, P99.9) while maintaining high I/O utilization.

PSLO [44] enforces any-percentile latency SLOs for latency-sensitive (LS) connections while maximizing throughput for best-effort (BE) connections.

UTSLO [44] targets user disengagement ratio (UDR) SLOs for UDR-sensitive (US) connections by regulating per-connection latency distributions. It maximizes BE throughput while meeting UDR targets cost-effectively.

ghOSt(sol) [34] is a kernel-space scheduler that assigns each worker thread a default 1 ms time quantum. It performs preemptive scheduling with low-latency responsiveness.

Cgroup(cpu) [1] is a Cgroup-v2 kernel scheduler ensuring fair CPU sharing among threads based on configurable weights. Equal weights are used by default for all threads.

Demeter [65] is a dynamic power management (DPM) scheme implemented as a workload consolidation and DVFS approach. It enables frequency-blind isolation (FBI) for the server application's worker threads at the CPU-core level (referred to as Demeter(SA)). Specifically, Demeter's FBI mechanism partitions CPU cores into three zones: hot, warm, and cold. Cores in the cold zone remain at the lowest frequency, while hot-zone cores are assigned to key server threads serving latency-sensitive (LS) connections and operate at the highest frequency. Warm-zone cores are managed by DVFS techniques [65] to meet the minimum total throughput target for best-effort (BE) connections. Demeter aims to minimize hot and warm cores to maximize energy savings under SLO constraints. To evaluate compatibility, we also run Demeter under the REEF runtime (i.e., *Demeter+REEF*), demonstrating REEF's ability to integrate with existing DPM strategies. Our QoS-centric workload consolidation approach (*REEF+QoS-WC*) dynamically groups all

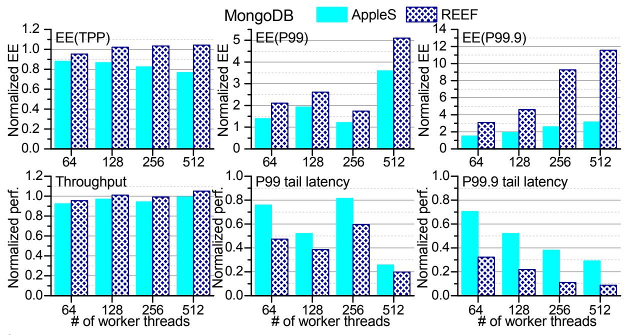

Figure 12: **Energy-efficiency (EE) and performance of AppleS- and REEF-controlled MongoDB, normalized to those of MongoDB without AppleS- and REEF-control.**

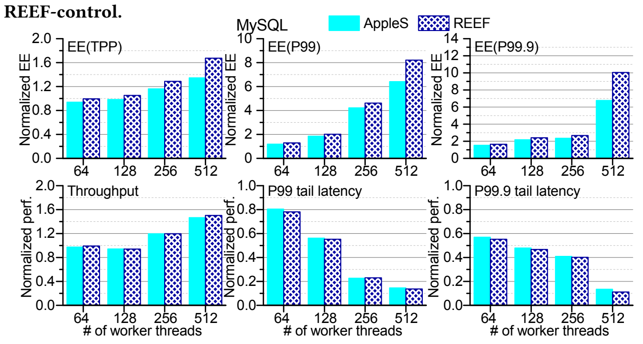

Figure 13: **Energy-efficiency (EE) and performance of AppleS- and REEF-controlled MySQL, normalized to those of MySQL without AppleS- and REEF-control.**

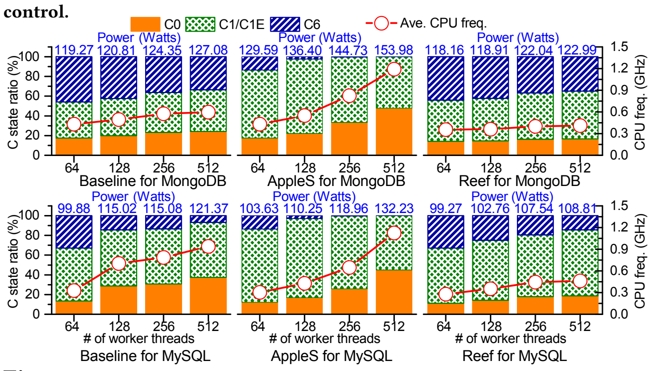

Figure 14: **C-state breakdowns, power consumption, and CPU frequency under different numbers of worker threads.**

worker threads into hot and warm zones based on their individual connection SLOs (e.g., tail latency and throughput). Leveraging REEF's Ideal Batching strategy, the QoS-centric consolidation aims to enhance stringent latency SLO enforcement while maintaining overall QoS and maximizing resource and energy efficiency.

SATORI [62] is an OS-level resource scheduler using Bayesian Optimization to maximize throughput and fairness across co-located applications.

We evaluate REEF's effectiveness in optimizing oversubscribed server applications (MongoDB [5] and MySQL [6]) when coexisting with various scheduling and DPM policies. Our goal is to enhance resource and energy efficiency without compromising QoS targets.

5.2 Connection-Level Scheduling Policies

In this section, we compare REEF against three connection-level QoS scheduling policies: AppleS, PSLO, and UTSLO.

As shown in Fig. 13 and Fig. 12 , for both MySQL and MongoDB, REEF significantly improves all three energy-efficiency (EE) metrics over AppleS, particularly at high thread counts, without degrading performance. For MongoDB, REEF not only enhances throughput and EE, but also gains substantially in tail-latency-related metrics. These improvements result from increased core idle time, reduced oversubscription inefficiencies, and lower kernel scheduling and I/O overhead. Fig. 14 shows that, even with I/O utilization above 95%, MongoDB experiences an average power reduction of 12.10%, a 42.25% drop in CPU frequency, and a 28.86% rise in deep C-state (C6) residency, similar for MySQL under REEF.

PSLO and UTSLO are SLO-centric scheduling policies. PSLO targets maximum throughput for 496 best-effort (BE) connections while enforcing tail-latency SLOs on 16 latency-sensitive (LS) connections: 4 with P95=20 ms, 4 with P99=40 ms, and 8 with P99.9=60 ms. UTSLO seeks to maintain a 0.1% user disengagement ratio (UDR) for 16 UDR-sensitive (US) connections while allocating remaining throughput to 496 BE connections.

We evaluate REEF against and with both PSLO and UTSLO, including some combinations with the OS-level Cgroup (cpu) resource manager. As shown in Fig. 15, PSLO+REEF improves PSLO in *EE*(TPP), *EE*(P99), and *EE*(P99.9) by 29.39%, 2.18×, and 2.99×, respectively, with a 2.99% throughput gain and P99/P99.9 latency reductions of 42.20% and 56.33%. With Cgroup(cpu), PSLO+REEF+Cgroup(cpu) still achieves EE gains, over PSLO, of 19.68%, 76.15%, and 55.30%, and tail latency reductions of 28.06% (P99) and 18.03% (P99.9), with a slight throughput drop of 5.2%, while maintaining all SLOs.

Similarly, UTSLO+REEF improves UTSLO in *EE*(TPP), *EE*(P99), and *EE*(P99.9) by 33.94%, 1.87×, and 5.33×, respectively, with a 4.98% throughput boost and P99 and P99.9 latency reductions of 30.31% and 75.10%, respectively. When paired with Cgroup(cpu), UTSLO+REEF+Cgroup(cpu) achieves EE improvements, over UTSLO, of 29.39%, 57.22%, and 77.11%, a 3.41% throughput gain, and P99 and P99.9 latency reductions of 20.00% and 29.08% respectively, while keeping all US connections within or near the 0.1% UDR threshold. To compare the efficiency of UTSLO and PSLO in meeting tail latency SLOs and UDR targets, we analyze throughput allocation between US/LS and BE connections. Higher efficiency means less throughput to US/LS and more to BE. As shown in Fig. 16, UTSLO and UTSLO+Cgroup(cpu) allocate 1.99K and 2.21K QPS to US connections, respectively. With the REEF control, US throughput drops to 1.16K and 1.10K QPS, while BE throughput rises by 11.16% and 9.26%. Similar trends appear in PSLO: the REEF control reduces LS throughput by up to 51.47% and boosts BE throughput by up to 31.17%. Additionally, REEF significantly lowers CPU frequency and power consumption. PSLO+REEF reduces them by 18.08% and 66.09%, while PSLO+REEF+Cgroup(cpu) achieves 18.53% and 67.54% savings. UTSLO+REEF shows comparable benefits.

5.3 Thread-Level Scheduling Policies

Both ghOSt(sol) and Cgroup(cpu) perform kernel-space thread-level scheduling with distinct goals: ghOSt(sol) focuses on preemptive scheduling to reduce tail latency of short queries (e.g., reads), while Cgroup(cpu) ensures fair CPU time allocation. Since REEF uses the Ideal Batch strategy for fair and energy-efficient query regulation,

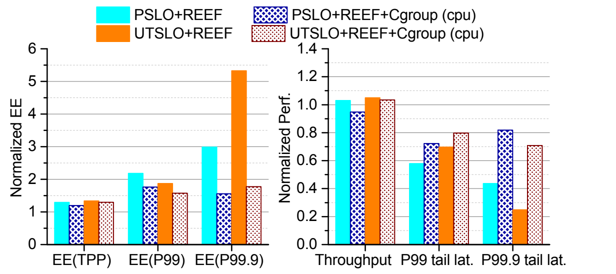

Figure 15: **Energy-efficiency (EE) and performance of four schemes, normalized to those of their corresponding baselines PSLO and UTSLO respectively.**

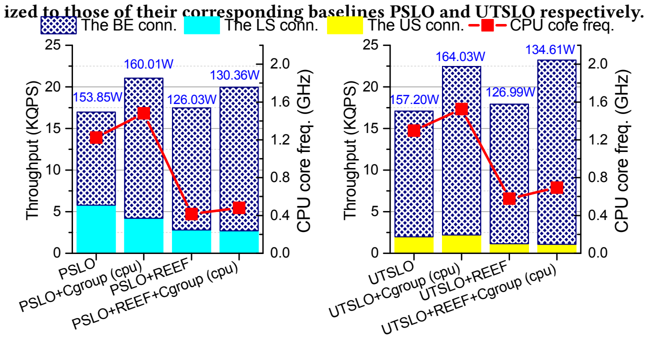

Figure 16: **Throughput breakdowns, CPU frequency and power consumption under different schemes.**

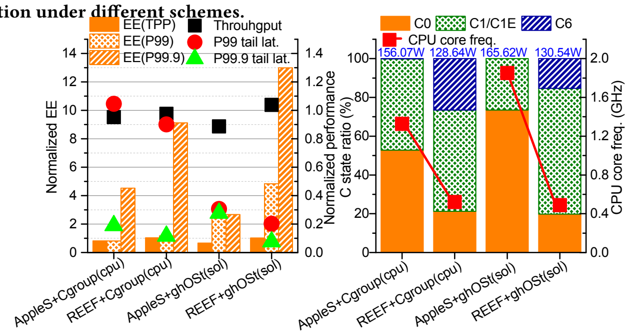

Figure 17: **Energy-efficiency (EE) and performance, normalized to the server application without the REEF control, and C-state breakdowns, power consumption, and CPU frequency across the four schemes.**

we use AppleS+ghOSt(sol) and AppleS+Cgroup(cpu) as baselines for evaluating REEF's optimizations.

With REEF, as shown in Fig. 17, ghOSt(sol)+REEF boosts EE(TPP) and throughput by 53.45% and 17.10%, and improves EE(P99) and EE(P99.9) by 99.08% and 4.86×. P99 and P99.9 tail latencies are reduced by 33.77% and 77.32%, respectively. Similarly, Cgroup(cpu)+REEF increases EE(TPP), EE(P99), and EE(P99.9) by 27.79%, 45.28%, and 2.01×, while increasing throughput by 2.5% and reducing tail latencies by 13.78% and 38.88%, all the while without compromising AppleS fairness. Additionally, REEF enhances CPU frequency, C6 residency, and power efficiency. In the ghOSt(sol) case, CPU frequency and power drop by 73.65% and 21.18%, while C6 residency rises by 15.29%. Comparable improvements are observed with Cgroup(cpu).

5.4 OS-Level Scheduling Policies

We evaluate REEF's compatibility with two advanced scheduling policies, Demeter(SA) and SATORI, focused on dynamic power management (DPM) and resource efficiency. For Demeter(SA), we configure MongoDB with 16 latency-sensitive (LS) connections, each with the same tail latency SLOs as in Section 5.2, and 496 best-effort (BE) connections with a minimum throughput requirement of 12.5 KQPS. We compare three setups: PSLO+Demeter(SA), PSLO+REEF+Demeter(SA), and PSLO+REEF+QoS-WC, by assessing energy and resource efficiency. As shown in Fig. 18, REEF transforms self-serving threads (94% of all threads) into on-call threads activated solely via Ideal Batch scheduling under PSLO. This shift cuts kernel scheduling overhead and thread contention, reducing latency variability and improving SLO enforcement. The impact is visible in the throughput allocated to LS connections in Fig. 18(a). Fig. 18(c) and (d) highlight that REEF enhances Demeter(SA)'s efficiency: EE(TPP), EE(P99), and EE(P99.9) improve by 20.43%, 75.16%, and 43.46%, while P99 and P99.9 latencies drop by 27.91% and 11.99% with stable average latency. Additionally, PSLO+REEF+Demeter(SA) reduces CPU frequency and power by 43.02% and 18.24%, respectively, and raises C6 residency by 20.65% over PSLO+Demeter(SA).

Next, we further validate the effectiveness of REEF-enabled QoS-centric workload consolidation (*REEF+QoS-WC*) in dynamically isolating LS and BE workloads at the core level. Specifically, we tighten the tail latency SLOs by reducing the P99 targets of 4 LS connections from 40 ms to 30 ms, and the P99.9 targets of another 8 LS connections from 60 ms to 50 ms. To visualize real-time SLO compliance, we normalize tail latency SLOs as $\max\limits_{1\le i\le n} \frac{x_i}{1-p_i}$, where n is the number of LS connections, x_i is the observed violation ratio, and p_i is the SLO-allowed ratio of tail latency SLO violations. Compliance is guaranteed if this value remains at or below 1. For best-effort (BE) connections, throughput SLO compliance is defined as the actual throughput divided by the target, satisfied when the value is greater than or equal to 1. As shown in Fig. 19, both *PSLO+Demeter+REEF* and *PSLO+Demeter* expand the cold core area under SLO constraints, enabling more cores to operate at minimal frequencies for energy savings. However, after tightening SLOs, both schemes see a sharp reduction in the cold area, although *PSLO+Demeter+REEF* maintains a significantly larger cold area due to REEF's high thread-level processing efficiency enabled by the Ideal Batch strategy. In contrast, *PSLO+REEF+QoS-WC* maintains stable hot and warm areas by proactively expanding the hot area to isolate LS-serving threads from BE interference, thereby ensuring consistent tail latency enforcement (see Section 4). Moreover, by decoupling QoS enforcement from power management, REEF resolves the DPM dilemma and ensures predictable performance across varying levels of resource provisioning in oversubscribed server environments. Consequently, as shown in Fig. 20, *PSLO+REEF+QoS-WC* exhibits minimal SLO violation spikes compared to the other schemes while delivering 7.60% higher BE throughput. Importantly, the power consumption of *PSLO+REEF+QoS-WC* remains close to that of *PSLO+Demeter+REEF* and significantly lower than standalone *Demeter* (see Fig. 18)(b), as REEF's Ideal Batch strategy effectively exploits core-level idleness, allowing cores to enter sleep states or operate at lower frequencies with high probability.

To further assess REEF's ability to improve resource efficiency and energy savings in a consolidated environment, we colocate MySQL with three compute-intensive applications: *frecmine*, *streamcluster*, and *fluidanimate*. We compare REEF with SATORI,

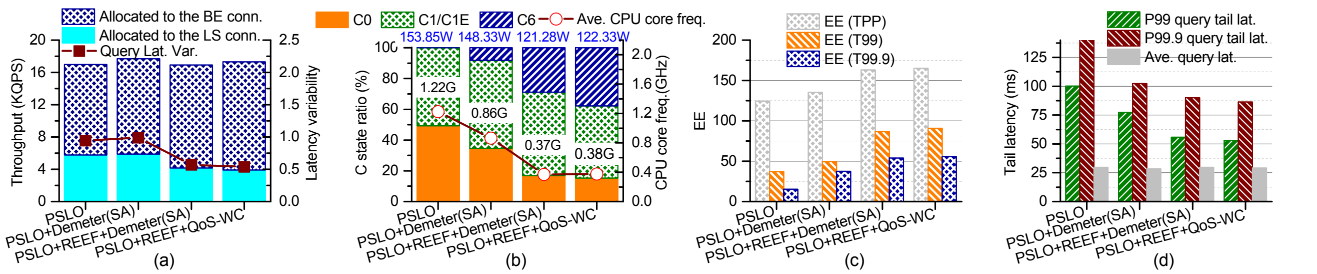

Figure 18: Query latency CDF, *P*99 and *P*99.9 tail latencies, latency variability measured by the coefficient of variation (CV) [4], energy-efficiency (EE-metrics), as well as the power consumption and C-state breakdowns under different schemes.

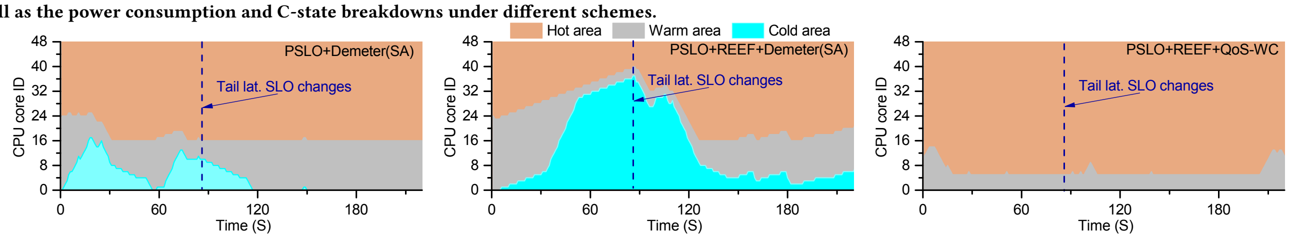

Figure 19: Comparison in core-level isolation under different schemes.

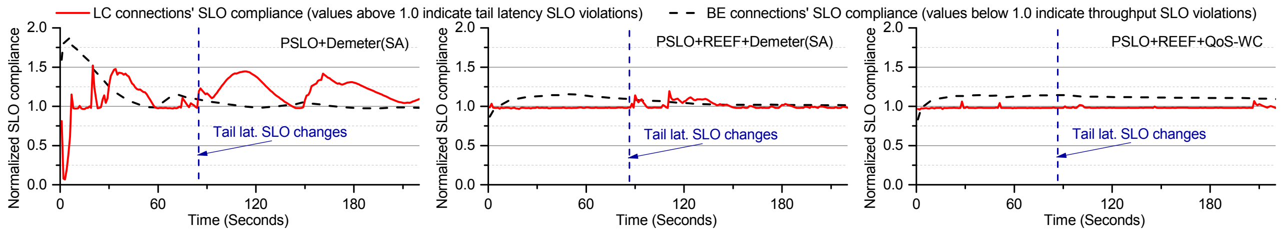

Figure 20: Real-time SLO enforcement under different schemes, with tighter tail latency SLOs applied at around 85s by reducing 4 LS connections' P99 targets from 40ms to 30ms and 8 others' P99.9 targets from 60ms to 50ms.

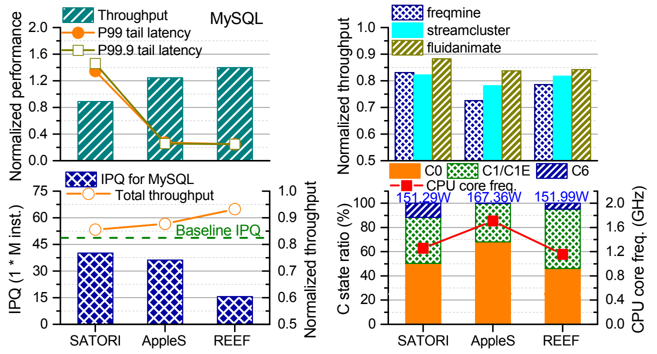

Figure 21: Performance of MySQL and three collocated applications, including normalized total throughput, instructions per query (IPQ), CPU frequency, power consumption, and C-state breakdowns across different schemes.

which optimizes system-wide throughput across co-located workloads, and AppleS, which focuses on application-level performance under fixed resources.

As shown in Fig. 21, AppleS improves MySQL's throughput and reduces P99 and P99.9 latencies relative to SATORI, but REEF achieves the best results across all metrics. Specifically, REEF increases MySQL throughput by 57.38% and lowers P99 and P99.9 latencies by 81.64% and 83.00%, respectively, compared to SATORI. By eliminating inefficient processing, REEF also reduces MySQL's average instructions per query (IPQ) by 61.09%, allowing better CPU sharing among co-located apps. This leads to higher throughput for the three co-runners than AppleS, though slightly below SATORI (by 3.54% on average).

While REEF consumes 0.7 W (0.46%) more power than SATORI, its 9% higher normalized total throughput yields better overall energy and resource efficiency. Moreover, combining REEF with SATORI boosts co-runner throughput by an additional 2.78% on average, with minimal impact on server performance. Experiments with MongoDB show similar results, though the corresponding data is not shown here due to space limit.

6 Conclusion

In this paper, we present REEF, a cross-layer, energy-aware scheduling framework that addresses the complex challenges of thread oversubscription in modern server applications. REEF combines Ideal Batch scheduling with lightweight syscall interception and signal-based thread control to reshape execution patterns, achieving near-optimal C6 residency and enhancing both resource and energy efficiency without compromising connection-level QoS. It also supports fine-grained strategies like QoS-centric workload consolidation and offers an extensible foundation for holistic resource and power management. Evaluations on representative platforms, including MongoDB and MySQL, show that REEF improves energy-efficiency metrics by up to 5.33×, reduces tail latencies by up to 83%, and lowers CPU power consumption by up to 73.65%, while maintaining or boosting throughput.

7 Acknowledgments

This work is supported in part by the US National Science Foundation grant CCF-SHF2226117. We thank the anonymous reviewers for their valuable comments.

References

[1] 2015. Cgroups v2. https://www.kernel.org/doc/Documentation/cgroup-v2.txt.

[2] 2019. The system call intercepting library . https://github.com/pmem/syscall_intercept.

[3] 2021. Intel Tweaking Ice Lake Xeon Linux Power Management Code For Higher C6 Latency. https://www.phoronix.com/news/Intel-C6-Ice-Lake-Xeon-Idle.

[4] 2022. Coefficient of variation. https://en.wikipedia.org/wiki/Coefficient_of_variation.

[5] 2022. MongoDB. http://www.mongodb.org/.

[6] 2022. MySQL. http://www.mysql.com.

[7] 2022. Weighted round robin. https://en.wikipedia.org/wiki/Weighted_round_robin.

[8] 2023. cgroups - Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.

[9] 2023. signal(7). https://man7.org/linux/man-pages/man7/signal.7.html.

[10] 2024. CPU idle power saving methods for real-time workloads. https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/cpuidle?utm_source=chatgpt.com.

[11] 2024. Intel® Solid-State Drive DC S3500 Series. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-s3500-spec.pdf.

[12] 2025. *SCHED(7) Linux Programmer's Manual*.

[13] Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, and Timothy Zhu. 2020. Peafowl: In-application cpu scheduling to reduce power consumption of in-memory key-value stores. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[14] Ali Bakhoda, John Kim, and Tor M. Aamodt. 2010. Throughput-effective on-chip networks for manycore accelerators. In *In 43rd Annual IEEE/ACM international symposium on microarchitecture*.

[15] Andrea Bartolini, Matteo Cacciari, Andrea Tilli, and Luca Benini. 2012. Thermal and energy management of high-performance multicores: Distributed and self-calibrating model-predictive controller. *IEEE Transactions on Parallel and Distributed Systems* 24, 1 (2012), 170–183.

[16] Zouhir Bellal, Laaziz Lahlou, Nadjia Kara, and Ibtissam El Khayat. 2024. Gas: DVFS-Driven Energy Efficiency Approach for Latency-Guaranteed Edge Computing Microservices. *IEEE Transactions on Green Communications and Networking* (2024).

[17] Reinaldo Bergamaschi, Guoling Han, Alper Buyuktosunoglu, Hiren Patel, Indira Nair, Gero Dittmann, Geert Janssen, Nagu Dhanwada, Zhigang Hu, Pradip Bose, and John Darringer. 2008. Exploring power management in multi-core systems. In *In Proceedings of the Asia and South Pacific Design Automation Conference*.

[18] Christian Bienia and Kai Li. 2009. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *In Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation, vol. 2011, p. 37*.

[19] Jian Chen and Lizy Kurian John. 2011. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *In Proceedings of the international conference on Supercomputing*.

[20] Xi E Chen and Tor M. Aamodt. 2009. A first-order fine-grained multithreaded throughput model. In *In Proceedings of the IEEE international symposium on high performance computer architecture (HPCA)*.

[21] Chih-Hsun Chou, Laxmi N. Bhuyan, and Daniel Wong. 2019. μdpm: Dynamic power management for the microsecond era. In *In Proceedings of the IEEE international symposium on high performance computer architecture (HPCA)*.

[22] Chih-Hsun Chou, Daniel Wong, and Laxmi N. Bhuyan. 2016. Dynsleep: Fine-grained power management for a latency-critical data center application. In *In Proceedings of the International Symposium on Low Power Electronics and Design*.

[23] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. 2011. Pack & cap: adaptive dvfs and thread packing under power caps. In *In Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[25] S. S. Craciunas, C. M. Kirsch, and H. Röck. 2008. I/O Resource Management Through System Call Scheduling. *SIGOPS Oper. Syst. Rev* 42, 5 (2008), 44–54.

[26] Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE micro* 28(3) (2008), 42–53.

[27] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in Multi-Resource clusters. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

[28] Jim N. Gray, Raymond A. Lorie, and Gianfranco R. Putzolu. 1975. Granularity of locks in a shared data base. In *In Proceedings of the International Conference on very large data base (VLDB)*.

[29] A. Gulati, I. Ahmad, and C. A. Waldspurger. 2009. PARDA: proportional allocation of resources for distributed storage access. In *Proccedings of the conference on File and storage technologies (FAST)*.

[30] A. Gulati, A. Merchant, and P. J. Varman. 2007. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*.

[31] A. Gulati, A. Merchant, and P. J. Varman. 2010. mClock: handling throughput variability for hypervisor IO scheduling. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*.

[32] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Foundations and Trends in Databases* 1, 2 (2007), 141–259.

[33] Hang Huang, Jia Rao, Song Wu, Hai Jin, Hong Jiang, Hao Che, and Xiaofeng Wu. 2021. Towards exploiting CPU elasticity via efficient thread oversubscription. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing (HPDC)*.

[34] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakha, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[35] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. 2006. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*.

[36] Veronia Iskandar, Cherif Salama, and Mohamed Taher. 2022. Dynamic thread mapping for power-efficient many-core systems under performance constraints. *Microprocessors and Microsystems* 93 (2022).

[37] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[38] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. 2012. Measuring interference between live datacenter applications. In *In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*.

[39] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast analytical power management for latency-critical systems. In *In Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[40] Alexey Kopytov. 2025. Sysbench: a system performance benchmark. https://github.com/akopytov/sysbench.

[41] Hsiang-Tsung Kung and John T. Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 2 (1981), 213–226.

[42] Ning Li, Hong Jiang, Hao Che, Zhijun Wang, and Minh Q. Nguyen. 2022. Improving scalability of database systems by reshaping user parallel I/O. In *Proceedings of the 3th European conference on Computer systems (EuroSys)*.

[43] Ning Li, Hong Jiang, Hao Che, Zhijun Wang, Minh Q. Nguyen, and Todd Rosenkrantz. 2023. User Disengagement-Oriented Target Enforcement for Multi-Tenant Database Systems. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[44] N. Li, H. Jiang, D. Feng, and Z. Shi. 2016. PSLO: Enforcing the X^{th} Percentile Latency and Throughput SLOs for Consolidated VM Storage. In *Proceedings of the 3th European conference on Computer systems (EuroSys)*.

[45] JDC Little. 1961. NA proof for the queuing formula: L= λW. *Operations research* 9, 3 (1961), 383–387.

[46] Du Liu, Jing Wang, Xinkai Wang, Chao Li, Lu Zhang, Xiaofeng Hou, Xiaoxiang Shi, and Minyi Guo. 2024. Improving the Efficiency of Serverless Computing via Core-Level Power Management. In *Proceedings of the 2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 125–135.

[47] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 301–312.

[48] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *In Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*.

[49] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. 2024. Efficient microsecond-scale blind scheduling with tiny quanta. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[50] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. 2011. Scalable power control for many-core architectures running multi-threaded applications. In *In Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*.

[51] J. B. MacQueen. 1967. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, Vol. 1. University of California Press, 281–297.

[52] David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2009. Powernap: eliminating server idle power. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[53] D. Meisner, B. T. Gold, and T. F. Wenisch. 2011. The PowerNap server architecture. *ACM Transactions on Computer Systems* 29, 1 (2011), 1–24.

[54] David Meisner, Christopher M. Sadler, Wolf-Dietrich Weber Luiz André Barroso, and Thomas F. Wenisch. 2011. Power management of online data-intensive services. In *In Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*.

[55] David Meisner and Thomas F. Wenisch. 2012. Dreamweaver: architectural support for deep sleep. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[56] M. Nguyen, S. Alesawi, N. Li, H. Che, and H. Jiang. 2018. Forktail: a black-box fork-join tail latency prediction model for user-facing datacenter workloads. In *Proceedings of the ACM International Symposium on High Performance Distributed Computing (HPDC)*.

[57] George Papadimitriou, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. 2019. Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore CPUs. In *In Proceedings of the IEEE international symposium on high performance computer architecture (HPCA)*.

[58] Tirthak Patel and Devesh Tiwari. 2020. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *In Proceedings of the IEEE international symposium on high performance computer architecture (HPCA)*.

[59] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. 2015. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[60] W. Reda, M. Canini, L. Suresh, D. Kostić, and S. Braithwaite. 2017. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of the 3th European conference on Computer systems (EuroSys)*.

[61] Basireddy Karunakar Reddy, Amit Kumar Singh, Dwaipayan Biswas, Geoff V. Merrett, and Bashir M. Al-Hashimi. 2017. Inter-cluster thread-to-core mapping and DVFS on heterogeneous multi-cores. *IEEE Transactions on Multi-Scale Computing Systems* 4 (2017), 369–382.

[62] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2021. Satori: efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains. In *In Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*.

[63] Jovan Stojkovic, Nikoleta Iliakopoulou, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2024. EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency. In *Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 471–486.

[64] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. Dynamollm: Designing LLM Inference Clusters for Performance and Energy Efficiency. In *Proceedings of the 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1348–1362.

[65] Wenda Tang, Yutao Ke, Senbo Fu, Hongliang Jiang, Junjie Wu, Qian Peng, and Feng Gao. 2022. Demeter: QoS-aware CPU scheduling to reduce power consumption of multiple black-box workloads. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[66] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. 2015. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *In Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*.

[67] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling multicore databases via constrained parallel execution. In *In Proceedings of the International Conference on Management of Data*.

[68] Rafael J. Wysocki. 2020. *intel_idle CPU Idle Time Management Driver*.

[69] Mingli Xie, Dong Tong, Kan Huang, and Xu Cheng. 2014. Improving system throughput and fairness simultaneously in shared memory cmp systems via dynamic bank partitioning. In *In Proceedings of the IEEE international symposium on high performance computer architecture (HPCA)*.

[70] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. 2018. dcat: Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *Proceedings of the 3th European conference on Computer systems (EuroSys)*.

[71] Jawad Haj Yahya, Haris Volos, Davide B. Bartolini, Georgia Antoniou, Jeremie S. Kim, Zhe Wang, Kleovoulos Kalaitzidis, Tom Rollet, Zhirui Chen, Ye Geng, Onur Mutlu, and Yiannakis Sazeides. 2022. AgileWatts: An Energy-Efficient CPU Core Idle-State Architecture for Latency-Sensitive Server Applications. In *In Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[72] Xin Zhan, Reza Azimi, Svilen Kanev, David Brooks, and Sherief Reda. 2016. Carb: a c-state power management arbiter for latency-critical workloads. *IEEE Computer Architecture Letters* 16, 1 (2016), 6–9.

[73] X. Zhan, R. Azimi, S. Kanev, D. Brooks, and S. Reda. 2017. CARB: A C-State Power Management Arbiter for Latency-Critical Workloads. *IEEE Computer Architecture Letters* 16, 1 (2017), 6–9.

[74] T. Zhu, D. S. Berger, and M. Harchol-Balter. 2016. SNC-Meister: Admitting More Tenants with Tail Latency SLOs. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[75] T. Zhu, M. A. Kozuch, and M. Harchol-Balter. 2017. WorkloadCompactor: Reducing datacenter cost while providing tail latency SLO guarantees. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.

[76] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. 2014. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the ACM symposium on Cloud computing (SoCC)*.