# Poster: Understanding Interactions between Overload Control and Core Allocation in Low-Latency Network Stacks

Eric Stuhr
Georgia Tech

Ahmed Saeed
Georgia Tech

## ABSTRACT

Modern data center applications require servers to respond to requests from thousands of clients while maintaining micro-second-scale SLOs. Efficient operation of the data center infrastructure requires assigning the exact amount of resources needed by the application, no more and no less. A burst of incoming traffic that exceeds allocated capacity can cause long queues, requiring efficient and responsive overload control schemes. A sudden drop in demand requires reallocation of resources to other applications. Within a single host, several mechanisms have been proposed for overload control [1, 3, 8, 9] and dynamic core allocation [2, 4, 6, 7]. The state of the art in both control loops is designed to react to microsecond-level changes in load. We present a simulation-based study of the interaction between Overload Controllers and Core Allocators at microsecond timescales, examining their macroscopic implications at larger timescales.

Core Allocators and Overload Controllers strive for the same delicate balance between high utilization and low latency, employing complementary approaches. In particular, Core Allocators adjust the provisioned capacity to an application, assuming a fixed load. Overload Controllers adjust the admitted load, assuming fixed capacity. Moreover, both control loops typically make their decision based on the amount of queueing in the system, adding more resources or admitting less load as queueing delay increases, targeting a specific level of queueing delay. The potential for interference between the two control loops arises because the quantity that one controller assumes to be fixed is changed by the other controller (i.e., load and capacity). Further, both controllers use the same signal to make their decision.

## CCS CONCEPTS

• **Networks** → **Data center networks**; **Network control algorithms**; **Network resources allocation**;

**ACM Reference Format:**
Eric Stuhr and Ahmed Saeed. 2023. Poster: Understanding Interactions between Overload Control and Core Allocation in Low-Latency Network Stacks. In *ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3603269.3610844

## KEYWORDS

Low-Latency Network Stacks; Overload Control; Dynamic Core Allocation


*ACM SIGCOMM '23, September 10, 2023, New York, NY, USA*

ACM ISBN 979-8-4007-0236-5/23/09…$15.00
https://doi.org/10.1145/3603269.3610844

## 1 STUDY OVERVIEW

We study the interference between overload controllers and core allocators by examining the interaction between two exemplar controllers: Shenango [6] for core allocation and Breakwater [1] for overload control.

Shenango[6] is a library operating system that allows applications to maintain high utilization and low latency by dynamically reallocating cores at high frequency – every 5$\mu$s. The core allocation decision is based on the amount of queueing in the system, observing thread queues and packet queues. An application is allocated an additional core if its queueing exceeds a preconfigured threshold. Shenango employs work stealing to balance load between cores. Upon failing to find work, a core parks (i.e., gets deallocated from the application).

Breakwater[1] is a credit based overload controller, only allowing clients to issue requests when they have credits. It adjusts a credit pool size using an additive increase multiplicative decrease (AIMD) rule every round trip time (RTT) by observing the maximum queueing delay in the system, including thread queues and packet queues. Breakwater overcommits the number of credits it issues to ensure high utilization, relying on active queue management (AQM) to drop requests if the delay in the system exceeds a preconfigured threshold.

Our study is simulation-based, allowing us to examine a wide range of parameters, sidestepping limitations posed by implementation specifics (e.g., core allocation latency) or the environment (e.g., RTT). We leverage a simulator developed to study the behavior of core allocation schemes, including Shenango [5]. We augment it by adding support for overload control algorithms and an implementation of Breakwater. Our simulator is open source.[1] We acknowledge that further research in this area requires examining the behavior of the real implementation, but simulations provide a sanitized view of the behavior, allowing us to develop an intuition of what to expect of the full implementation as well as the conditions that are likely to produce problematic behavior.

Unless stated otherwise, the results were produced using a load that has an average service time of 1$\mu$s with a bimodal distribution, and Shenango and Breakwater have target delays of 10 µs. Breakwater's AIMD algorithm has an $\alpha$ of 1 and a $\beta$ of 0.08. The network RTT is 30$\mu$s. A server has 32 cores that Shenango can dynamically allocate to the application. There is a single application running. Load is normalized by the ideal throughput of the system with 32 cores fully utilized (i.e., a load of 1 refers to the ideal 32 million 1 $\mu$s tasks per second). The delay for allocating a core is 5$\mu$s.

## 2 PROBLEMATIC INTERACTIONS

**Slow reaction to load changes**. Consider a scenario where load sharply increases, from 0.2, requiring only 7 cores, to 1.2 requiring all 32 cores. At low load, the core allocator parks underutilized cores.

[1] https://github.com/estuhr1206/scheduling-policies-sim

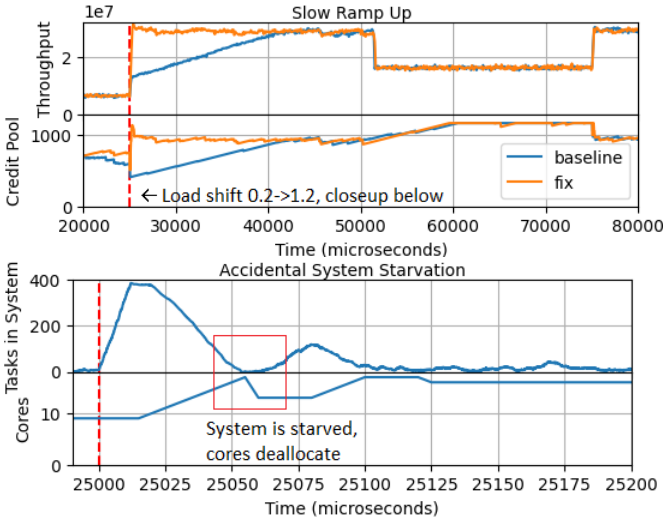


**Figure 1: Top plot: slow ramp up in credit pool size, leading to a slow growth in throughput in the baseline. Bottom is a closeup at the 25000 $\mu$s load shift (red line), with tasks in system (cumulative length of all queues) and number of allocated cores. Load shifts every 25,000 $\mu$s (0.2->1.2->0.5->1.2).**

Thus, the system can still have high latency because the available capacity is reduced to satisfy the offered load.[2] For example, even though the system has 32 cores, only 7 are active. An incoming request that finds all 7 cores busy will have to wait for one of the cores to finish its work or for a new core to get allocated. Two problems occur due to this situation.

First, such occasional queueing delay can actually cause the overload controller to reduce its credit pool size to a level corresponding to the allocated capacity, not the total available capacity (i.e., the 7 cores not the 32 cores). Overload controllers are typically designed assuming fixed capacity. Assuming that in the steady state the credit pool size is near its ideal size, only small adjustments to the credit pool size are needed when high delays are observed, making overload controllers slow to react. Dynamic core allocation breaks the fixed capacity assumption. The slow reaction of the overload controller can lead to a very slow ramp up, leading to low utilization. The top figure in Figure 1 shows the issue as the credit pool takes time to reach the level needed to achieve high throughput after the change in load.

Second, when the load sharply increases, incoming requests face long queueing delay as they wait for more cores to get allocated. It's important to note that the core allocator needs to observe high latency in order to allocate more cores to an application. Further, once the allocation decision is made, core allocation is not instantaneous and can take a few microseconds. Thus, most of the burst ends up facing long queueing delay, triggering the AQM algorithm, leading to the dropping of a significant portion of the burst. It takes at least an RTT for clients to get notified of the dropped requests and issue more requests. During that RTT, the system utilization can be fairly low where cores don't find work and park, reducing capacity when load increases! The bottom figure in Figure 1 shows this problem where load increases from 0.2 to 1.2, creating a burst. Much of the burst gets dropped, leading cores to park.

[2]The excess capacity can be allocated to some background batch job that gets preempted if the application needs the capacity.

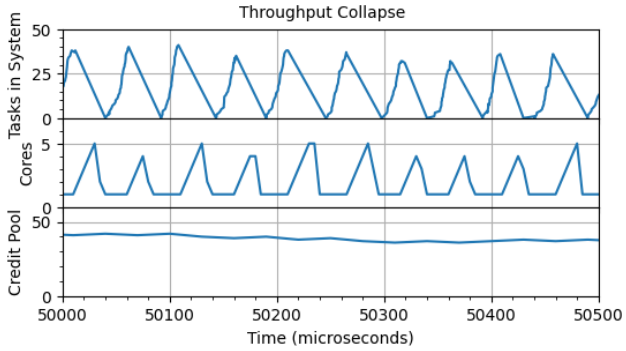


**Figure 2: An example of throughput collapse at load of 1. A credit pool size of 40 and an RTT of 30$\mu$s leave the system idle that only a single core remains active. Incoming bursts always observe high latency, preventing the credit pool size from growing.**

**Throughput Collapse.** This problem occurs when the credit pool size drops to a very low level (e.g., 40 in our example, where the minimum pool size allowed by Breakwater is 32). Such a small pool size can happen due to severe congestion or poor configuration. A small credit pool size leads to low load, as very little work is admitted to the server, leading the core allocator to park most cores.

Consider an incast scenario occurring under these conditions (i.e., a small credit pool size and most cores parked). Each client has a single credit and a backlog of requests. Incoming requests arrive in a burst and face long delays, awaiting cores to be allocated. The long delays prevent the overload controller from increasing its credit pool size. All requests are processed and responses are sent out. If the network RTT is large enough compared to request service time, the system is left idle for long enough that most cores to park. Thus, when a new burst of requests arrive, the same behavior repeats. The system gets stuck in this cycle. In our simulations, a network RTT of 25$\mu$s is enough for this problem to manifest. Figure 2 shows this cyclic behavior, where 1.0 load should use all 32 cores.

## 3 PROPOSED SOLUTION

We propose modifying overload controllers to take into account decisions made by the core allocator. We use an increase in the number of allocated cores as a signal for increased capacity, triggering the admission of more load. In particular, we modify Breakwater to increase its credit pool size when new cores are allocated. The credit pool ($C_{total}$) increases proportional to the added capacity.

$$C_{total} \leftarrow C_{total} + N_a \cdot \frac{C_{total}}{N_t}$$

where $N_a$ is the number of newly allocated cores and $N_t$ is the total number of allocated cores. Further, to avoid the throughput collapse problem, we set a minimum credit pool size that depends on the maximum number of available cores and the RTT, unlike in Breakwater where it depends only on the number of available cores. Figure 1 shows the behavior after employing the mitigation technique. The solution yields a 7% improvement in overall throughput while maintaining similar latency levels to the baseline.

## REFERENCES


[1] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for µs-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 299–314.

[2] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 281–297.

[3] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Mike Ryan, David J. Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*.

[4] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 399–413.

[5] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA, 1–18.

[6] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[7] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160.

[8] Matt Welsh and David Culler. 2002. Overload Management as a Fundamental Service Design Primitive. In *SIGOPS European Workshop*.

[9] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, , Beng Chin Ooi, and Junfeng Yang. 2018. Scalable Overload Control for Large-scale Microservice Architecture. In *SoCC*.