Modeling the Interactions between Core Allocation and Overload Control in μs-Scale Network Stacks

Jehad Hussien, Pratyush Sahu, Eric Stuhr, Ahmed Saeed
Georgia Institute of Technology

***Abstract*—Modern datacenter operators aim to maximize the utilization of limited and expensive resources, especially CPU cores. Achieving such an objective requires fast and accurate core scheduling policies. Meanwhile, operating at high utilization requires the employment of overload controllers that shed excess load beyond the allocated capacity. Currently, no analytical techniques exist to study the interactions between these controllers. In this paper, we use performance verification to establish bounds on the throughput and latency achieved by a server that employs state-of-the-art fine-grained core allocation and overload control mechanisms. Our model enables system performance analysis under a wide range of workload and system configurations (e.g., RTT, load, and burstiness). We show that worst-case throughput and latency degrade by 1.8$\times$ and 2.3$\times$, respectively, under high load and burstiness due to the interactions between overload control and fast core allocation. We validate our findings using simulations that demonstrate the plausibility of the identified worst-case behavior under realistic conditions.**

I. Introduction

Modern large-scale servers operate under strict performance requirements while maintaining high levels of utilization. Operators expect requests to finish within tight latency Service Level Objectives (SLO) while utilizing exactly the amount of resource they need, no more and no less. Operating at high utilization requires fast reaction to bursts in demand to ensure that requests meet their SLOs. To satisfy these requirements, many proposals have been developed to improve resource allocation performance within a single server, spanning dynamic core/thread allocation [1]–[4] and overload control [5]–[8]. The design, analysis, and deployment of core allocation and overload control schemes is done independently.

In this paper, we model the performance of servers that employ overload control and fast core allocation simultaneously, providing guarantees on their achieved throughput and latency. To model the interactions between two controllers, we leverage performance verification, a new technique used to establish bounds on the performance of resource allocation algorithms, while capturing realistic behavior [9]–[12]. In performance verification, an engineer models the behavior of their algorithm in first-order logic formulas of Boolean variables or linear arithmetic inequalities. Then, they formulate a hypothesis about the performance of the system (e.g., latency can exceed a predefined SLO). An automated solver is then employed to find a satisfying assignment for the model under the hypothesis. If such a satisfying assignment is found, it serves as a counterexample, demonstrating poor performance. Otherwise, the engineer obtains a proof that the performance property holds under all conditions consistent with the model.

We focus on the interactions between two state-of-the-art algorithms: the Shenango core allocation policy [1], and the Breakwater overload controller [5]. Both algorithms operate at microsecond-timescale, increasing the likelihood of interference between their decisions. To contextualize our analysis, we compare their combined behavior to scenarios where only overload control is employed and core allocation is fixed (i.e., a fixed number of cores is allocated to the application). Our model captures the throughput and latency of a server under different conditions: workload burstiness, load on the system relative to its capacity, network RTT, and system configuration.[1] Our results demonstrates that interactions between Shenango and Breakwater, under high load and burstiness, increase worst-case latency by 2.3$\times$, when only overload control is employed. Moreover, interactions between the two algorithms reduce throughput by 46%.

The intuition behind poor performance is that the core allocator can decide to deallocate many cores when load momentarily drops, even when the overload controller has already admitted a large number of requests. Then, a burst of requests arrives, facing high delays because only a small number of cores are available. High delays lead the overload controller to admit less load, lowering throughput. This cycle can repeat, leading to persistently high latency and low throughput. We validate our finding in simulation, reproducing the behavior captured by our model in a realistic simulation of the system. Moreover, insights obtained through this modeling effort led to the design of CoreSync [13], a protocol for combined core allocation and overload control. To summarize, this paper makes the following contributions:

- Develop a model of systems that employs fine-grained core allocation and overload control algorithms.
- Leverage the model to establish bounds on the throughput and latency of the modeled system, compared to a system that has all cores statically allocated.
- Validate the results of the model through simulation, demonstrating that the worst-case analysis can provide practical information.

II. Background and Problem Statement

A. *Core Allocation and Overload Control: An Overview*

Core allocation. High-frequency CPU core allocation schemes have recently received a lot of attention motivated by the need to meet the performance requirements of

[1]Model implementation and steps to reproduce our results are available at https://github.com/GT-ANSR-Lab/core-allocation_overload-control_model

microsecond-scale tasks while maintaining high utilization of large-scale servers [1]–[4]. High-frequency core allocation schemes are deployed in scenarios where a latency-critical application is deployed on a server along with a best-effort application. The objective of the core allocation algorithm is to allocate any unused CPU capacity by the latency-critical application to the best-effort application, without harming the performance of the latency-critical application. Core allocation decisions are made periodically, every few microseconds, based on load. If the load offered to the latency-critical application is high, more cores are allocated to it, and vice versa. Dynamic core allocation schemes are deployed with load balancing algorithms, to maximize the utilization of allocated cores. A recent study provides a comprehensive simulation-based analysis of state-of-the-art CPU scheduling policies for microsecond-scale RPCs [3]. Further, the paper provides a proof that optimally solving the core allocation problem is NP-Hard. Analysis of core allocation schemes is typically done using simple queueing models [1], [3], [14].

Overload control. Overload occurs when the load exceeds the available capacity. Overload can be triggered by incast scenarios or load imbalance between machines. In extreme scenarios, overload can lead to livelock where all incoming requests starve while waiting to be processed in long queues [15]. Overload controllers are employed to limit such queueing delays. Client-based overload controllers employ congestion windows whose size is determined based on an estimate of the load on the server [8], [16]. On the other hand, receiver-driven overload controllers allow the server to issue credits, with each credit representing permission for a client to send exactly one request to the server [5]. Overload controllers typically employ Active Queue Management (AQM) to drop requests that might experience large delays [5], [7], [17]. Most existing overload controllers rely on queueing delay to make their decision, admitting more requests when queueing delay is low, and vice versa. Overload control algorithms resemble congestion control algorithms, enabling the use of fluid models and control theory to study their behavior [18].

B. *Problem Statement*

Overload control and core allocation can be developed and deployed independently within the same system. One controls the allocated capacity given a certain load. The other controls the admitted load given a certain capacity. Thus, these algorithms can interfere with each other, especially because both controllers operate at comparable timescales when deployed to manage microsecond-scale RPCs [3], [5], [16]. Our objective is to develop a model that allows us to analytically study and establish guarantees on the performance of a server that employs overload control and core allocation simultaneously. Although our proposed model can be generalized, we focus on the Shenango core allocator and the Breakwater receiver-driven overload controller. We use performance verification to study their combined behavior and compare it to a system that only uses overload control, while keeping all cores statically allocated to the latency-critical application.

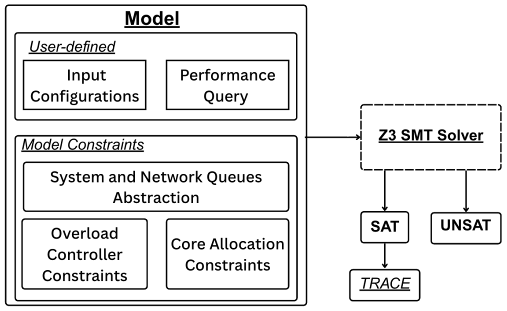


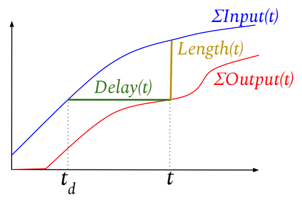
Fig. 1: Overview of the model architecture

C. *Performance Verification*

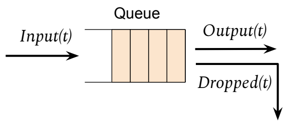
Performance verification leverages recent advances in Satisfiability Modulo Theories (SMT) solvers to verify that an algorithm maintains a certain level of performance under all valid assignments of a given model. Such worst-case analysis is particularly useful given the strict performance requirements imposed on modern algorithms. Performance verification requires a **model** of an algorithm, its environment, and its workload. The model is constructed using a set of first-order logic formulae over predicates of Boolean variables and real linear arithmetic inequalities. Performance properties are studied by formulating a **query**, a predicate of poor performance: can throughput be lower than threshold x? can latency be higher than threshold y? The SMT solver is used to find a satisfying assignment to the model. A satisfying assignment to the model can be viewed as a **trace** of poor performance. If the query is unsatisfiable, it is proof that the algorithm does not suffer from the poor performance property specified in the query. Figure 1 provides a high-level description of our model architecture.

A significant advantage of performance verification, over conventional modeling techniques (e.g., control theory), is that it relies on overapproximation instead of oversimplification when building a model. Overapproximation is an approach for improving the tractability of a model by allowing it to capture a superset of behaviors that the modeled system can exhibit. Thus, if a particular poor performance behavior is found to be unsatisfiable by the model, it follows that such behavior is also impossible in the real system. However, if a valid trace of poor performance is produced using an overapproximated model, it is up to the human modeler to verify whether such behavior is possible in the real system. If a trace shows behavior that cannot happen in practice, the model is further constrained to prevent such behavior. This iterative process enables the development of tractable, high-fidelity models.

How can worst-case analysis be of practical use? It is well established that measuring the maximum or minimum value of a performance measure of an evaluated practical system is not an effective measure to study its performance (e.g., maximum latency and minimum throughput). In particular, such extreme measures capture rare events that might even be completely exogenous to the system under study (e.g., power or network failure when studying a server scheduling algorithm). Thus, it is typical to consider different statistics of the performance measure (e.g., median and tail latency



(a) Visualization of the relationship between input and output curves



(b) A simple example to illustrate constraints.

Fig. 2: Model building block.

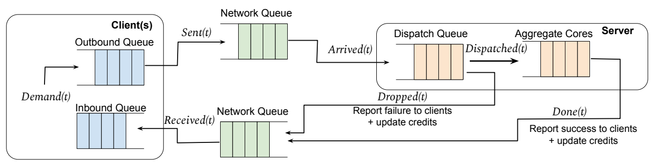


Fig. 3: Schematic view of the model. Each of $Demand(t)$, $Arrived(t)$, $Admitted(t)$, $Dropped(t)$ and $Done(t)$ is a cumulative curve representing the number of requests that passed through the respective point up to time t. $Admitted(t)$ is defined as the number of requests that didn't get dropped at the Dispatch queue (i.e., $Admitted(t) = Arrived(t) - Dropped(t)$).

and average throughput). However, a performance verification model only captures behaviors relevant to the system under study, ignoring any exogenous behaviors. Moreover, the model can be constrained to avoid any unreasonable or unrealistic behavior. Thus, performance verification can be viewed as a conditional worst-case analysis tool, where conditions are set to avoid obvious or unrealistic poor performance scenarios.

III. The Model

Our methodology relies on modeling the behavior of individual queues on the path of a request from the client until a response is generated by the server. The system is modeled as a sequence of queues where the output of a given queue is constrained by its input and the behaviors allowed by the resource management algorithm. Our model captures the behavior of the system in discrete time steps. For ease of exposition, we describe our model in continuous terms, and then describe how we discretize it. Finally, we describe our model of Shenango and Breakwater, respectively.

A. Modeling a Single Queue.

We define cumulative counters that represent the number of requests incoming and exiting a queue, referred to respectively as input and output curves, inspired by Network Calculus [19]. We allow the solver to pick any behavior for input and output curves, subject to physical and algorithmic constraints.

Physical constraints ensure the correct behavior of the model. A basic constraint ensures that task service times are positive values between T_{min} and T_{max}. Moreover, all requests that arrive at a queue must eventually leave. Thus, all queues have the following constraints: $\Sigma_j Output_j(t) \leq \Sigma_i Input_i(t)$, where $Input_i$ and $Output_j$ represent the cumulative number of requests at time t that arrived at input i and left the queue from output j, respectively. We model request drops as one output of the queue. Physical constraints also allow us to describe the state of the queue. The length of the queue is given by $Length(t) = \Sigma_i Input_i(t) - \Sigma_j Output_j(t)$. For a simple FIFO queue, the queuing delay experienced by an element dequeued at time t is $Delay(t) = t - t_d$, such that $\Sigma Input_i(t_d) = \Sigma Output_i(t)$. These relationships can be visualized as shown in Figure 2a.

Algorithmic constraints capture the specifications of the algorithm operating on the queue. We discuss the specification of Shenango and Breakwater later. As an example, consider an AQM policy that drops an incoming request if queueing delay exceeds some threshold, Th. To model such a queue, we define an auxiliary curve $InQueue(t) = Input(t) - Dropped(t)$ such that $InQueue(t_d) = Output(t)$ (Figure 2b). The following constraints define the AQM policy:

- If requests are dropped, it must be that the delay is too large and no new requests are admitted,
$(\frac{d}{dt}Dropped(t) > 0) \Rightarrow (Delay(t) > Th \wedge \frac{d}{dt}InQueue(t) = 0)$
- If requests are not dropped, it must be that the delay is small or no new requests arrive at all,
$(\frac{d}{dt}Dropped(t) = 0) \Rightarrow (Delay(t) \leq Th \vee \frac{d}{dt}Input(t) = 0)$

Another common property of queue management algorithms is work conservation. To model this property, we use constraints that have the format:
The resource is not full. $\Rightarrow$ All work is done.
Formally, consider elements dequeued by a server with capacity R. If fewer elements are dequeued than R, it should be either: 1) the queue is empty and the number of requests that arrived is smaller than R, or 2) the queue was emptied but it did not have enough load to consume all available capacity R.

B. Modeling a Sequence of Queues.

Our model captures the behavior sketched in Figure 3. In particular, we capture queues at the client whose output is constrained by the admission control decisions made at the server. At the server, we model the dispatcher queue, at which requests arrive at the server and are then dispatched to be processed by cores. The dispatcher queue can also drop requests, following Breakwater's AQM policy. Finally, we model the CPU queues that are constrained by the service time of requests (i.e., the processing capacity of the CPU and the nature of the requests).

The fundamental concern when modeling a sequence of queues is ensuring that the model size does not grow too large by incorporating every single queue in the model system. Thus, we identify opportunities to coalesce physical queues into a single queue in the model. In particular, our model captures the aggregate behavior of multiple cores using a single queue. We assume ideal load balancing, where all requests await in a single queue. Then, a request is dispatched to a core as soon as a core is available [3], [14] In addition, we assume that preemption is not allowed, consistent with the execution-to-completion model used by many high-performance stacks [1]–[3], [20]. Another step we take to improve the tractability of the model is to ignore any queueing in the network, leading to a network that only introduces a $RTT/2$ one-way delay.

The modeled sequence of queues is summarized in Figure 3. The cumulative number of requests sent by the client at time

Model Variables	
$Delay(t)$	Delay of most recent admitted request at time t
$Credit(t)$	Credits allocated to each client at time t
$Parked(t)$	Cores parked from application's perspective at time t
$T_{idle}[t-\tau,t]$	Total idle time of all cores in the interval $[t-\tau,t]$
$T_{processing}[t-\tau,t]$	Total processing time of all cores in the interval $[t-\tau,t]$
$T_{idle}^{parked}[t-\tau,t]$	Total time of all cores where no work (including busy polling) was done for the application
$T_{idle}^{allocated}[t-\tau,t]$	Total time of all cores spent on busy polling for work

TABLE I: Variables used in the implementation of the model alongside the variables introduced in Fig 3. **Model Variables** are assigned values by the backend solver (if possible), ensuring all the strict constraints are met.

Configuration Parameters		
T_{trace}	Duration of trace	$200\mu s$
RTT	Round-Trip Time	$20\mu s$
T_{min}	Minimum Service Time	$1\mu s$
T_{max}	Maximum Service Time	$5\mu s$
N	Maximum Allocated Cores Possible	20
Th	Maximum Queue Length (drop threshold)	$88\mu s$
t_{alloc}	Core Allocation Overhead	$2.5\mu s$
τ	Shenango Allocation Interval	$5\mu s$
TO	Shenango Parking Delay	$2\mu s$
α	Breakwater additive factor	1
β	Breakwater multiplicative factor	2.2
d_t	Breakwater Delay Threshold	$44\mu s$

TABLE II: **Configuration Parameters** are constants configured by policy/user, which can be classified into either policy-independent variables (such as RTT or T_{min}/T_{max}) and policy-dependent variables (such as TO or d_t).

t is $Sent(t)$. All requests sent by the client arrive at the server after an $RTT/2$ microseconds. Arriving requests are admitted to the server ($Admitted(t)$) or dropped based on an AQM policy ($Dropped(t)$). Admitted requests are dispatched to be processed by cores ($Dispatched(t)$), when CPUs are available. Completed requests are tracked with the cumulative variable $Done(t)$.

C. Model Discretization

The model described above captures the behavior of a sequence of queues through a set of continuous-time functions. We discretize these functions over time, representing their value at each discrete time step as a separate real variable. For example, the delay function of queue A, $Delay_A(t)$, is modeled as $Delay_A^0$, …, $Delay_A^{T-1}$ for T time steps, where $t \in \{0, \dots, T-1\}$. The constraints on the continuous function are applied to each of the individual variables. Discretization introduces two complications: 1) the need to define representative time steps to model and 2) the loss in model resolution by ignoring events between two discrete time steps.

To address these challenges, we leverage the observation that all algorithms studied are invoked periodically (e.g., every RTT). Periodic invocation is necessary to provide enough time for the control decision made by an algorithm to impact the behavior of the system. We leverage that fact by discretizing time at an interval corresponding to the most frequently made control decision. Further, we assume that other algorithms, invoked less frequently, are synchronized with that algorithm. In particular, core allocation is the most frequently made control decision (i.e., once every 5 microseconds). Overload control decisions are made once per RTT. Thus, we assume that an RTT is a multiple of 5 microseconds. This assumption limits the possible events that can happen between two time steps in our discrete model.

As we cannot precisely model the behavior of the system between two discrete time steps, we develop our model to overapproximate the behavior of the real system. In particular, we allow the solver to nondeterminsitically pick any valid behavior between two time steps. Thus, we define queueing delay for the queue in Figure 2b nondeterministically such that $t - t_d - \tau \leq Delay(t) < t - t_d$, where $InQueue(t_d) < Output(t) \leq InQueue(t_d+\tau)$. The solver can pick any value that satisfies these constraints. The only other overapproximated behavior is part of our model of Shenango, described next.

D. Modeling Shenango

The specification of Shenango is as follows. Core allocation is done periodically, every τ microseconds, based on request delay. In particular, if a request has been in the system for more than τ microseconds, an additional core is allocated to the latency-critical application. A core is deallocated (aka it parks), if it fails to find work after polling for TO microseconds. Modeling core allocation is straightforward because it is periodic, allowing us to allocate cores exactly at the modeled timesteps. Thus, the condition for core allocation is: $Dispatched(t) < Admitted(t-\tau)$.

Core deallocation is not periodic. Thus, we overapproximate the core deallocation behavior. A core parks based on the amount of time it spent idle. Thus, we track the aggregate idle time of cores between two time steps. In particular, the total idle time of cores at time t, since the last timestep $t-\tau$, is $T_{idle}[t-\tau,t]$. To accurately track core utilization, we differentiate between two aggregate metrics: the idle time of cores that did not park ($T_{idle}^{allocated}[t-\tau,t]$), and the idle time of the cores that parked including the time spent busy polling to find work ($T_{idle}^{parked}[t-\tau,t]$), where $T_{idle}[t-\tau,t] = T_{idle}^{allocated}[t-\tau,t] + T_{idle}^{parked}[t-\tau,t]$. Note that we use the notation $X[t_1,t_2]$, to indicate the aggregate value of X between time steps t_1 and t_2.

In order for a core to park between two time steps, $t-\tau$ and t, it must be that there has been a period TO where the core did not receive work. In particular, all tasks that were admitted at $t-\tau$, have been dispatched to other cores, leaving at least one core idle. Thus, the number of parked cores at time t can only be positive under the following condition:

$$Parked(t) > 0 \Rightarrow Admitted(t-\tau) \leq Dispatched(t)$$

The following constraints summarize the relation between $T_{idle}^{parked}[t-\tau,t]$, $T_{idle}^{active}[t-\tau,t]$, and $Parked(t)$ in a system with N cores:

- $Parked(t) \times TO \leq T_{idle}^{parked}[t-\tau,t] \leq Parked(t) \times \tau$

- $T_{idle}^{active}[t-\tau, t] \leq (N - Parked(t)) \times TO$.

We use the number of allocated cores and admitted requests to calculate the number of dispatched and completed requests between two modeled time steps $t-\tau$ and t. We categorize requests into three types, assuming $T_{max} \leq \tau$: (1) some requests, R_0, started processing before $t-\tau$, but did not finish yet by $t-\tau$, (2) other requests, R_1, started and finished processing within the interval, and (3) other requests, R_2, started before t but did not finish by t. Formally,

- $R_0 = Dispatched(t-\tau) - Done(t-\tau)$,
- $R_1 = Done(t) - Dispatched(t-\tau)$,
- $R_2 = Dispatched(t) - Done(t)$.

We refer to the time spent processing each of these categories within an interval T_0, T_1, and T_2, respectively, where

$$T_{busy}[t-\tau, t] = T_0[t-\tau, t] + T_1[t-\tau, t] + T_2[t-\tau, t]$$

All requests cannot have taken less than T_{min} or more than T_{max}. Thus, the following constraints must hold:

- $R_0 \times T_{min} \leq T_0[t-\tau, t] + T_2[t-2\times\tau, t-\tau] \leq R_0 \times T_{max}$,
- $R_1 \times T_{min} \leq T_1[t-\tau, t] \leq R_1 \times T_{max}$.

The above constraints represent the core allocation and deallocation behavior in Shenango, along with the corresponding dispatching and completion of requests.

E. Modeling Breakwater

Breakwater is a server-driven credit-based overload controller. Breakwater observes queueing delay at the server. If it is below the target delay, d_t, Breakwater additively increases the number of credits by α every RTT. If delay exceeds d_t, it multiplicatively decreases the number of credits proportional to the observed delay with a maximum multiplicative decrease factor of β. Credit adjustments can happen only once every RTT, ensuring that the effect of updates can be observed by the controller before making any further updates. Credit information are communicated explicitly to clients when they are granted and revoked. At low loads, Breakwater overcommits by issuing more credits than available capacity. Breakwater clients are work-conserving (i.e., if they have credits and demand, they have to send requests). The RTT is constrained to be a multiple of τ, ensuring that core allocation and overload control decisions are made at the modeled time steps. Only core parking decisions are overapproximated as stated earlier. Breakwater makes its decisions based on delay at the dispatch queue (i.e., time between a request being admitted and disptached). The complete algorithm is specified as follows:

- $(Delay(t) < d_t) \Rightarrow (Credit(t) = Credit(t - RTT) + \alpha)$
- $(Delay(t) \geq d_t) \Rightarrow$
$$\Big(Credit(t) = Credit(t-RTT) \times Max(1 - \beta \times \tfrac{Delay(t) - d_t}{d_t}, 0.5)\Big)$$

IV. Modeling Results

Our objective is to establish bounds on the throughput and latency of a latency-critical application running on a server that employs Shenango and Breakwater. To contextualize our results, we compare these bounds to those of a server that uses Breakwater but statically dedicates all available cores to the latency-critical application. We refer to the former as *Shenango* and the latter as *Static*.

Parameter configuration. We configure the parameter of the model so that it corresponds to the characteristics of real, deployed systems. Table II lists all the configuration parameters used in the model. We use a fixed RTT of $20\mu s$. We set T_{min} to $1\mu s$ and T_{max} and τ to the same value of $5\mu s$. The maximum number of cores that can be allocated is $N = 20$. Our model captures traces of $200\mu s$. We find this to be enough to identify significant performance issues in studied systems. This observation aligns with previous work on performance verification [9]–[11], which can be considered a manifestation of the "small scope" hypothesis [21].

We notice that for a duration of 200 μs, the server can process no more than $\frac{200N}{T_{min}}$ requests. Hence, we define the load as a fraction $L > 0$, and specify that $L \times \frac{200N}{T_{min}}$ requests should be generated at the client by the end of the trace. We sweep over values of $L \in \{0.25, 0.75, 1.25\}$, capturing states of low load, sufficient load, and very high load (overload). Moreover, we capture different levels of burstiness. In particular, we enforce n_{tasks} to arrive at the client in an interval of τ microseconds, with $n_{min} \leq n_{tasks} \leq n_{max}$, where n_{min} and n_{max} determine the burstiness of the load. We represent burstiness as a probability value $B = P(n_{min} \leq n_{tasks} \leq n_{max})$, assuming that n_{tasks} is a normally distributed random variable, with the distribution defined as $\mathcal{N} = (\mu, \sigma)$, where $\mu = \frac{L}{\frac{200}{\tau}}$ and $\sigma = \sqrt{\frac{L}{\frac{200}{\tau}}}$. We examine the values of $B \in \{0.25, 0.575, 0.9\}$. Note that unconstrained burstiness trivially leads to very high delays as the solver can create scenarios with arbitrarily large bursts, constrained only by the load on the system.

Initial conditions. Constraints on initial conditions ensure that a trace starts in a favorable condition, so that any subsequent bad performance can be more easily attributed to decisions by the studied algorithms. For example, poor initial conditions can lead to a worst-case latency of $200\mu s$ (the whole trace) if there were too many requests enqueued at the beginning of the trace. To exclude such behavior, we define the following initial conditions: (1) all cores are allocated for the first interval, (2) the initial credits, $Credits(0)$, are between $L \times \frac{RTT \times N}{T_{max}}$ and $L \times \frac{RTT \times N}{T_{min}}$, and (3) there are no standing queues (i.e., all queues are empty).

Computing bounds on performance. We formulate a query $\mathcal{Q}_{L,B}^{metric}(x)$ that asks: given load L and burstiness B, can $metric$ be lower or higher than x? For each pair (L, B), a very optimistic x makes $\mathcal{Q}_{L,B}^{metric}(x)$ satisfiable, and a very pessimistic x makes it unsatisfiable. For queries measuring worst-case throughput, we sweep over $x \in \{0..L\}$, and find the lowest satisfiable x. For latency queries, we allow the worst state possible to be chosen by the solver for the initial one-third of the trace, and then sweep over values of x starting from zero to find the largest satisfiable x that represents the delay of the system at that state.

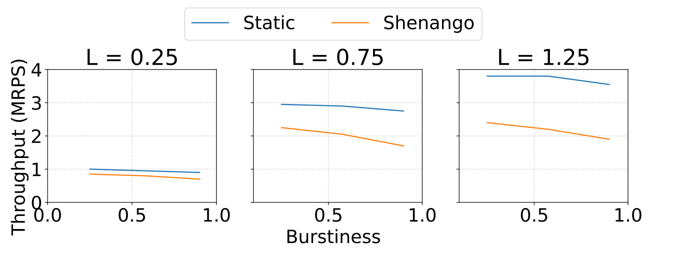


Fig. 4: Lower bound on throughput in Million Requests per Second (MPRS) as a function of burstiness (B) and load (L). Shenango's worst-case throughput is half of Static at high load. Shenango can park most cores even at high loads, reducing capacity which increases latency, prompting Breakwater to admit less load.

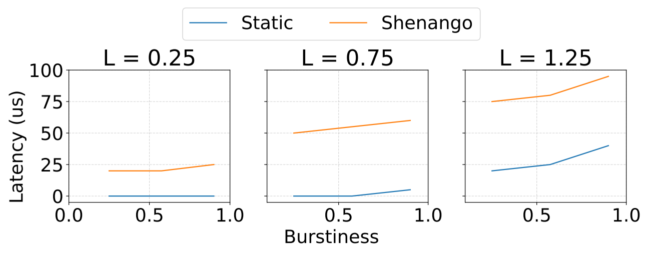


Fig. 5: Upper bound on queueing delay at the server in microseconds as a function of burstiness and load. As load increases, the upper bound on delay grows. Note that Static achieves near zero delay at low loads because all incoming requests can be processed as soon as they arrive. As load increases, the effect of burstiness on Static grows. On the other hand, Shenango suffers from high delay because all cores can be deallocated, leading to high delays even at low loads.

1) Establishing bounds on throughput

Ideally, throughput should be equal to the minimum of load and capacity. In particular, given N cores and requests that can take at most T_{max} time units, the worst-case throughput should be $L\frac{200N}{T_{max}}$ over a trace of 200μs. Figure 4 summarizes our findings. As load increases, the worst-case throughput increases, as expected. However, the gap between Shenango and Static grows as load increases. Similarly, as the burstiness grows, both systems suffer. However, the gap between them increases with medium and low loads. The reasoning behind these observations is described next.

Static. Statically allocating all cores in the system leads to near-ideal behavior for throughput, at the expense of keeping cores allocated even when load is below capacity. In particular, the throughput of Static is bounded by $Credits(0)$, achieving a throughput of $Credits(0) \times \frac{200}{RTT}$. If $Credits(0) = L \times \frac{RTT \times N}{T_{max}}$, the system achieves the maximum possible worst-case throughput. Otherwise, Breakwater's additive increase issues more credits until ideal throughput is achieved.

Shenango. Shenango's worst-case scenario, across different loads, occurs when the allocated cores finish their work simultaneously (within the same time interval) and all remain idle for $TO = 2\mu$s, eventually not finding enough work and parking together. This behavior limits available capacity, as Shenango can only add a single core every 5μs. Despite this behavior, Shenango can sustain near-ideal throughput at low loads. In particular, it can sustain loads for which it can allocate all the needed capacity in a single RTT. Under our configurations, Shenango can allocate a maximum of four cores in a single RTT, allowing it to sustain $L \leq 0.2$ when $N = 20$. Our results show that at $L = 0.25$, the worst case throughput of Shenango is between 72-85% of the lower bound of Static. At high loads, $L \geq 0.75$, even when Shenango starts with all cores allocated, the frequent and repeated deallocation of cores leads to queue buildup and high latency, forcing Breakwater to reduce the credit pool. In turn, Shenango finds less requests to process, allowing it to deallocate cores even more frequently. Another point to note is that our traces for high loads have initial conditions similar to final conditions, allowing for the possibility of this behavior repeating indefinitely. Thus, we conclude that the interaction between core allocation and overload control is detrimental to throughput, potentially lowering average throughput over large timescales. We were able to observe the reproduction of this worst-case behavior in simulation. An added value of this model is that practitioners can plug their system configurations in the model (e.g., expected load and burstiness) and decide if the worst-case behavior of Shenango (or other core allocation policies) is acceptable to their use case.

2) Establishing bound on latency

Figure 5 summarizes the worst-case latency results. *Static* exhibits near-ideal performance, with a worst-case latency of 0μs for low loads. With higher loads, the worst-case latency increases with burstiness, resulting in 5μs for very bursty traffic at moderate load, and up to 40μs for overloaded scenarios. Shenango exhibits high latencies even at low loads. Note that employing an overload controller ensures that latency is always bounded to a value near the target delay of the overload controller.

Static. The model creates worst-case delays by attempting to create the largest possible burst to create a long queue at the server. At low loads, queueing is impossible because even the largest burst is smaller than the capacity dedicated by the Static policy. As load increases, large bursts can overwhelm the server. However, the latency never exceeds 10 T_{max}, which is a reasonable latency SLO [5].

Shenango. Cores are deallocated even when the number of issued credits is large (e.g., $Credit(t) = L \times \frac{RTT \times N}{T_{min}}$). Thus, a queue length of $Credit(t)$ is possible (e.g., up to 25μs of delay at low loads). Furthermore, Shenango allocates cores slowly, letting the queue stand while the allocated capacity ramps up. A standing queue causes Breakwater to reduce the number of credits, lowering throughput. While high tail latency is intrinsic to Shenango, the interaction between the two controllers can also degrade throughput when Shenango causes high latency. Such behavior can be mitigated if the number of credits is adjusted to be proportional to the number of allocated cores.

V. Model Validation

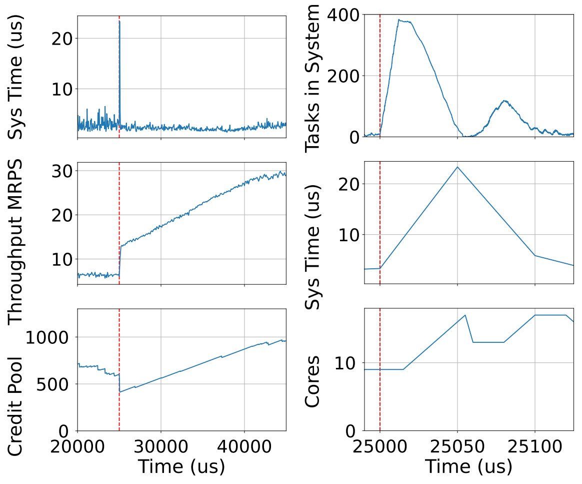
Our objective is to validate the plausibility of the worst-case behavior identified by our model. Worst-case analysis can potentially lead to contrived behavior that cannot be reproduced in simulations or in practice. In this section, we reproduce the worst-case behavior identified by the model in simulation. We augment the simulator developed in prior work to study core allocation policies [3] by implementing overload control [22]. If some simulations show particularly bad performance statistics over the whole trace, we examine the trace to check the microsecond-scale behavior. We omit results over longer traces for brevity. However, we note that the performance results observed over long traces were consistent across simulations and the implementation of the policies.

Our simulations capture the behavior of Shenango and Breakwater with high fidelity. Both algorithms are configured with target delays of $10\mu s$. Breakwater's AIMD algorithm has an α of 1 and a β of 0.08. The network RTT is 25μs. The server has 32 cores that Shenango can dynamically allocate to a single application. 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 μs tasks per second). The delay for allocating a core is 5μs. Request service time has an average of 1μs with a bimodal distribution. Our request of short requests with a bimodal distribution increases the likelihood of bursts and queueing due to head of line blocking.

The essence of the poor performance scenarios produced by the model is that some arrival patterns can lead to the deallocation of most cores, requiring a considerable amount of time for the system to recover while building up queues. Our simulations find two examples of such scenarios: 1) incast scenarios where many clients send their requests simultaneously, eventually leading to throughput collapse, and 2) a sharp increase in load.

Incast-induced Throughput Collapse. Consider a scenario where many clients, 40 in this scenario, each possessing a single credit, submit their requests simultaneously. In this scenario, the credit pool size is small and almost all cores are deallocated. Such severe conditions can occur after a period of repeated congestion, or a long period of idleness (i.e., hundreds of microseconds). Incoming requests arrive in a burst and face long delays, awaiting cores to be allocated. Long delays prevent the number of credits from growing. Clients cannot send more requests until they receive responses, because the credit pool did not grow. Moreover, there can be a delay between a core finishing a request and receiving another, allowing cores to be deallocated. This is especially true in scenarios where the RTT is much larger than TO and T_{min}. Thus, the system gets stuck in the same state: many clients, few credits, and a small number of allocated cores. This behavior was also observed by our model, when worst-case throughput is achieved at high loads. Even in the presence of high demand, this behavior can persist until some perturbation is introduced (e.g., requests arrive in a small burst, allowing the credit pool to grow).

Slow reaction to load changes. Consider a scenario where load sharply increases, from 0.2 million requests per second (MRPS), requiring only 7 cores, to 1.2 MRPS requiring all 32 cores. Incoming requests will need to wait for additional cores to be allocated, experiencing high delays. This high delay



(a) Overall View (b) Zoom at 25,000 μs

Fig. 6: Load increases (0.2 to 1.2 capacity at 25,000 μs). Left is a macroscopic view of throughput and credit pool size. Right is a microscopic view of allocated cores and processed tasks.

can trigger the overload controller to reduce the number of allocated credits. Under such conditions, the core allocator adds more capacity while the overload controller reduces the load, leading to low throughput. This was observed in worst-case scenarios for latency in our model.

Figure 6a illustrates the problem: the credit pool size decreases at the instant load increases, elongating the time it takes for the credit pool to grow to achieve high throughput. The above scenario is exacerbated when the credit pool size decreases too much that queueing delay at the server is so small that no additional cores are allocated. Specifically, the sharp increase in load creates a very large burst. The AQM in the overload controller drops most of the burst, allowing the load at the server to drop, requiring fewer cores. Figure 6b shows the microscopic behavior of the system. Much of the burst gets dropped (tasks in system decreases sharply). Remaining tasks observe high delays (high system time). Because most of the burst was dropped and the credit pool size was reduced, the system becomes idle, allowing cores to park. Thus, even though load increased, both the credit pool size and the number of allocated cores decreased, harming throughput and latency.

Empirical Validation: Our contemporary work provides empirical validation of our findings [13]. In particular, we quantified the performance degradation due to poor interactions between the two controllers. Based on our observations, we propose CoreSync, a server-driven credit-based protocol that maintains partial proportionality between cores allocated and credits available to clients for better performance. Specifically, under low load, CoreSync improves tail latency by 1.7x compared to Shenango, and in overloaded scenarios, it improves average throughput by 5.8%.

VI. Related Work

Modeling Network Congestion Control. Overload control algorithms share many commonalities with congestion control algorithms. The traditional approaches for analytically studying the performance of congestion control algorithms focused on studying the resource allocation aspects of the problem (e.g., network utility maximization [23]–[25]) or the algorithmic aspects (e.g., stability and convergence with control-theoretic approaches [26], [27]). These models tend to make many oversimplifying assumptions. Further, it is hard to leverage similar techniques to model multiple algorithms interacting with each other. In this paper, we follow a similar approach, developing a new model that captures the behavior of an overload controller operating with a dynamic core allocation algorithm. Our methodology represents a departure from that used by CCAC [9], where a single queue captures the aggregate behavior of the entire network path. CCAC's abstraction is warranted due to its focus on end-to-end congestion control algorithms which do not have visibility nor control over the individual queues in the network.

Performance Verification. Applying formal verification methods to study the performance of algorithms is a nascent area of research [10], unlike its application to studying the correctness of systems and network implementation and configuration. CCAC [9] and FPerf [11] build formal models for congestion control and packet scheduling algorithms. Our approach has been directly inspired from these principles of building abstract models with specifications to verify them. FPerf provides a tool that allows for synthesizing generalized traces that cause poor performance. MetaOpt [28] allows identifying the gap between a heuristic and an optimal algorithm. MetaOpt focuses on cases where the optimal algorithm and the heuristic can be defined precisely as optimization problems, providing an efficient solver for cases where the search space is convex. Our focus is on studying the performance of heuristics, providing absolute bounds on their performance. We leave identifying the optimal algorithm for future work.

VII. Conclusion

We show that the growing complexity of resource management systems in large-scale servers can result in conflicting decisions that can lead to significant performance degradation. In particular, we focus on the interaction between core allocation and overload control algorithms developed for microsecond-scale tasks. In such scenarios, both controllers operate at comparable timescales, increasing the chances of interference between their decisions. We employ performance verification to model a system that employs the two controllers simultaneously. Our model allows us to establish lower bounds on throughput and upper bounds on latency as a function of load and burstiness with the combined behavior of overload control and core allocation significantly degrading performance in the worst case, compared to scenarios where a fixed number of cores is dedicated to an application. Our model can be extended to capture other core allocation and overload control policies, helping practitioners in a broader range of scenarios. We validate our findings in simulations, showing that worst-case behaviors identified by the model are possible even under stochastic arrival patterns and service times. We hope that this paper serves as proof that dealing with the growing complexity of resource management systems requires relying on better modeling tools to provide more concrete guarantees on performance. Moreover, we believe that coordination policies between such controllers are necessary to provide better guarantees on overall system performance.

Acknowledgment

The authors thank anonymous reviewers. This work was funded in part by NSF grants CNS-2212103 and NS-2212098 and a Google Research Award.

References

[1] A. Ousterhout *et al.*, "Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads," in *USENIX NSDI*, 2019.
[2] J. Fried *et al.*, "Caladan: Mitigating interference at microsecond timescales," in *USENIX OSDI*, 2020.
[3] S. McClure *et al.*, "Efficient scheduling policies for Microsecond-Scale tasks," in *USENIX NSDI*, 2022.
[4] H. Qin *et al.*, "Arachne: Core-Aware thread management," in *USENIX OSDI*, 2018.
[5] I. Cho *et al.*, "Overload control for µs-scale RPCs with breakwater," in *USENIX OSDI*, 2020.
[6] ——, "Protego: Overload control for applications with unpredictable lock contention," in *USENIX NSDI*, 2023.
[7] H. Zhou *et al.*, "Scalable overload control for large-scale microservice architecture," in *SoCC*, 2018.
[8] M. Welsh *et al.*, "Overload management as a fundamental service design primitive," in *SIGOPS European Workshop*, 07 2002.
[9] V. Arun *et al.*, "Toward formally verifying congestion control behavior," in *ACM SIGCOMM*, 2021.
[10] S. Goel *et al.*, "Quantitative verification of scheduling heuristics," *arXiv preprint arXiv:2301.04205*, 2023.
[11] M. T. Arashloo *et al.*, "Formal methods for network performance analysis," in *USENIX NSDI*, 2023.
[12] A. Agarwal *et al.*, "Towards provably performant congestion control," in *USENIX NSDI*, 2024.
[13] B. Pardeshi *et al.*, "Coresync: A protocol for joint core scheduling and overload control of µs-scale tasks," in *IEEE ICNP*, 2025.
[14] G. Prekas *et al.*, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *ACM SOSP*, 2017.
[15] J. C. Mogul *et al.*, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, 1997.
[16] G. Kumar *et al.*, "Swift: Delay is simple and effective for congestion control in the datacenter," in *ACM SIGCOMM*, 2020.
[17] K. Nichols *et al.*, "Controlling queue delay," *Communications of the ACM*, vol. 55, no. 7, pp. 42–50, 2012.
[18] R. Srikant *et al.*, *The mathematics of Internet congestion control.* Springer, 2004.
[19] J.-Y. Le Boudec *et al.*, *Network calculus*, 2001st ed., ser. Lecture notes in computer science, J.-Y. Le Boudec *et al.*, Eds. Springer, Jul. 2001.
[20] A. Belay *et al.*, "IX: A protected dataplane operating system for high throughput and low latency," in *USENIX OSDI*, 2014.
[21] D. Jackson *et al.*, "Elements of style: Analyzing a software design feature with a counterexample detector," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 239–249, 1996.
[22] E. Stuhr *et al.*, "Poster: Understanding interactions between overload control core allocation in low-latency network stacks," in *ACM SIGCOMM*, 2023.
[23] F. P. Kelly *et al.*, "Resource pooling in congested networks: proportional fairness and product form," *Queueing Systems*, vol. 63, pp. 165–194, 2009.
[24] F. Kelly, "Charging and rate control for elastic traffic," *European transactions on Telecommunications*, vol. 8, no. 1, pp. 33–37, 1997.
[25] J. Jaffe, "Bottleneck flow control," *IEEE Transactions on Communications*, vol. 29, no. 7, pp. 954–962, 1981.
[26] M. Alizadeh *et al.*, "Stability analysis of qcn: the averaging principle," in *ACM SIGMETRICS*, 2011.
[27] ——, "Analysis of dctcp: stability, convergence, and fairness," in *ACM SIGMETRICS*, 2011.
[28] P. Namyar *et al.*, "Finding adversarial inputs for heuristics using multi-level optimization," in *USENIX NSDI*, 2024.