CoreSync: A Protocol for Joint Core Scheduling and Overload Control of μs-Scale Tasks

Bhaskar Pardeshi, Eric Stuhr and Ahmed Saeed
Georgia Institute of Technology
Atlanta, Georgia

***Abstract*—Modern servers employ multiple resource management algorithms, including fast core schedulers and overload controllers to balance application performance and resource utilization. Individual algorithms and their amalgamation are required to meet these tight performance requirements. In this paper, we demonstrate that state-of-the-art core schedulers and overload controllers produce poor performance when deployed simultaneously. Fundamentally, the design assumptions of each controller are violated by the other controller. An overload controller assumes that all resources are dedicated to an application, while a core scheduler assumes that all incoming load will be admitted. To overcome this fundamental limitation, we present CoreSync, a server-driven credit-based protocol for joint core scheduling and overload control. CoreSync relies on the basic idea that the admitted load should be proportional to the allocated resources. However, strict proportionality can lead to low utilization when admitted load does not materialize at the server (e.g., when demand drops). Thus, CoreSync uses partial proportionality to balance latency, throughput, and utilization. Our evaluation across synthetic and real-world workloads shows that CoreSync outperforms state-of-the-art schedulers and overload controllers. In particular, in overload scenarios, CoreSync improves throughput by up to 6%. At low loads, CoreSync reduces the 99th percentile latency by up to 1.7× and improves CPU utilization by up to 1.4×.**

I. Introduction

Large-scale servers operate under strict requirements on the performance they deliver, while maintaining high levels of utilization. Operators expect requests to finish within tight Service Level Objectives (SLO) while consuming exactly the amount of resources they need, no more and no less. To satisfy these requirements, two key controllers are employed: high-frequency CPU core scheduling to precisely allocate only the needed amount of resources for an application [1]–[8] and overload control to only admit load that can be processed by a server while meeting its SLO [9]–[14]. A core scheduler aims to maximize CPU utilization by allocating idle CPU cycles to preemptable best-effort applications when it is not being used by a latency-critical application. An overload control algorithm aims to balance the throughput and latency of latency-critical applications when a server is overloaded, admitting enough load to maximize application throughput without building long queues. Previous work studies and develops each controller independently.However, both types of controller can and should be employed simultaneously.

In this paper, we find that existing core schedulers and overload controllers produce poor performance when deployed jointly. The fundamental reason behind poor performance is that the assumptions made in the design of each controller are violated by the other controller. In particular, core schedulers assume that all load will be admitted to the server, ignoring the throughput of the latency-critical application as an evaluation metric. On the other hand, overload control algorithms assume that the entire server capacity is allocated to the application, attributing changes in performance solely to their admission control decisions. However, when the two algorithms are combined, the overload controller can react to a reduction in the number of allocated cores by reducing the number of admitted requests, harming the throughput of latency-critical applications. Moreover, existing core allocation policies react to short-term load at the server, ignoring potential requests in flight, leading to high tail latency even at low loads. We demonstrate that the tradeoffs between the throughput and latency of a latency-critical application and the overall CPU utilization are fundamental to state-of-the-art controllers and cannot be overcome with simple parameter tuning (§II).

To address these challenges, we develop CoreSync, server-driven credit-based protocol that jointly optimizes core scheduling and overload control (§III). CoreSync employs a delay-based controller for adjusting the total volume of admitted load (i.e., the size of the credit pool), issuing more credits if delay is below a certain target and fewer credits when delay is above the target. Moreover, it employs a simple delay-based core scheduling policy, allocating additional cores if the maximum per-core delay exceeds a configurable target, much smaller than the target for credit control. The basic idea behind CoreSync is to maintain proportionality between the admitted load and the allocated CPU capacity. Strict proportionality is undesirable. For example, overcommitment in admission control is essential to ensure high utilization at low loads [9], [10], [15], [16]. Strict proportionality can leave allocate cores only to leave them idle when admission is overcommited at low loads. Thus, CoreSync employs partial proportionality where the majority of admitted load is proportional to allocated cores and a minority is overcommitted. To reduce interference between the two controllers, CoreSync prevents core deallocation if the incoming load to the server matches its capacity. It detects such scenarios based on the behavior of the admission controller. Specifically, successive reductions in the number of issued credits are used as an indication of persistent overload, requiring the allocation of all cores to the latency critical application.

979-8-3315-0376-5/25/$31.00 ©2025 IEEE

We implemented CoreSync as a part of the Caladan libOS [2]. Our extensive evaluation of various workloads demonstrates that CoreSync always lies on the Pareto frontier of the three evaluation metrics: the latency and throughput of the latency critical application and overall CPU utilization. In particular, CoreSync outperforms four state-of-the-art core schedulers when they are deployed with the Breakwater, a state-of-the-art overload controller. We compare CoreSync to the core scheduling policies used in the Shenango [1] and Caladan [2] systems. We also compare CoreSync with the proactive Utilization Range and Delay Range policies [3]. When load is below the capacity of the server, all systems provide comparable throughput. However, CoreSync matches the low CPU utilization of Shenango and Caladan, while providing up to 1.7× lower 99th percentile latency. Moreover, it improves CPU utilization by 1.4× compared to Utilization Range and Delay Range, while providing comparable tail latency. When the system is overloaded, all systems achieve comparable latency. However, CoreSync delivers up to 6% higher throughput than Shenango, Utilization Range, and Delay Range, while remaining within 2–3% of Caladan. The CoreSync implementation and reproduction scripts are available at https://github.com/GT-ANSR-Lab/CoreSync.git.

II. Background and Motivation

A. Background

High-frequency core scheduling. The basic idea of high-frequency core allocation algorithms is to maximize CPU utilization, by allocating idle resources to a preemptable best-effort application, without harming the performance of the latency critical application. There has been significant interest in supporting this behavior in software [1]–[6], [17] and hardware [7], [8], [18], [19]. Fast core allocation is combined with load balancing, typically employing work stealing [20] or Join-Bounded-Shortest-Queue (JBSQ) [8], [21]. We focus on fast scheduling of tasks that run to completion [1]–[3], [19].

A recent study compares the performance of multiple core allocation policies, classifying them into two categories: reactive policies and proactive policies [3]. Both categories allocate additional cores to the latency-critical application, one at a time, upon detecting high delay or high CPU utilization. The main difference between the two classes is how they deallocate cores. Reactive policies deallocate a core when the core fails to find work. Thus, reactive cores waste resources searching for work to steal from other cores, potentially busy-polling for long durations before a core is deallocated. Proactive policies observe the average load on the system, across all cores, and deallocate cores when the load drops below a configurable threshold. Thus, proactive policies avoid the overhead of searching for work when the load is lower than the capacity. Average core utilization and average queueing delay across cores can be used to observe the average load on the system.

Overload control. The fundamental idea behind overload control is shedding load that exceeds the server's capacity, potentially redirecting it to other replicas or triggering the allocation of additional resources. Overload controllers can be used directly by clients or indirectly through load balancers. There are three broad categories of overload controllers: 1) client-based, where clients estimate the state of the server to determine a limit on the number of requests it can have in flight to the server [12], [13], 2) active queue management (AQM) at the server [11], [22], where the server drops requests if it estimates that they will violate their SLO, and 3) server-driven admission control, where the server issues credits to clients, indicating that it can receive requests [9], [10]. Overload control is typically part of the RPC communication logic. The admission control and load shedding decisions made by the server are communicated to clients or load balancers, depending on the deployment, as a part of the RPC protocol.

Combining both controllers changes their individual assumptions, creating a three-way tradeoff. Core scheduling policies are designed assuming that all requests from the latency-critical application will be admitted to the server, implying that the throughput of the latency-critical application is a non-concern. Thus, core schedulers are designed to balance the utilization of idle CPU capacity and the latency of the latency-critical application. In contrast, overload controllers are typically designed to balance the latency and throughput of a latency-critical application. Overload controllers are designed assuming that all server resources are allocated to the latency-critical application. Thus, the utilization of idle capacity at low loads is a non-concern for overload controllers.

The joint deployment of overload control and fast core scheduling changes their individual assumptions. In particular, it introduces a tradeoff between CPU utilization and the throughput of the latency critical application. For example, under overload, an overload controller that employs an Additive Increase / Multiplicative Decrease (AIMD) admission controller can momentarily reduce load on a server when it multiplicatively decreases load admission. Such a momentary reduction in load can lead fast core schedulers to reduce the number of allocated cores to the latency critical application, leading to high latency or even throughput degradation. Moreover, fast core schedulers only take into account load at the server, ignoring load in flight, leading to poor tail latency at low loads. We quantify these problems next.

B. Analysis of the Tradeoff Space

We demonstrate that existing schemes cannot jointly optimize the latency and throughput of latency-critical applications as well as the utilization of idle CPU cycles. We explore the tradeoff space by attempting to configure state-of-the-art algorithms to balance all three metrics. We choose the Caladan core allocation policy as representative of reactive core allocation policies. We choose the Utilization Range policy as a representative of proactive core allocation policies. We employ Breakwater as an overload control system. All three controllers are implemented in the Caladan library operating system. We evaluate their performance using a synthetic workload composed of requests whose service time is exponentially distributed with an average execution time of $10\mu s$. Our evaluation setup is presented in detail later (§IV-A). We evaluate server performance in two broad scenarios: *below capacity* (not overloaded) and *near or beyond capacity* (overloaded). In the *below capacity* case, we classify loads of 20%, 50%, and 80% of server capacity as low, medium, and high, respectively. In the near or beyond capacity case, the load is 100% or more of server capacity.

Throughput v/s utilization. When load is below capacity, the overload controller admits all load, leading all policies to achieve the same throughput. Their achieved utilization depends on their configurations and workload burstiness [3]. When load exceeds capacity, utilization is typically near 100% as load is enough to keep all cores busy. However, we observe that different core scheduling policies yield different levels of throughput. Proactive core scheduling policies achieve 2-4% worse throughput than reactive policies. To understand this tradeoff, consider a server with 4 CPU cores, each handling 100k RPS for 10 µs requests (total 400k RPS), and a 10 µs RTT with 100 clients. Such a server needs at least 8 in-flight requests per RTT for full utilization. To maintain consistently high utilization, controllers allow queueing up to a target delay (e.g., 80 µs) and overcommit credits [9]. Here, the server can distribute up to 36 credits; even if all are used at once and perfectly load balanced, the maximum queueing delay is 80 µs (9 requests/core, 9th waits for 8×10 µs).

Assume the 100 clients send bursts of up to 36 requests in total. Since such bursts keep delay below the 80 µs target, the controller remains in additive-increase mode, growing the credit pool while the core scheduler keeps all cores allocated. When the pool reaches 436 credits, assume that all clients send a large burst simultaneously, fully utilizing the overcommitted credits. With perfect load balancing, each core gets 109 requests, and the last waits 1080 µs (uniform service time). This delay triggers a 25% multiplicative decrease in admitted load, reducing credits to 327 [9]. The sudden drop causes some steady-rate clients to send fewer or no requests, lowering utilization. A proactive policy will react to this load drop by deallocating a core. Arriving requests face longer queues as they await the core to be reallocated. Long queues trigger multiplicative decrease. This behavior continues to repeat, lowering the average number of admitted requests (i.e., leading to low throughput). This problem is amplified with aggressive AIMD-based overload controller which can revoke a large number of credits when overload is detected (e.g., with a multiplicative decrease factor of 0.5). However, we still observe it even with a smooth AIMD controller using a multiplicative decrease factor of 0.02; the value used in all our experiments. Reactive policies do not suffer from the same problem, because they poll for work before deallocating cores. Busy polling can lower utilization at low loads but improves throughput when the system is overloaded. Proactive scheduling can improve utilization at low loads but harms throughput at high loads. We have separately shown analytically that interference between controllers considerably degrades their worst-case performance [23].

Utilization v/s latency. When load is below server capacity, existing core scheduling policies provide knobs to balance these two objectives. Caladan allocates cores in reaction to queueing delay and polls for work before deallocating a core. Increasing the queueing delay threshold improves utilization at the expense of latency. On the other hand, increasing the polling duration improves tail latency at the expense of utilization. We argue that existing policies are myopic, leading to a strict prioritization between latency and utilization. Increasing the scheduler's reaction time degrades utilization by making it harder to park cores but does not necessarily improve latency because it does not account for in-flight requests. To highlight this behavior, we study the performance of the Caladan allocation policy while varying its busy polling interval for a synthetic workload (Figure 1). We contrast its performance with that of our proposal, CoreSync, to highlight the opportunity gap.

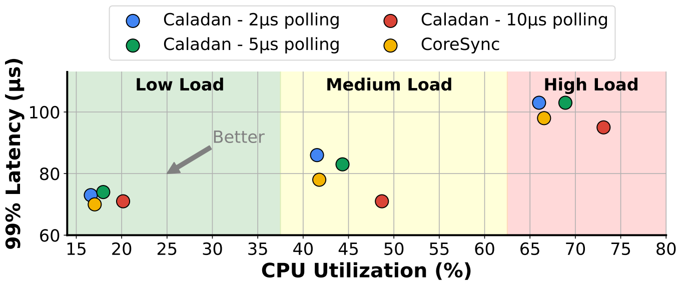

Fig. 1: Latency v/s Utilization, below capacity, for exponentially distributed workload with an average of $10\mu s$. Increasing the polling interval of Caladan improves latency at the expense of utilization. The performance of CoreSync dominates the performance of Caladan under all configurations of Caladan.

Clearly, increasing the polling interval sacrifices utilization to improve tail latency. However, CoreSync dominates the performance of Caladan under all configurations, with several Caladan configurations not being on the Pareto frontier of latency and utilization. Under all loads, CoreSync achieves comparable or better latency than Caladan with a $10\mu s$ polling interval while improving utilization by up to 13% (that is, using one less core to handle the same load). Slight improvements in tail latency are possible because CoreSync takes into account requests in flight when making its core allocation decision. Increasing the polling interval beyond $10\mu s$ does not improve Caladan's latency, but can lead to poorer utilization. Thus, we conclude that configuring existing knobs for balancing utilization and latency in core allocation algorithms might not lead to Pareto optimal performance, requiring finer grain control that takes into account overall load on the system, including requests in flight.

Throughput v/s latency. Overload controllers provide knobs to navigate the tradeoff between throughput and latency when the server operates near its capacity. For example, Breakwater is delay-based, generating more credits when delay is below a given target delay and less credits when delay is higher than the target delay. Configuring Breakwater to have a smaller target delay reduces tail latency but harms throughput. The recommended configuration for a workload with an average service time of $10\mu s$ is a target delay of $80\mu s$. Setting a smaller target delay for Breakwater (e.g., $40\mu s$) reduces throughput by up to 10% at high loads while reducing tail latency by 20-30%. In contrast, sacrificing latency can improve throughput. Thus, it is possible to achieve good throughput and utilization if the tail latency is doubled. However, there are no existing policies that balance all three metrics.

Summary. Different performance tradeoffs are observed depending on load. At low loads, existing schemes maximize throughput but sacrifice latency and/or utilization. In overload scenarios, existing schemes optimize latency but sacrifice throughput. Attempting to optimize throughput when the system is overloaded leads to worse utilization at low loads. Moreover, we observe that existing algorithms can be tuned to optimize for a specific performance metric. However, their performance is not Pareto optimal. *We conclude that a new policy is needed to optimize core scheduling decisions given overload control decisions, and vice versa, balancing throughput, latency, and utilization.*

C. Challenges

The fundamental limitation of existing core schedulers and overload controllers is that the design of one controller ignores the behavior of the other. Moreover, the desired combined behavior of both controllers depends on the load level. In particular, coordination between overload control and core schedule needs to differentiate between manageable load and unmanageable load made manageable by the overload controller. This issue leads to the following challenges:

1. Differentiating between persistent overload of all available cores and momentary spikes in demand with few allocated cores. Both scenarios can result in a high delay. Moreover, an overloaded system can still experience low delay when a good overload controller is used because the admitted load exactly matches the available capacity. An overload controller should admit less load only when the system is truly overloaded. On the other hand, a core allocator should never deallocate cores when the system is overloaded. Thus, we need a signal that detects persistent overload scenarios.

2. Coordinating two controllers that operate at comparable timescales. Core allocation and overload control belong to different parts of the system. Core allocation is typically a function of the operating system. Overload control is typically part of the RPC layer. Tight integration of their logic can lead to untenable solutions that can never be deployed in practice (e.g., giving the RPC layer privileged access to the operating system). Moreover, both controllers operate on microsecond timescales. A complex coordination mechanism can easily incur significant CPU overhead, potentially harming performance. Thus, coordinating between the two controllers requires a non-intrusive approach that does not incur significant processing overhead.

3. Short request service times. Our focus is on microsecond-scale tasks, requiring techniques that can detect and react to overload at high frequency without incurring high overhead. In particular, overload control requires coordination between the server and the clients. Generating a coordination message consumes resources comparable to those of requests that take a few microseconds. Thus, the overload controller must require minimal coordination between the server and the client. In addition, the core allocation policy should minimize the queueing delay.

4. The need to handle a large number of clients. The above challenges are exacerbated by the fact that the server should be able to serve hundreds or even thousands of clients. A large number of clients leads to incast scenarios, making it harder to differentiate between persistent overload and momentary spikes in demand. Moreover, the overhead of admission control coordination is typically proportional to the number of clients. Thus, we need a protocol whose overhead does not grow significantly as a function of the number of clients.

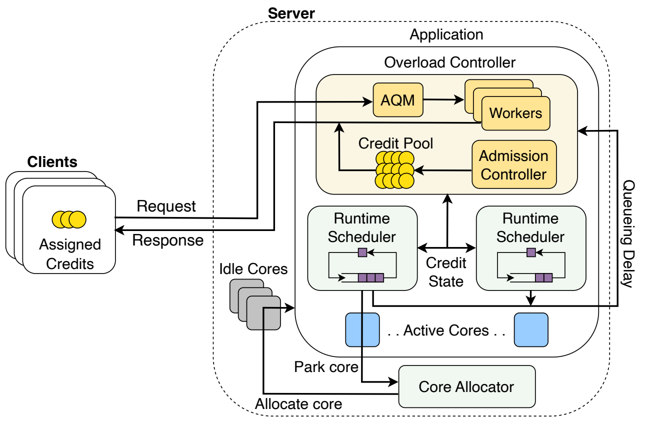

Fig. 2: CoreSync architecture

III. The CoreSync Protocol

A. Overview

We present CoreSync, a server-driven credit-based protocol that jointly decides the number of credits to issue to clients, as well as the number of cores to allocate to serve those clients. Figure 2 illustrates an overview of the operations of a CoreSync server and one of its clients. An RPC client initializes an RPC session by sending a message indicating its demand in terms of number of requests currently in its queue. The session initialization message can also include a request, to reduce the delay of request processing at session initialization. The server employs AQM to determine whether to process or drop incoming requests. If a request is processed, a response is sent to the client, including the number of credits for future requests. If a request is dropped, a failure message is sent to the client with the number of credits set to zero. A single credit allows a client to send one request to the server. The server will attempt to piggyback credit information on responses to the clients. However, if a session is inactive (i.e., with no requests at the server), the server will generate explicit messages indicating the allocation or revocation of credits.

The basic idea behind CoreSync is to judiciously maintain proportionality between the number of credits issued and the number of cores allocated. Proportionality is desirable, as it ensures that core allocation decisions take into account all admitted load to the system, currently being processed and in flight. Moreover, it limits the maximum burst size at low loads to a value proportional to the amount of allocated resources, improving tail latency at low loads. Naively enforcing proportionality is clearly suboptimal. To better understand its limitations, consider that credit-based admission controllers typically rely on credit overcommitment to ensure high throughput, especially at low load [9], [15], [16]. Thus, strict proportionality can severely degrade utilization by allocating more cores than necessary at low loads. Moreover, an erroneous decision to deallocate cores at high load can lead to significant degradation in throughput if strict proportionality is applied. Revoking credits from sessions that have no pending requests requires the exchange of explicit messages to inform the client of credit revocation, consuming CPU and harming utilization.

To avoid the downsides of strict proportionality, we design CoreSync to offer partial proportionality. In particular, when load is below capacity, a fraction of the credits are strictly overcommitted while the remaining credits are allocated proportional to the number of allocated cores. When the server is overloaded, proportionality is not maintained at all, with all cores allocated and none permitted to be deallocated, even as admitted load exhibits sawtooth-like behavior. CoreSync requires coordination between the core allocation logic in the operating system and the overload control logic in the RPC layer. Coordination is done by exposing an API that allows the operating system to query and atomically modify the state of the overload controller. Our design focuses on tasks that run to completion. We leave designing policies for preemptable workloads to future work.

There are two key design choices in CoreSync: 1) detecting when the server is persistently overloaded to disable core deallocation, and 2) designing a strategy for partial proportionality the balances throughput, latency, and utilization. In the following sections, we examine each of the design choices.

B. Detecting Persistent Overload

Our objective is to develop a lightweight signal that can differentiate between persistent overload and transient spikes in demand. The signal is necessary to enable CoreSync to prevent core deallocation when the server is facing persistent overload. There are two standard signals for detecting overload: queueing delay and CPU utilization. However, both signals are not suitable for detecting persistent high load. First, consider queueing delay or any other queueing statistic (e.g., queue length). Even at high load, a good overload controller should keep queues short (e.g., around or below some target value). The target queueing delay value is typically picked so that the latency under low load is not much worse than at high load. Moreover, in our setting, resources are dynamically allocated. Thus, it is typical to observe high tail latency at low load when a large spike of load arrives and requires additional CPU cores to be allocated. CPU utilization faces similar issues. An overload controller can create temporary drops in utilization during its congestion avoidance phase. Thus, the average utilization of a server utilization will be very close to 100% but not exactly at 100%. Such behavior conflates the scenario where the system is facing persistent overload and when the system is receiving high load and can afford to park at least one core.

To detect persistent overload, we instead observe the long-term behavior of the overload controller itself (i.e., over multiple RTTs). In particular, when an overload controller experiences persistent overload, the number will be reduced, eliminating any overcommitment. Moreover, it will fluctuate, following sawtooth-like behavior. Thus, reacting to a single multiplicative decrease event might not be an appropriate signal because it can be caused by a single spike in demand. Instead, we observe credit allocation across sessions. Reduction due to a transient spike will leave most clients with several credits. However, when the server is overloaded, credits will have to be reduced to a level where some active clients are issued zero credits, especially in the presence of a large number of clients (i.e., a hundred or more). Note that such behavior can occur with smaller connection counts as well. It will just require higher average load by individual connections. In other words, the system will likely be considered overloaded at higher offered loads.

There are many possible approaches to implement the tracking of per-session credit assignment. We opt for implementing a light-weight signal to facilitate exchanging state information between the RPC library and the operating system. In particular, we use the number of sessions with zero credits to indicate persistent overload. The RPC library tracks the total number of sessions that are issued zero credits, called drained sessions. The operating system atomically accesses the value to detect persistent overload. Such an overload detection signal can lead to different system behaviors as the number of clients varies. With a very large number of clients, it is highly likely that at least one will have zero credits, indicating persistent overload. Thus, the system will not park any cores, favoring throughput over utilization. The inverse is also true. With a small number of clients, the likelihood that one of them has no credits is low, except when the system is highly overloaded. Thus, the system in such cases will favor utilization over throughput. We examine the sensitivity of CoreSync to the number of clients in Section IV-D and observe this behavior.

C. Proportional Credit and Core Allocation

Basic overload control operations. CoreSync's basic credit management operations are constrained by the proportionality between admitted load and allocated cores. We draw inspiration from Breakwater, a state-of-the-art overload controller. CoreSync relies on a delay-based AIMD algorithm to control the number of credits it generates, $Credits_{total}$. In particular, if the measured delay of the oldest request in the system, D_m, exceeds a configurable target delay, D_t, the total number of credits is multiplicatively decreased proportional to the delay.

$$Credits_{total} \leftarrow Credits_{total} \ . \ \max(1 - \beta \frac{D_m - D_t}{D_t}, 0.5)$$

When delay is below D_t, credits are additively increased.

$$Credits_{total} \leftarrow Credits_{total} + \alpha$$

Algorithm 1 CoreSync Runtime Scheduling Algorithm

```
 1: $Credits_{total}$: Total credits available with the server
 2: $Credits_{issued}$: Credits distributed to the clients
 3: $num\_sess$ : Number of clients
 4: $num\_drained\_sess$ : Number of clients with zero credits
 5: $num\_cores$: Cores allocated to the application
 6:
 7: procedure UTHREADSCHEDULE
 8:     $start \leftarrow time()$
 9:     $C_{sub} = 0$
10:     $has\_started\_parking = False$
11:     if uthread(s) in local or remote runqueue then
12:         steal, if required, and jump to uthread context
13:     if $time() - start < POLL\_PERIOD$ then
14:         // Search for work again
15:         go to step 11
16:     if $num\_drained\_sess > 0$ then
17:         // Server overloaded, do not park
18:         go to step 11
19:     if another core is parking then
20:         // Park only one core at a time
21:         go to step 11
22:     // Server is not overloaded and only one core is parking
23:     if $Credits_{issued} - num\_sess > R \times num\_cores$ then
24:         if not $has\_started\_parking$ then
25:             // Signal overload controller to reduce issued credits
26:             $C_{sub} = (Credits_{issued} - num\_sess)/num\_cores$
27:             $Credits_{total} -= C_{sub}$
28:             $has\_started\_parking = True$
29:         // Wait for the issued credits to be revoked
30:         go to step 11
31:     else
32:         park the core
33:     // On wakeup, return the credits
34:     $Credits_{total} += C_{sub}$
35:     // Search for work again
36:     go to step 9
37: end procedure
```

AQM decisions are also made based on the value of D_m. In particular, for every incoming request, if D_m exceeds a configurable threshold D_{drop}, the request is dropped and a failure message is sent to its client. Otherwise, the request is processed. CoreSync atomically modifies $Credits_{total}$ without interfering with the above logic.

Basic core scheduling operations. We implement a simple delay-based reactive policy, similar to the Shenango policy, that allocates additional cores when delay exceeds a configurable threshold. Work stealing is used. When a core does not find work by polling and stealing, it is deallocated. Deallocation logic is dictated by CoreSync.

Enforcing partial proportionality. CoreSync allocates credits proportional to the number of allocated cores, while maintaining a portion of overcommitted credits to ensure high throughput. Following the above basic operations of credit generation, when the load is below capacity, the number of issued credits can grow as an additive increase is applied repeatedly, leading to overcommitment. CoreSync distributes credits to all sessions to maximize utilization, ensuring that any session that can generate load has at least one credit. The total number of credits generated by CoreSync, $Credits_{total}$, reflects - credits used by requests currently being processed at the server, requests in flight to the server, and credits issued to clients for future demand (i.e., overcommitted credits). Core schedulers attempt to park cores when demand at the server drops. Thus, CoreSync enforces proportionality between allocated cores and requests that might arrive in the next RTT.

To estimate incoming load, CoreSync tracks the number of issued credits, $Credits_{issued}$, indicating the total number of credits communicated to clients but are not currently used by requests at the server. The number of issued credits is made up of two parts $Credits_{issued} = Credits_{OC} + Credits_{prop}$, where $Credits_{prop}$ are proportional credits and $Credits_{OC}$ are overcommitted credits. Proportional credits are computed on based on the number of allocated cores N_{cores}: $Credits_{prop} = R \times N_{cores}$, where R is the proportionality parameter of CoreSync. The overcommitted credits are computed such that each session gets at least one credit. Our rationale is that when the system is receiving a load below its capacity, each client should be able to submit requests to the server.

CoreSync proportionality logic can read $Credits_{issued}$, computed by the RPC library but does not directly change its value. Instead, the value of $Credits_{issued}$ is indirectly changed by modifying the value of $Credits_{total}$. Updates to $Credits_{total}$ take at least a network RTT to impact $Credits_{issued}$ as credit information are communicated to clients. Algorithm 1 shows the procedure for proportional allocation of cores and credits. The procedure is triggered when a core finishes its current work. First, it attempts to steal work for a $POLL\_PERIOD$. Then, it can only park if there are no sessions with zero credits (i.e., $num\_drained\_sess = 0$) and a proportional number of credits have been revoked from clients. In particular, $Credits_{prop}/N_{cores}$ credits are subtracted from the total number of available credits. The core will busy poll until the value of $Credits_{issued}$ reaches its target value. This busy polling improves tail latency by allowing a core to only park when the maximum possible burst size has been reduced.

Intuitively, setting a lower value for R causes the partial proportionality to be violated more frequently during core deallocations. This decreases $Credits_{issued}$, limiting the potential for large request bursts and thereby improving tail latency. However, because the deallocating core must busy poll until the reduction in $Credits_{issued}$ takes effect, a lower value for R harms utilization. Conversely, setting a higher value for R results in fewer violations of the partial proportionality during deallocations. This allows the core to be parked immediately, improving utilization. Yet, since $Credits_{issued}$ are not reduced proportionally in such cases, request bursts may harm tail latency. The parameter R thus provides a knob for service operators to balance latency and utilization according to their objectives. We examine the sensitivity of CoreSync to the value of R in Section §IV-D.

CoreSync prioritizes the exchange of credit information by piggybacking it on responses, instead of explicit messages. Thus, credit revocation starts at active clients, before explicit messages are sent to inactive clients. This behavior ensures that when a client is issued a credit, it is not revoked until the system is experiencing persistent load beyond its capacity. Thus, setting $Credits_{OC}$ to be equal to the number clients ensures that each client receives at least one credit. On the other hand, configuring the parameter R can increase the level of overcommitment of the system. Indeed, we find that as the number of clients increase, R should increase to ensure that some clients receive more than one overcommitted credit, ensuring high throughput. In our experiments, we configure the parameter R statically according to the fixed number of clients in the test. However, the system could monitor the number of active clients and adjust R dynamically, linearly scaling its initial value in proportion to the active session count. We demonstrate that such linear scaling yields optimal performance in Section §IV-D.

D. Implementation

We implement CoreSync as part of the Caladan LibOS [2]. Overload control logic is implemented in a custom RPC library built, extending the Breakwater [9] and Protego [10] APIs. Core scheduling logic is divided between Caladan's monolithic scheduler (IOKernel) and the per-application runtime. Applications are modified to use the CoreSync RPC library. All applications are implemented following a dispatcher threading model: upon receiving a request, a new Caladan thread is spawned to process it and destroyed after completion. Separate asynchronous sender threads handle responses. This design enables accurate queueing delay measurements by simply tracking the queueing delay of dispatched threads.

The RPC layer exposes a single read/write variable: the total number of credits. It also exposes three read-only variables: the total number of active sessions, the number of drained sessions, and the number of issued credits. Each application is linked with a separate RPC library and Caladan LibOS instance, ensuring complete isolation of credit state of one application from other application's scheduler.

We use the implementation of the Shenango scheduler, augmenting its runtime component to implement Algorithm 1. In particular, we modify the logic invoked when a core fails to find work. In total, CoreSync adds ∼40 lines of code (LOC), with runtime changes in C and RPC library updates in C++.

IV. Evaluation

A. Evaluation Setup

Testbed. We conduct our experiments on eleven xl170 nodes on Cloudlab [24]. Each node is equipped with a 10-core (20 logical cores) Intel E5-2640v4 2.4GHz processor, 64 GB ECC RAM, and a Mellanox ConnectX-4 25 GbE NIC. The nodes are interconnected via a Mellanox 2410 switch. The average and 99th percentile round-trip latencies between any two nodes are 10 μs and 20 μs, respectively. All nodes run Ubuntu 18.04 with Linux kernel version 4.15. All applications are implemented to run on the Caladan [2] system and are linked with its runtime. We dedicate one node as the RPC server and use the remaining ten as RPC clients. The server utilizes up to 16 logical cores to run the target latency-sensitive application.

Workload. We evaluate CoreSync using one real-world and four synthetic applications. For the synthetic case, we use workloads with exponential and bimodal service time distributions. For each distribution, we run the tests for average service times of $1\mu s$ and $10\mu s$. For the real-world application, we use Memcached, a latency-sensitive key-value store. Further, we evaluate the system under different load intensities between 0.1× and 1.5× the system's capacity. When load is below capacity, we report results for 0.2× capacity (low), 0.5× capacity (medium), and 0.8× capacity (high) for ease of exposition. When load is near or above capacity, we report results for 1, 1.3, and 1.5× capacity. When all cores are allocated, the maximum capacity of the system is around 1.2 million requests per second and 4 million requests per second for 10 μs and 1 μs average service times respectively. The system's capacity for Memcached is around 3.5 million requests per second. Load is generated equally by all client machines. Each client machine simulates a large number of clients by creating more sessions. Our default is ten sessions per machine. Requests are generated following an open-loop Poisson arrival process. Our default workload is the exponentially distributed synthetic workload with $10\mu s$ average service time.

Baselines. We compare CoreSync's performance against multiple state-of-the-art core scheduling policies. This includes two reactive policies - Shenango [1] and Caladan [2], and two proactive policies - Utilization and Delay range [3]. We use Breakwater [9] as the overload controller for all core scheduling policies.[1]

Metrics. We focus on three metrics: the throughput and latency of a latency critical application and the overall CPU utilization. When load is below capacity, we report the 99th percentile (tail) latency of the latency-critical application and the application's CPU utilization. Latency is measured for every request at the client: it includes the time required to traverse the network, wait in a queue at the server, and execute at the server. CPU utilization is the percentage of total (20) cores available on the server. We also measure CPU utilization as a function of the throughput of a best-effort synthetic workload. When load is near or beyond capacity, we report the throughput of the latency-critical application. Throughput is the number of requests per second whose responses were received successfully by clients. As discussed earlier, throughput is almost identical for all policies when load is below capacity (i.e., within 1% differences). On the other hand, tail latency is similar for all policies when load is near or beyond capacity (i.e., within 3% differences).

Parameter Configurations. We have three separate sets of configurations: the core allocation policy, the overload controller (i.e., Breakwater), and CoreSync.

Core Allocation System Parameters. We use the recommended parameters for each of the studied core allocation

[1]Note that Breakwater was originally evaluated with the all cores statically allocated to a single application [9].

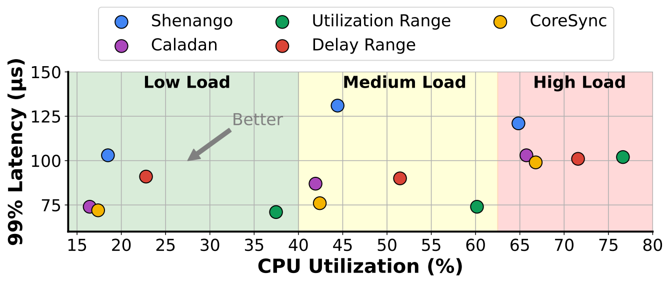

Fig. 3: Latency v/s Utilization, below capacity, for exponentially distributed workload with an average of 10 μs.

Policy	Normalized Throughput					
	Exponential			Bimodal		
	100% Load	**130% Load**	**150% Load**	**100% Load**	**130% Load**	**150% Load**
CoreSync	1	1	1	1	1	1
Shenango	0.942	0.974	0.973	0.939	0.966	0.975
Caladan	1.029	1.007	0.998	0.991	0.997	1.002
Util. Range	1.003	0.988	0.976	0.974	0.976	0.979
Delay Range	0.979	0.968	0.977	0.962	0.970	0.976

TABLE I: Throughput, near and beyond capacity, for synthetic workloads with an average of 10 μs.

policies. For all policies we use a queueing delay threshold of 5μs and allocation interval of 5μs. For Shenango and Caladan, the polling period, before parking a core, is set to 2μs. Further, for Utilization range, we use the CPU utilization range of 75-95%, and for Delay range, we use the delay range of 0.5-1μs.

Breakwater Parameters. We use the recommended Breakwater parameters [9]. α is set to 0.1%. β is set to 2%. The target delay is set to 80μs for requests with 10μs average service times, 45μs for those with 1 μs average service times, and 25μs for Memcached. In all cases, the AQM drop threshold is set to twice the target delay.

CoreSync Parameters. CoreSync's basic overload control and core allocation operations resemble Breakwater and Shenango. Thus, we use similar configurations for them. The CoreSync-specific parameter is the proportionality parameter R. We set R to 50 for synthetic workloads and 20 for Memcached. Section §IV-D examines the impact of varying R on CoreSync's performance.

B. Overall Performance for Synthetic Workloads

Performance when load is below capacity. The overload controller admits all load, leading all policies to achieve comparable throughput. However, they differ significantly in the way they balance utilization and latency. Figure 3 shows their achieved performance for the synthetic workload with exponentially distributed service time and a 10 μs average. Reactive policies like Shenango and Caladan quickly allocate cores in response to bursts but also deallocate aggressively during traffic dips, unaware of potential upcoming bursts. Thus, we observe that they typically achieve good utilization at the expense of latency. In particular, CoreSync improves tail latency by 1.7× compared to Shenango. Caladan remains on the Pareto frontier offering slightly better utilization (a reduction of 2%) at the expense of latency (an increase of 16%), as compared to CoreSync.

Proactive policies are less sensitive to bursts created by overcommitment, maintaining the same number of allocated cores for a given workload. However, both proactive policies strictly prioritize latency over utilization. Moreover, the Delay Range policy is never on the Pareto frontier. At medium loads, CoreSync improves CPU usage by 1.4× and 1.2× over Utilization Range and Delay Range policies, respectively, while maintaining similar tail latency to Utilization Range. Moreover, it improves tail latency compared to Delay Range by 3-21%. Overall, CoreSync is not only on the Pareto frontier when load is below capacity, it also provides the best balance between utilization and latency.

Performance when load is near and beyond capacity. When the server is overloaded, all policies achieve comparable tail latency because overload control behavior is similar across all policies. However, they differ primarily in achieved throughput (Table I). In particular, the overload controller forces the admitted load to exhibit sawtooth-like behavior. Existing policies are sensitive to momentary drops in load. Thus, all policies, except Caladan, exhibit throughput degradation. Caladan is more conservative in deallocation, taking into account the state of other system bottlenecks, leading to fewer allocation changes and the highest throughput under overload among all policies. CoreSync, in contrast, avoids deallocating cores during persistent overload. It uses the number of client sessions with zero credits as a stable signal to suppress deallocation and maintain throughput. CoreSync outperforms Shenango, Utilization Range, and Delay Range policies by up to 5.8%, 3%, and 4%, respectively. Moreover, Caladan outperforms CoreSync by only 2.9% in its best case.

Impact of different service time distributions. Figure 4 shows the performance, below capacity, for a workload whose service time follows a bimodal distribution with 10μs average. Bimodal workloads are more bursty in terms of their CPU requirements, creating challenges for existing policies. In particular, we find that CoreSync remains on the Pareto frontier, with the gap between it and other policies growing, in terms of both latency and utilization. For example, at medium loads, CoreSync reduces latency compared to reactive policies like Shenango and Caladan by 50% and 10%, respectively. Compared to proactive policies, CoreSync improves utilization by 1.3× and 1.2× compared to Utilization range and Delay range, respectively. When load exceeds capacity, CoreSync outperforms or exactly matches all existing policies in terms of throughput. The results are shown in Table I. CoreSync improves performance by up to 6% compared to Shenango, 2.6% compared to Utilization Range, and 3.8% compared to Delay Range. The performance of Caladan and CoreSync is very similar (within less than half a percent).

Impact of different average service times. We now consider CoreSync's performance for shorter requests. For short requests, we find that no policy has a clear advantage when load is beyond capacity. However, CoreSync maintains its advantage when load is below capacity. Figure 5 shows performance below capacity for workloads with exponential

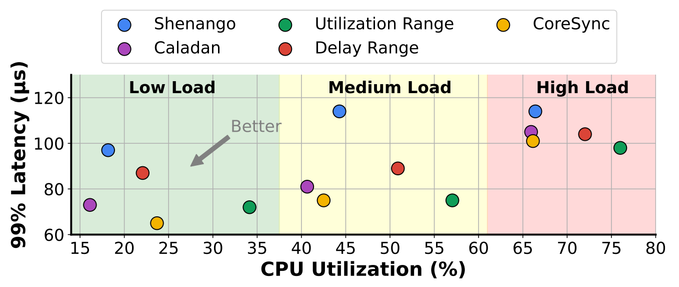

Fig. 4: Latency v/s Utilization, below capacity, for bimodally distributed workload with an average of 10 μs.

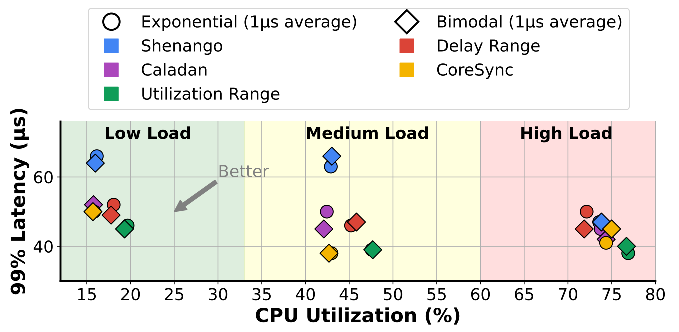

Fig. 5: Latency v/s Utilization for exponential and bimodal service time distributions with an average of 1 μs.

and bimodal service time distributions, both averaging $1\mu s$. The trends observed for the $10\mu s$ workload largely hold here as well. Reactive policies continue to aggressively adjust core allocations in response to load changes, achieving better utilization but worse tail latency. Proactive policies, influenced by frequent bursts, tend to allocate more cores as average metrics approach the upper end of the configured range. However, the excess allocation compared to reactive schemes is less pronounced than in the $10\mu s$ case.

CoreSync uses the credit state from the overload controller to guide core deallocations, keeping operation near the bottom-left of the latency–utilization curve across loads. At medium load, for the exponential case, CoreSync achieves up to 1.6× and 1.3× lower latency than Shenango and Caladan, while providing 10% and 5% better utilization than the Utilization and Delay range policies, respectively. For the bimodal case, CoreSync delivers up to 1.7× and 1.2× lower latency than Shenango and Caladan, and 10% and 7% better utilization than the Utilization and Delay range policies, respectively.

C. Overall Performance for Real-World Workloads

We evaluate CoreSync's effectiveness on real-world low-latency workloads using Memcached with the USR workload from [25], where 99.8% of requests are GET and 0.2% are SET. Each GET has a latency under $1\mu s$. Memcached's request service time is almost constantly distributed. Memcached is therefore expected to behave a lot like the $1\mu s$ synthetic workload we saw in Figure 5. To assess CPU efficiency, we co-locate Memcached with a background Best-Effort (BE) application— a CPU-bound task performing square root calculations. The BE task opportunistically uses idle CPU cycles. Without any Memcached load, it utilizes all 16 logical cores on

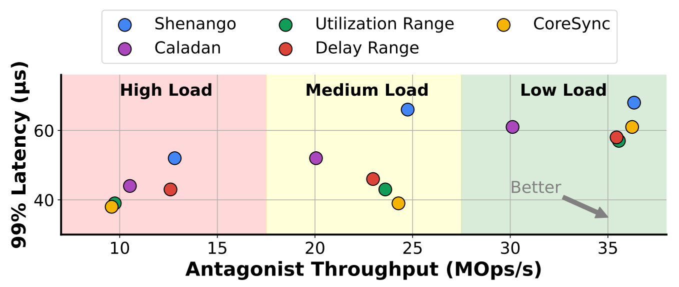

Fig. 6: Latency v/s Antagonist Throughput, below capacity, for Memcached.

Policy	Normalized Throughput		
	100% Load	130% Load	150% Load
CoreSync	1	1	1
Shenango	1.019	0.997	0.999
Caladan	1.002	1.001	1.004
Utilization Range	1.006	0.986	0.995
Delay Range	0.986	0.965	0.984

TABLE II: Throughput results, near and beyond capacity, for Memcached

the server. An ideal core allocation policy should maintain low Memcached latency while maximizing BE throughput (i.e., square roots per second).

When load is below capacity, latency and CPU utilization results for Memcached closely mirror those observed in the $1\mu s$ synthetic case. Note that we use the throughput of the BE application as a proxy for CPU utilization. Results are shown in Figure 6. An important observation is that improved CPU utilization of the latency-critical application, does not necessarily translate to improved throughput for the BE application. In particular, Caladan exhibits low BE throughput when Memcached has low and medium load. This is probably due to its Hyperthread and Memory Bandwidth Controllers, which can block cores from being allocated to a BE application to reduce interference. CoreSync remains on the Pareto frontier, achieving up to 1.6× lower latency and 1.2× higher BE throughput compared to other policies. Notably, at high loads, CoreSync favors latency over BE throughput - driven by the configured value of parameter R. Increasing R can improve efficiency (BE throughput) at the cost of latency, as we will see in Section IV-D. Under overload, CoreSync improves throughput by up to 3% over Shenango, Utilization Range, and Delay Range policies, while remaining within 0.4% of Caladan's throughput (Table II).

D. Sensitivity to Different Parameters and Settings

Effect of varying the proportionality parameter, *R*. We vary the value of R from 10 to 100, and observe how the overall performance as we change load on the server. We use a workload having exponentially distributed service times with a mean of $10\mu s$. Figure 7 shows that for lower values of R, CoreSync favors optimizing latency over utilization. This is because during core deallocation, credits will be reduced more frequently, for a lower value of R, to maintain partial proportionality. Reducing the number of issued credits reduces the maximum possible burst size, improving latency.

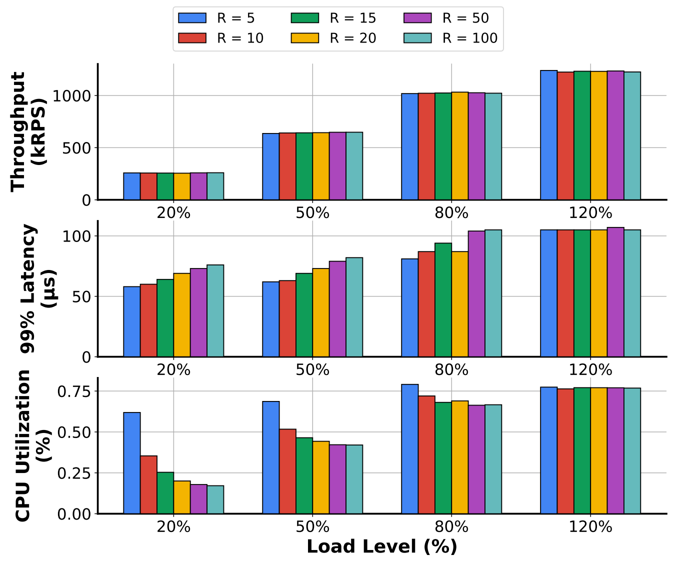

Fig. 7: Throughput, Latency, and Utilization of CoreSync, for varying values of the parameter R.

Moreover, a core will busy poll for longer as it awaits more credits to be revoked, harming utilization. Increasing the value of R has the opposite effect, allowing it to behave more strictly according to rules of CoreSync's basic core allocation logic (i.e., Shenango in our design).

Scalability to a large number of clients. We vary the number of client sessions from 100 to 10000 to evaluate how well CoreSync's performance scales with a large number of clients. As discussed in Section III, the parameter R should also scale with the number of clients in the system, hence, we set $R = 50$, $R = 500$, and $R = 5000$ for 100, 1000, 10000 sessions experiments, respectively. Figure 8 shows performance when load is below capacity, demonstrating that the benefits of CoreSync are sustained regardless of the number of sessions. We compare CoreSync with Shenango - a reactive policy and Utilization range - a proactive policy. Similar to our earlier findings, CoreSync provides 1.5× lower latency than Shenango and 40% better CPU utilization than Utilization range, across all connection counts. Throughput improves at high connection counts above capacity because there's a higher chance of at least one drained session, which delays core deallocation.

V. Discussion

Basic overload control and core scheduling operations in CoreSync. We choose to base the basic overload control and core scheduling operations of CoreSync on Breakwater and Shenango, respectively. While we demonstrate that this design choice of CoreSync outperforms existing techniques, the advantages of CoreSync stem from the proportionality between admitted load and the number of cores allocated to serve it. Thus, we believe that other systems should be able to employ different basic overload control and core allocation strategies, depending on the constraints of their deployments.

Practical relevance of CoreSync. Overload controllers are developed for scenarios where load is expected to exceed capacity. Fast core schedulers are developed for scenarios where load is expected to change rapidly below the capacity of the server. At first glance, it might seem that they are built for very different settings, and that might have been historically true. However, recent trends push the operations of servers towards true elasticity, motivated by FaaS applications [5], [26]. Such advancements necessarily enable scenarios where load quickly shifts from below the capacity of a server to beyond it. Such scenarios would require CoreSync-like systems.

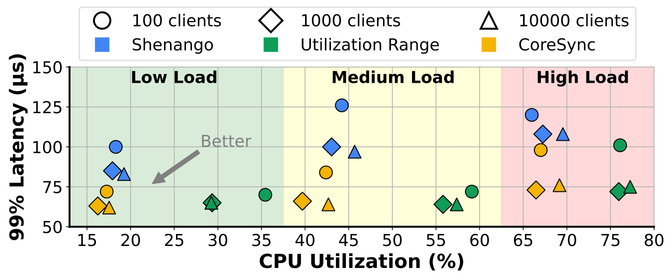

Fig. 8: Latency vs. Utilization under exponential service times (mean 10 μs) across connection counts. CoreSync uses $R =$ 50, 500, and 5000 for 100, 1k, and 10k clients, respectively.

Limitations. CoreSync requires a server-driven admission controller. It is unclear if joint core scheduling and admission control can be performed accurately when client-based admission controllers are used (e.g., [12], [13], [27]). Moreover, CoreSync assumes that operating systems can access and modify the state of the RPC-layer, limiting its practicality to scenarios where an operator controls the whole stack. However, CoreSync might be offered as a service akin to autoscaling. Finally, CoreSync was designed for run-to-completion tasks. Supporting more sophisticated schedulers would require further investigation beyond the scope of this paper.

VI. Conclusion

This paper presents CoreSync, a policy for optimizing the combined decision-making of dynamic core allocation and overload control. CoreSync is a first step in tackling the downsides of the growing complexity of resource management in large-scale servers. It demonstrates the value of having an integrated view of the behavior of modern servers. Such a view led to the design of a simple-yet-effective algorithm that maintains partial proportionality between admitted load and the amount of resources allocated to serve it. We show that CoreSync lies on the Pareto frontier of utilization, latency, and throughput, dominating some state-of-the-art solutions. In particular, CoreSync can improve throughput by up to 6% at high loads while reducing latency by 58% and improving utilization by 40% at low loads.

Acknowledgment

We thank our shepherd and the anonymous reviewers. We also thank Cloudlab [24] for providing us with the infrastructure to run our experiments. This work was funded in part by NSF grants CNS-2212098 and CNS-2212103 and a Google Research Award.

References

[1] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 361–378.

[2] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 281–297.

[3] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy, "Efficient scheduling policies for Microsecond-Scale tasks," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, 2022, pp. 1–18.

[4] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-Aware thread management," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 145–160.

[5] J. Fried, G. I. Chaudhry, E. Saurez, E. Choukse, I. Goiri, S. Elnikety, R. Fonseca, and A. Belay, "Making kernel bypass practical for the cloud with junction," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 55–73. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/fried

[6] J. Lin, Y. Chen, S. Gao, and Y. Lu, "Fast core scheduling with userspace process abstraction," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*, 2024, pp. 280–295.

[7] Y. Li, N. Lazarev, D. Koufaty, T. Yin, A. Anderson, Z. Zhang, G. E. Suh, K. Kaffes, and C. Delimitrou, "Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 922–936.

[8] J. Lin, A. Cardoza, T. Khan, Y. Ro, B. E. Stephens, H. Wassel, and A. Akella, "{RingLeader}: efficiently offloading {Intra-Server} orchestration to {NICs}," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023, pp. 1293–1308.

[9] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay, "Overload control for µs-scale RPCs with breakwater," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020, pp. 299–314.

[10] I. Cho, A. Saeed, S. J. Park, M. Alizadeh, and A. Belay, "Protego: Overload control for applications with unpredictable lock contention," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 725–738. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/cho-inho

[11] H. Zhou, M. Chen, Q. Lin, Y. Wang, X. She, S. Liu, R. Gu, , B. C. Ooi, and J. Yang, "Scalable overload control for large-scale microservice architecture," in *SoCC*, 2018.

[12] M. Welsh and D. Culler, "Overload management as a fundamental service design primitive," in *SIGOPS European Workshop*, 07 2002.

[13] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: A microkernel approach to host networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, 2019, p. 399–413.

[14] R. Bhattacharya, Y. Gao, and T. Wood, "Dynamically balancing load with overload control for microservices," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 19, no. 4, pp. 1–23, 2024.

[15] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: a receiver-driven low-latency transport protocol using network priorities," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 221–235. [Online]. Available: https://doi.org/10.1145/3230543.3230564

[16] K. Prasopoulos, R. Kosta, E. Bugnion, and M. Kogias, "SIRD: A Sender-Informed, Receiver-Driven datacenter transport protocol," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. Philadelphia, PA: USENIX Association, Apr. 2025, pp. 451–471. [Online]. Available: https://www.usenix.org/conference/nsdi25/presentation/prasopoulos

[17] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "ghost: Fast & flexible user-space delegation of linux scheduling," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, 2021, pp. 588–604.

[18] K. Yasukata and K. Ishiguro, "Developing process scheduling policies in user space with common os features," in *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2024, pp. 38–44.

[19] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar *et al.*, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*, 2021, pp. 195–211.

[20] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[21] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion, "R2P2: Making RPCs first-class datacenter citizens," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 863–880.

[22] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis, "The nebula rpc-optimized architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 199–212.

[23] J. Hussien, P. Sahu, E. Stuhr, and A. Saeed, "Modeling the Interactions between Core Allocation and Overload Control in μs-Scale Network Stacks," in *2025 34nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2025.

[24] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/duplyakin

[25] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.

[26] T. Kuchler, P. Li, Y. Zhang, L. Cvetković, B. Goranov, T. Stocker, L. Thomm, S. Kalbermatter, T. Notter, A. Lattuada *et al.*, "Unlocking true elasticity for the cloud-native era with dandelion," *arXiv preprint arXiv:2505.01603*, 2025.

[27] B. Wydrowski, R. Kleinberg, S. M. Rumble, and A. Archer, "Load is not what you should balance: Introducing prequal," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024, pp. 1285–1299.