Efficient Scheduling Policies for Microsecond-Scale Tasks

Sarah McClure and Amy Ousterhout, *UC Berkeley;*
Scott Shenker, *UC Berkeley, ICSI;* Sylvia Ratnasamy, *UC Berkeley*

https://www.usenix.org/conference/nsdi22/presentation/mcclure

This paper is included in the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Open access to the Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Efficient Scheduling Policies for Microsecond-Scale Tasks

Sarah McClure[⋆]*, Amy Ousterhout*[⋆]*, Scott Shenker*[⋆†]*, Sylvia Ratnasamy*[⋆]
[⋆]*UC Berkeley* [†]*ICSI*

Abstract

Datacenter operators today strive to support microsecond-latency applications while also using their limited CPU resources as efficiently as possible. To achieve this, several recent systems allow multiple applications to run on the same server, granting each a dedicated set of cores and reallocating cores across applications over time as load varies. Unfortunately, many of these systems do a poor job of navigating the tradeoff between latency and efficiency, sacrificing one or both, especially when handling tasks as short as 1 μs.

While the implementations of these systems (threading libraries, network stacks, etc.) have been heavily optimized, the policy choices that they make have received less scrutiny. Most systems implement a single choice of policy for *allocating cores* across applications and for *load-balancing* tasks across cores within an application. In this paper, we use simulations to compare these different policy options and explore which yield the best combination of latency and efficiency. We conclude that work stealing performs best among load-balancing policies, multiple policies can perform well for core allocations, and, surprisingly, static core allocations often outperform reallocation with small tasks. We implement the best-performing policy choices by building on Caladan, an existing core-allocating system, and demonstrate that they can yield efficiency improvements of up to 13-22% without degrading (median or tail) latency.

1 Introduction

Modern datacenter applications often involve many short Remote Procedure Calls (RPCs) to other servers. These RPCs allow applications with large memory footprints to access memory on other servers [2, 49, 51, 62, 69], enable applications to leverage large amounts of compute over short timescales [6, 25, 46], and provide replication and consensus [58]. The service times of these tasks grow ever smaller, and today are often a single microsecond or less [10, 34].

Tasks with short service times are particularly vulnerable to latency inflation; even small overheads can increase the latency of a 1 μs task by an order of magnitude [10]. This is problematic for today's applications, which depend on low latency both at the median and at the tail of the distribution (e.g., 99% latency) [5, 19]. As a result, researchers have proposed many techniques to reduce the overheads of handling these short tasks. These systems improve software with low-latency network stacks and better load balancing (DPDK [1], ZygOS [66], Shinjuku [36], eRPC [38], etc.) or propose new hardware to deliver packets to cores more quickly (RPC-Valet [18], NeBuLa [74], NanoPU [34], Cerebros [65]). They offer tail latencies of a few dozen microseconds with existing hardware [26, 38] or several microseconds with new hardware [34].

However, as Moore's Law slows [23], datacenter operators are increasingly concerned not just with providing low latency but also with achieving high CPU efficiency [79]. To do so, they pack multiple applications on the same server so that background applications can use any CPU cycles not used by latency-sensitive applications, as their load varies over time [11, 35, 80]. Several recent research systems enable this deployment model by allocating a set of cores to each application and then reallocating cores across applications as load changes (e.g., IX [12], PerfISO [35], Arachne [67], Shenango [60], Caladan [26], and Fred [40]). These systems walk a delicate tightrope, attempting to make spare cycles available for batch applications without harming the latency or throughput of latency-sensitive applications. Thus researchers have heavily optimized these systems' implementations, squeezing spare CPU cycles and extraneous cache misses out of their network stacks, threading libraries, and core-allocation mechanisms.

While there have been significant advances in these mechanisms, less effort has gone into studying the *policies* that these core-reallocating systems implement. Each system implements two main policies: (1) a policy for **load-balancing** tasks across cores within an application and (2) a policy for when to **reallocate cores** from one application to another. There are many possible choices for each policy: popular load-balancing policies include work stealing [14], work shedding, and steering tasks to less-loaded cores when they are first enqueued [55] while core-allocation policies may be based on queueing delay [12, 26, 60], the arrival of new tasks [40], or CPU utilization [35, 67]. And yet, each system typically implements a single choice of load-balancing and core-allocation policy, providing little clarity about how different policies compare.

Unfortunately, as we will show (§2), these policy choices can contribute to suboptimal performance, with existing systems sacrificing significant CPU efficiency in order to maintain low latency, especially with short tasks. The root of the problem is that as task durations shrink from 100 μs to 1 μs, the overheads of balancing tasks or reallocating cores (e.g., a 50 ns cache miss to probe state on a different core) become relatively more significant, and inefficient policies become much more costly. In this paper, we focus on these policies and ask: *what load-balancing and core-allocation policies*

yield the best combination of latency (median and tail) and CPU efficiency for microsecond-scale tasks?

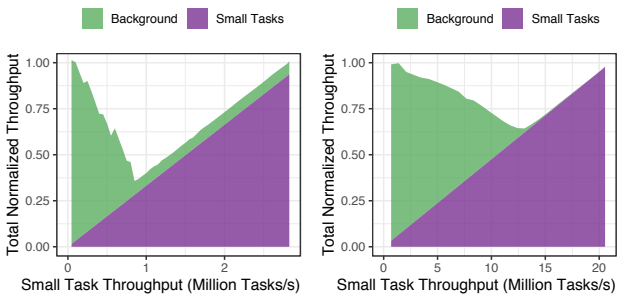
We focus on the *combination* of latency and efficiency because while ideally we would like to optimize both, there is an inherent tradeoff between the two. For example, allocating infinite cores could achieve optimal latency at the cost of terrible efficiency, while allocating a single core could achieve good (but perhaps not optimal) efficiency, but potentially high latency. The best one can hope for is to operate on the Pareto frontier of latency and efficiency; i.e., a point where it is not possible to improve one quantity without harming the other.

To compare policies fairly and independently from any specific implementation, we turn to simulations (§4). We use measurements of real systems to estimate the overheads of balancing tasks across cores within an application and of reallocating cores across applications. We then model simple versions of common load-balancing and core-allocation policies, and simulate them using our estimated overheads. We use these simulations to conduct an extensive factor analysis, teasing apart the impact of load-balancing policies and core-allocation policies on both latency and efficiency. From this analysis, we glean three key insights:

First, assuming commodity NIC hardware, *work stealing* is the load-balancing policy that yields the best latency and CPU efficiency and forms the Pareto frontier. We find that this conclusion is remarkably robust across different average service times, service time distributions, numbers of cores, latency metrics (e.g., median vs. 99%), whether cores are dynamically reallocated or statically partitioned, and how much overhead load-balancing a task entails.

Second, in contrast, our analysis of core-allocation policies shows that multiple policies can perform similarly well (though some policies perform significantly worse). We find that *revoking cores proactively, rather than waiting until they go idle to yield them to another application, makes it easier to achieve good efficiency with small tasks, especially with many cores*. We identify two policies (based on average queueing delay and average CPU utilization) that fit this criteria, perform well, and can be configured to make different tradeoffs along the Pareto frontier; two other policies used in current systems yielded worse latency, CPU efficiency, or both.

Third, even with the best core-allocation policies, if the average load is fixed, *with small tasks it is difficult to achieve better performance by reallocating cores than by allocating a fixed number of cores*. For our request patterns (modeled with exponentially-distributed inter-arrival times), reallocating cores in response to transient bursts does not improve latency (median or tail) relative to statically allocating the same average number of cores. Thus the main benefit of reallocating cores over short timescales with short tasks is the ability to quickly adapt to changes in *average* load. In contrast, when average task service times are longer—several microseconds or more—we find that reallocating cores does improve performance even with constant average load.



(a) Arachne. **(b)** Caladan.

Figure 1: Total useful work done by two colocated applications—one background, the other handling small (about 1 μs) memcached tasks—as we vary memcached's load (for two existing systems).

From this factor analysis we conclude that barring technology changes (e.g., commercialization of recently proposed NIC hardware [18, 34, 65, 74]), for low latency and high CPU efficiency, work stealing is the best load-balancing policy, and our two new core-allocation policies based on average delay or average utilization (we refer to these policies as "delay range" and "utilization range") perform best. We implement these policies in a real system by extending Caladan [26], a state-of-the-art system for reallocating cores which already supports work stealing. We demonstrate that when running memcached, a key-value store, delay range and utilization range can save up to 13-22% of cores relative to Shenango's and Caladan's core-allocation policies, without degrading median or tail latency (§6).

2 Motivation

To demonstrate the inefficiencies of existing systems when handling short tasks, we conduct an experiment in which we run two applications on a server: a latency-sensitive application that handles short tasks and a background application that consumes all extra CPU cycles. We use memcached [49], a key-value store with service times of about 1 μs, as our latency-sensitive application. We vary the offered rate of memcached tasks and measure how much useful application-level work each application completes. We perform this experiment with two existing systems: Arachne [67] and Caladan [26].

Both systems yield latency improvements: Arachne's 99% latency improves on that of Linux by hundreds of microseconds, while Caladan reduces this further, due partially to replacing Linux's network stack with kernel bypass. However, in their efforts to provide low latency for the small tasks, these systems waste significant CPU resources. Figure 1 shows the total throughput achieved by each system, normalized by the maximum throughput the application can achieve when running alone on the configured set of cores (16 for Arachne and 32 for Caladan). Thus at the lowest and highest loads (where only one of the applications is running[1]), both systems are

[1] Arachne dedicates one core to each application, so its background throughput never reaches zero.

at their highest possible efficiency, achieving a total normalized throughput of 1.0. Ideally, the total throughput of both applications would remain at 1.0 as the small task load varies. However, at moderate loads, both systems suffer significant efficiency losses, wasting up to 64% or 36% of their cores, with Arachne and Caladan, respectively. This inefficiency is not exclusively bad; the excess cycles can be used to handle small tasks sooner, lowering latency.

From these results, it is clear that these systems are able to multiplex cores between applications, but they are extremely inefficient while doing so. When handling longer tasks (e.g., 10 μs or 100 μs), these systems become much more efficient. This begs the question: what is responsible for these efficiency losses with short tasks? These systems differ along many different dimensions: their core-allocation policies, their load-balancing policies, their threading libraries, and whether they use the Linux network stack (Arachne) or kernel-bypass (Caladan). The latter implementation aspects can contribute significantly, but they have been studied extensively by prior work. We focus instead on the policy aspects and seek to understand which load-balancing and core-allocation policies yield the best performance for small tasks.

3 Design Space of Policies

If reallocating cores across applications and load balancing tasks between cores incurred no overhead (i.e., they could be done instantaneously), the optimal policies would be: (1) immediately grant an application a new core whenever a task arrives and yield the core when the task completes and (2) steer each newly arrived task to its newly granted core. With these policies, CPU usage would exactly match the time spent on tasks (100% efficient) and if an additional core was always available then tasks would never queue (zero added latency).

These idealized policies are sufficient with long task service times (e.g., 100 μs or more), because the overheads of load balancing and core reallocation are relatively small (§4.3). However, with tasks as short as a single microsecond, load-balancing and core-allocation overheads become significant and we can no longer afford to perform both a core-allocation and a load-balancing action for every task that arrives; doing so wastes considerable CPU resources. For good performance with short tasks we must consider other policies. The key difference between distinct policies is when they choose to incur overheads (e.g., when a task arrives vs. when a queue builds up), and these choices determine their latency and CPU efficiency. Thus finding the best load-balancing and core-allocation policies amounts to asking the question: *given that load balancing and core allocation incur overheads, how should we spend those overheads most effectively?*

3.1 Setting and Assumptions

While exploring different policies, we make several assumptions about our setting (illustrated in Figure 2). We assume that each server runs one or more *applications*, where each application is either a batch application that seeks high throughput and is latency-insensitive or a latency-sensitive application that handles short *tasks*. Each application is always allocated a specific number of cores; when an application yields a core, the core will be granted to another application if possible.

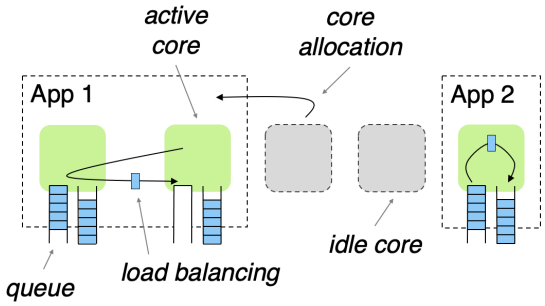


Figure 2: Applications use *load balancing* to balance tasks across cores and *core allocations* to adjust the number of cores available to each application.

Tasks can either arrive from external sources (e.g., a packet arrives over the network or a storage operation completes) or be created by the local CPU (e.g., a thread spawns a new thread). We focus on settings with commodity NICs that spray packets randomly over available cores (e.g., with RSS [3]), though we also discuss how performance could change with recent proposals for new NIC hardware with advanced steering capabilities (§4.2.1). Unless specified otherwise, we assume that each core maintains its own queue(s) of tasks and that tasks are not intentionally re-ordered (cores handle them in FIFO order). We assume no preemption of running tasks and no *a priori* knowledge of how long each task will take to run.

3.2 Policies

In this section, we summarize the main policies used for load balancing and core allocation today and describe when each incurs overheads; these are the policies we evaluate in our factor analysis (§4). The list is not exhaustive but rather an attempt to cover the main classes of existing policies as well as the theoretically optimal policies.

3.2.1 Load-Balancing Policies

Load-balancing policies can perform load balancing either when a task arrives or once it has already been queued. The latter category can be further divided based on what triggers load balancing (either a lack of tasks for a core or a core with too many tasks). We begin by describing a theoretical optimum, and then describe four practical policies that fall into these categories. Note that these policies are not necessarily mutually exclusive.

Single queue. With no overheads, the theoretically ideal load-balancing policy places all tasks in a single shared queue. However, this approach limits throughput in practice due to contention for the single queue. Shinjku [36] and RAMCloud [62] take this approach; Shinjuku can support only

System	Load-balancing Policy	Core-allocation Policy	
		Trigger for Adding a Core	Trigger for Revoking a Core
IX [12]	none	packet queueing delay	low CPU utilization
Arachne [67]	choice on enqueue with power-of-two choices [55]	number of runnable threads	low CPU utilization
Shenango [60], Caladan [26]	work stealing	max queueing delay of threads or packets	failure to work-steal
Fred [40]	steering on arrival or work-stealing once all cores are allocated	task arrives	task completes
Go [76]	work stealing	task arrives and no cores work stealing	failure to work-steal

Table 1: Load-balancing and core-allocation policies used by existing systems. Core-allocation policies are highlighted to indicate whether they rely on queueing, utilization, task arrival, or failure to find work.

about 5 million requests per second with a single queue.

No load balancing. Without load balancing, tasks are handled by the core they first arrive at, such as the core that spawns a thread or the core a packet or storage completion is steered to by hardware [12, 42, 64]. This approach incurs no load-balancing overheads.

Enqueue choice. Enqueue choice policies make a load-balancing decision about which core to assign a task to when the task is first created; tasks cannot be moved later. Existing systems commonly use "power of two choices" [55] to enqueue a task to the less-loaded of two randomly sampled cores [33, 67, 81]. When a task is first created, the creating core incurs overhead to sample queues on other cores (which can be done in parallel for small numbers of sampled cores) and to enqueue the task to the chosen core.[2]

Work stealing. When a core is idle, it searches for a core that has queued work, and then steals half the tasks from that core and moves them to its own queue [14]. This approach is used by the Go runtime [76], several multithreading platforms [9, 16, 17, 45, 47, 59, 68], and many research systems [26, 40, 44, 50, 60, 66, 81]. It incurs overhead to check other cores for queued work and move work to its local queue.

Work shedding. With work shedding, overloaded cores can shed load to other cores or request that other cores take some of their load. This has been considered by several theoretical papers [22, 73, 77] and for load-balancing systems in a variety of contexts [56, 78]. We consider a work-shedding policy in which a core that has had tasks queued for longer than a specified threshold selects a random core and indicates that it is overloaded. That core will then respond by stealing half of the overloaded core's tasks; this is the primary source of overhead for this policy.

3.2.2 Core-Allocation Policies

All core-allocation policies incur overhead in the same way: by adding or revoking a core. Their overheads are primarily determined by how often they reallocate cores and consist of both the latency until a core is available after a reallocation decision is made and the CPU cycles that cannot be used productively while a core is being reallocated. The performance of each policy is determined by how effective the signals are that it uses to trigger core reallocations. Most policies make core-allocation decisions at fixed time intervals (e.g., every 5 μs [60]), though some are triggered by other conditions.

We cannot easily model or compute an optimal core-allocation policy, i.e., one that achieves the optimal tail latency for a given CPU efficiency or vice versa. This is because finding the optimal tail latency for a given CPU usage bound or vice versa is NP-hard assuming a finite number of cores and non-constant service times; this can be shown by a reduction from the multiprocessor scheduling problem (see Appendix A.1). We now list the core-allocation policies we consider.

Static. With static core allocations, the number of cores allocated to each application cannot change over time, as in several research systems [42, 64, 66]. This incurs no overhead for core reallocations. However, each application must be provisioned with enough cores for peak load, wasting significant CPU resources as load varies over time, which is typical of datacenter workloads [11, 35].

Per-task. Systems such as Fred [40] with per-task core allocations grant a core to an application every time a task arrives. This incurs the overhead of a core allocation for each task, except when all cores are in use.[3]

Queueing-based. Policies based on queueing delay grant an application an additional core if the queueing—as measured by either the number or delay of threads, packets, or storage completions—exceeds a certain threshold. These policies vary in whether they trigger based on the maximum queueing across cores [26, 60] or use an average [12, 67].

CPU utilization-based. Utilization-based policies add or revoke cores based on the number of idle cores [35] or the average fraction of time cores spend working on tasks (as opposed to sitting idle or busy-spinning) [12, 67].

Failure to find work. In some systems, an application will yield a core when the core is unable to find any tasks to work on. This can happen when a core fails to find another core with queued work to steal from [26, 60, 76] or when it finishes its current task, with a per-task core-allocation policy [40].

3.3 Overheads

Both load balancing and core allocation entail overheads; in this section we discuss the magnitude of these overheads in typical systems today.

[2] Note that "no load balancing" is a special case of enqueue choice in which there is only one choice and no overhead.

[3] Once all cores are allocated to an application, Fred places additional arriving tasks in per-core queues and cores use work stealing to find them.

Load-balancing overheads. Load-balancing overheads can be impacted by several factors: the CPU architecture (how long does it take to handle a cache miss? how many cache misses can be outstanding simultaneously?), the workload (how often is load-balancing state cached locally vs. modified on remote cores?), and speculative execution (how successfully can the CPU overlap cache misses with other instructions via speculative execution?). Despite these factors, we attempt to estimate the overheads of different load-balancing policies and in Section §4.2.1 we demonstrate that our conclusions about the relative performance of different policies are unlikely to change with different overheads.

Because load balancing requires communication between cores, its overhead arises primarily from cache misses while retrieving cache lines from the L2 cache of another core. Depending on the CPU microarchitecture, one such cache miss can cost between 30 ns (Intel Haswell) and 200 ns (Xeon Phi) [72]. A load-balancing operation moves state from one core to another; this typically entails about three cache misses to read a remote cache line, invalidate it so that it can be written in the local cache, and then a third cache miss when the remote core reads the modified cache line [67]. The overhead incurred by the core performing the load balancing will then be about two cache misses, or 60-400 ns.[4] For comparison, we measured that Caladan [26] takes about 120 ns on average to check via work stealing if another core has stealable work (in the form of queued packets, threads, or timers).

Note that a single core can typically have up to about 10 cache misses outstanding at once [24] (we confirmed through a microbenchmark [48] that this seems to be about 10-12 for our Intel Skylake servers). This enables small numbers of independent cache misses (such as those to sample the load on two different cores) to incur in parallel.

Core-allocation overheads. The latency for a core allocation to complete varies depending on the mechanism used to reallocate the core. At a bare minimum, reallocating a core requires an inter-processor interrupt (IPI) from the core that makes the reallocation decision to the core that will be reallocated to a different application; this takes about 1993 cycles or roughly 1 μs [36]. Existing systems report slightly higher core-allocation latencies, varying from 2.2 μs to reallocate an idle core or 7.4 μs to reallocate a busy core in Shenango [61] to 29 μs to reallocate a core in Arachne [67].

4 Factor Analysis

In this section, we perform a factor analysis to determine the relative performance of the load-balancing and core-allocation policies defined in §3.2. We cannot effectively compare different policies by comparing existing systems that implement them (e.g., Caladan vs. Arachne), because these systems differ in many aspects besides their policies (threading libraries, network stacks, etc.). Even comparing different policies within a single implemented system can be challenging, because the optimal system design may vary depending on the policy. For example, systems may use different locking mechanisms to protect thread queues depending on whether only the local core can enqueue to them (as in work stealing) or if remote cores can also enqueue to them (as in enqueue choice). Thus, to decouple the behavior of the policies from the behavior of the systems that they are implemented in, we use simulations.

Our simulations rely on several parameters which define both the workload and assumptions about the possible underlying system. We find that our conclusions are quite robust to variations in these parameters, and therefore may be applicable to a wide variety of implementations and workloads. We have made the source code for our simulations available at `https://github.com/smcclure20/scheduling-policies-sim`.

4.1 Simulation Methodology

While our focus is on policy choices rather than implementation details, we do seek to model realistic overheads for cross-core communication and for allocating cores to applications. In order to fairly compare different policies, we use consistent values for each overhead, based on the overheads measured above (§3.3). We model the cross-core communication generally required for load balancing as taking 100 ns. We model the core-allocation overheads (both latency to allocate a core and wasted CPU cycles) as 5 μs per core allocation. In §4.2.1, we will consider some different values for load-balancing overheads, though varying them by even 100% does not have a profound impact on our results. We discuss the implications of varying core-allocation overheads in §4.3.

Our overall model assumes that each core has a single local queue (i.e., no distinction between packet and thread queues) and that tasks arrive randomly at the queues of allocated cores. This is representative of a NIC randomly steering tasks to cores or of running threads randomly spawning an additional thread. Our simulator models each of the general policy approaches outlined in §3.2, with specific implementation choices made based on real system implementations whenever possible. We acknowledge that our model is a simplified view of these systems, but we found that the general trends of latency and efficiency are consistent between simulations and experiments, for the systems we evaluated (§6). Our simulator does not support preemption but could be extended to model systems which do [20, 36, 82]. We now describe the specific load-balancing and core-allocation policies that we simulate.

Load-balancing policies. We model no overheads for the idealized *single-queue* policy or for the *no load balancing* policy. For *enqueue choice*, when a task arrives, the core at which it arrives incurs the 100 ns overhead to move the task to its destination queue (the shortest queue from two randomly

[4] This is an approximation; the exact overhead will depend on application behavior.

sampled options).[5] When *work stealing* is enabled and a core does not have any work in its local queue, it begins iterating through the other cores, checking if there is available work to steal. Each check of a remote queue incurs the 100 ns overhead, as does the act of stealing any found tasks. With *work shedding*, each core checks if its queue's current queueing delay is higher than the configured threshold after each task it finishes. If so, it selects a random core to notify or "flag." The remote core will check for flags between each of its tasks, respond (if a flag is present) by stealing tasks from the overloaded queue so that the two queue lengths are balanced, and incur the 100 ns overhead.

Core-allocation policies. Our *per-task* policy (based on Fred [40]) immediately grants a new core to an application if one is available in the system whenever a new task arrives. The core at which the task is initially randomly placed pays a 100 ns overhead to place the task at the new core. When a core finishes a task, it checks if there are more queued tasks in the system than available cores and yields if there are not.

The remaining core-allocation policies make decisions at fixed time intervals. To model *Shenango* [60] and *Caladan* [26], at the end of every core-allocation interval, the simulation determines the maximum queueing delay across cores within an application. If it exceeds a specified threshold (typically the length of the interval itself), the simulation grants an additional core to that application. An application yields a core if the core attempts to work steal from every other core in the application and fails to find any tasks to steal. Shenango and Caladan have very similar policies; the main distinguishing factor in our model is the difference in their interval/threshold values (Table 2).

We also design and simulate two new core-allocation policies. First, we design a queueing-based policy called *delay range* which attempts to maintain a specified average queueing delay across all cores within an application. Every core-allocation interval (every 5 μs), the simulation checks the average queueing delay. If it is below the specified lower bound, a core is revoked; if it is above the upper bound, a core is added. Similarly, with our *utilization range* policy, a core is added or removed whenever the average CPU utilization over the past interval (fraction of time spent handling tasks) falls outside the specified range.

There are three notable aspects of core-allocation systems that we do not model. First, some systems dedicate a scheduler core to making core-allocation decisions and initiating core allocations [26, 60, 67] while others have application cores perform these tasks in a distributed way [40, 76]. We do not model these distinctions and assume that all work for initiating core reallocations could be offloaded to a separate dedicated core. Second, we do not model the overheads incurred by applications measuring and exposing statistics to the dedicated core; in practice these overheads are small and simply require application cores to write a small amount of state (e.g., timestamp when a task was queued) to shared memory. Third, we do not model the caching implications of reassigning a core from one application to another.

[5] We assume the options may be checked in parallel as explained in §3.3.

Parameter	Default Value
Work shedding delay threshold	2 μs
Enqueue choices	2
Utilization range	75-95%
Delay range	0.5-1 μs
Shenango max queueing threshold	5 μs
Caladan max queueing threshold	10 μs

Table 2: Canonical configuration parameters.

Configuration. Each policy has its own unique parameters. Unless stated otherwise, we use the default parameter values shown in Table 2. We chose these specific values based on the best overall performance seen for each policy, though we will discuss the implications of configurability throughout this section.

In all of our simulations, we use a canonical configuration of 32 cores, exponentially-distributed service times with an average of 1 μs, Poisson arrivals, and an offered load that occupies 50% of the cores on average. Experiments below will vary many of these dimensions independently, but we will use this configuration by default. To contextualize the policy overheads described above, with the average task time set at 1 μs, the overhead for load balancing is 10% of average task time while the overhead of core reallocation is 500%.

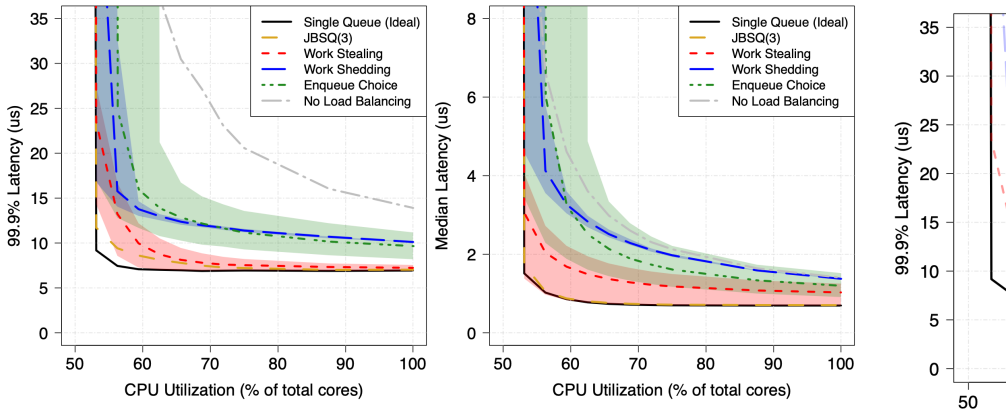
4.2 Load Balancing

To understand how load-balancing policies impact performance, we first evaluate different load-balancing policies in a setting where cores are statically allocated (cores are never reallocated) (§4.2.1), and then evaluate whether core reallocations impact these findings (§4.2.2).

4.2.1 With Static Core Allocations

Individual policies. We first evaluate each load-balancing policy independent of any particular core-allocation policy by running each experiment with a fixed number of cores. This allows us to determine the relative performance of each approach when given the same number of total CPU cycles, since a given allocation policy will make different allocation decisions depending on the behavior of the specific load-balancing scheme, even under the same traffic. By decoupling the two, we can determine which end-to-end effects are due specifically to load-balancing policies.

Figure 3 shows the tail and median latencies (y-axis) of different load-balancing policies as we vary the number of statically-allocated cores (shown on the x-axis as a fraction of the total possible), while offering an average load of 50%. Each curve corresponds to a load-balancing policy with 100 ns overheads, while the shaded regions vary this from 0 ns to 200 ns. In general, approaches that operate lower and to the left in this graph are preferable. We will discuss the JBSQ



(a) 99.9% latency. **(b)** Median latency.

Figure 3: Performance of each load-balancing policy with different numbers of statically allocated cores. Shaded regions cover overheads 0-20% of average task time. The line in each region shows the canonical case of 10% load-balancing overheads (100 ns).

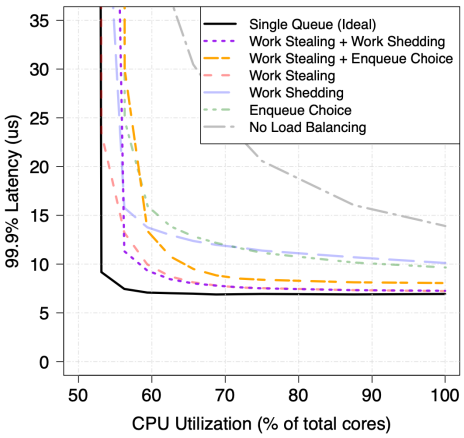


Figure 4: Latency curves for combinations of work stealing with other load-balancing policies, with static core allocations.

curve later.

Finding 1: *With static core allocations, work stealing achieves better latency (at the median and tail) for a given efficiency (number of allocated cores) than work shedding or enqueue choice.*

While all load-balancing policies yield significant improvements over no load balancing, work stealing consistently has significantly lower median and tail latency for the same number of statically-allocated cores than the other approaches; work stealing Pareto-dominates enqueue choice and work shedding. The relative performance between enqueue choice and work shedding is less consistent and varies depending on system and workload parameters such as the number of allocated cores, latency percentile, and service time distribution (Appendix A.2.2).

The enqueue choice curve is consistent with the well-known "power-of-two choices" result [55], showing that two choices of queues is much better than one (the "No Load Balancing" curve). This is particularly true when there is no overhead (as modeled in [55]) which is demonstrated by the lower bound of enqueue choice's shaded region in Figure 3. Despite this, enqueue choice still performs worse than work stealing. Further measurements revealed that this is due to three main limitations: (1) per-task load-balancing overheads that cap the possible throughput and add latency to all tasks, (2) a limited number of queue choices, and (3) placement based on number of queued tasks rather than the sum of service times of queued tasks. Overall, (2) and (3) can result in periods of load imbalance in which tasks are queued and cores are idle, but there is no way for the idle cores to assist with those "stranded" tasks. Choosing by the sum of the service times in the queue [30, 31] or increasing the number of choices can improve tail latency, though these are not always practical, and reducing the overheads to 0 provided a bigger performance benefit than either of those changes individually.

The tail latency gap between work stealing and work shedding can be explained by the steps necessary to move a task that ends up contributing to tail latency to the core that ultimately handles it. With work shedding, for a task at an overloaded core, time is spent waiting to cross the signalling threshold, waiting for the core to complete its current task and raise a flag, and waiting for the remote core to respond. In work stealing, tasks simply wait until a work-stealing core checks their queue; the latency of this depends primarily on the number of excess cores. With a work-shedding queueing threshold of 2 μs we found that on average tasks that were shed spent 3.1-4.5 μs queued on cores other than the one that ultimately handled the task, compared to 0.3-1.4 μs with work stealing. Most tasks at the tail are stolen at least once, explaining the corresponding gap between the two in tail latency. Lowering the queueing threshold only yields marginal improvements, because at higher loads most cores will always have a flag pending. In addition, without preemption, tasks still incur delays from the other two steps described above.

We now investigate the robustness of these results to changes in overheads in case our overhead estimates are not representative of certain underlying hardware or better technology arises in the future. Note that this does not apply to single queue simulations or those with no load balancing as they have no overheads. By looking at the upper or lower bounds of the shaded regions in Figure 3, we see that for the same overhead, work stealing consistently outperforms the other approaches. Even if inter-core communication was free for enqueue choice and work shedding, work stealing with 200 ns overheads outperforms for most numbers of allocated cores. Further, work stealing consistently achieves the best performance even if we model the load-balancing overhead as 400 ns, the upper bound of our estimate from §3.3.

Work stealing's superior performance is robust across different latency percentiles (median to 99.9%) (Figure 3), average service times (*e.g.,* 1, 10, 100 μs) (Figure 5), numbers of cores (Figure 6), loads (Appendiex A.2.1), and ser-

vice time distributions (exponential, constant, bimodal) (Appendix A.2.2). For all service time distributions evaluated, the ordering of the static curves remained the same as in exponential distributions shown. Though, when service times are constant, the specific choice of load-balancing policy has less overall impact on system performance.

Combining policies. Notably, these load-balancing approaches are not mutually exclusive. Since each policy takes effect at a different time in the handling of a task, work stealing can take advantage of extra cycles while work shedding addresses excessively loaded cores or enqueue choice proactively tries to balance queues. Accordingly, we simulated work stealing combined with each other approach with static core allocations.

Finding 2: *With static core allocations, adding shedding on top of work stealing provides some latency benefit (primarily at the tail) while adding enqueue choice to work stealing makes performance unchanged or worse.*

This is demonstrated in Figure 4. We see that adding enqueue choice to a system that already employs work stealing does not improve performance. There are two reasons for this, depending on what efficiency (x-axis) we are operating at: (1) with few cores available, enqueue choice adds significant overhead per-task which degrades throughput, and (2) with many cores available, there is little room for improvement between work stealing and single queue. When adding work shedding to work stealing, however, there are some benefits since the shedding mechanism can help balance out queues under high-load conditions when work stealing lacks the extra cycles to help, though the benefit is fairly limited to certain efficiencies as the overheads of flagging can become excessive when spare cycles are rare.

Leveraging hardware. Given these results, we ask two questions motivated by recent advances in hardware: (1) what if the NIC can perform more intelligent distribution than simple hashing? and (2) what impact would handling many cache misses in parallel have? (1) is motivated by recently proposed systems such as the NanoPU [34] which selects queues for incoming packets according to join bounded shortest queue (JBSQ) [43].[6] JBSQ is known to achieve good performance with tail latency improvements up to 10 μs over work stealing, as shown in Figure 3. However, this boost requires new hardware to direct incoming traffic intelligently.

To address (2), we simulated scenarios where the underlying hardware could resolve several cache misses at once (as described in §3.3). Ultimately, this capability means that load-balancing policies may communicate with multiple cores for the price of one (e.g., check 10 cores for the presence of work in work stealing). However, we found that these modifications provided marginal benefits at best, even assuming that processing the results of parallel checks incurs no overhead.

In general, work stealing was consistently the best performing load-balancing policy when given the same number of cycles as other approaches even as overheads and workload parameters vary. Broadly, work stealing achieves high performance by avoiding per-task overheads and leveraging idle cores to avoid stranding tasks at overloaded cores. Ultimately, absent new hardware, work stealing is the best option for load-balancing approaches among those we evaluated. While work stealing's superiority may seem unsurprising given its widespread use, we believe that we are the first to compare it against other policies and demonstrate its benefits when handling microsecond-scale tasks with realistic load-balancing overheads.

4.2.2 With Dynamic Core Allocations

Next, we consider how load-balancing policies perform when cores can also be reallocated: does reallocating cores change the findings above? When the number of cores allocated to a given application varies over time, it becomes harder to compare approaches (combinations of load-balancing and core-allocation policies). Each combination represents a single point in the tradeoff space between latency and efficiency. If one combination has better latency but worse efficiency than another (i.e., neither is Pareto dominant), which is preferable? Some core-allocation policies are configurable and could be tuned to operate at the same efficiency to compare their latencies. However, not all approaches are tunable (e.g., per-task allocations), so this methodology cannot be used to compare all policies. Thus it is not always possible to say that one policy combination is definitively better than another.

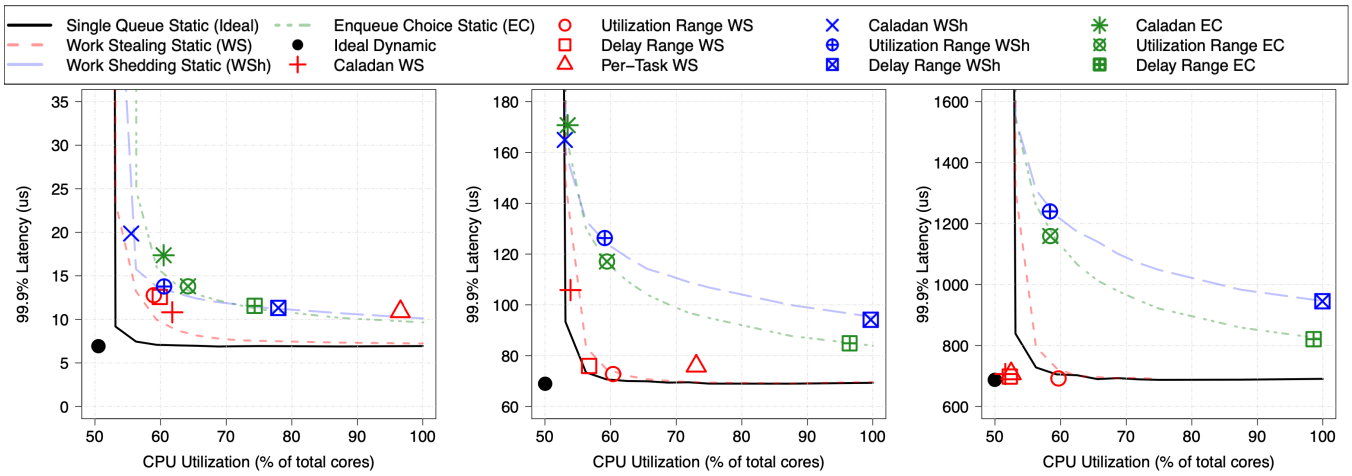
We attempt to pair each load-balancing policy with each other core-allocation policy, but some pairings require modifications or are not reasonable. In Shenango/Caladan, cores park upon failing to find any work to steal. We modify this to work with other load-balancing policies by revoking cores after they spin for the time it would take to check all cores in traditional work stealing, assuming no additional work arrives in the meantime. Per-task allocations maintain the invariant that the number of active cores is equal to the minimum of the number of tasks present and the total number of cores. This is only reasonable with a work-conserving load-balancing policy, so we only evaluate per-task core allocations with the work-stealing load-balancing policy.

With this in mind, we simulated all coherent combinations of load-balancing and core-allocation policies to compare how they explore the available tradeoff space. The results across different average task durations are shown in Figure 5.

Finding 3: *When cores are dynamically reallocated, work stealing performs better than shedding or enqueue choice. This is robust against all factors mentioned in Finding 1.*

Figure 5a shows each combination of load-balancing and core-allocation policies with static-allocation curves for ref-

[6] JBSQ(n) queues up to n outstanding tasks at each core (including the task currently being handled) and maintains any surplus in a central queue [43]. We evaluate the case of 3 outstanding tasks, as in NanoPU, though we label this as JBSQ(3) in the terminology of [43] rather than JBSQ(2) as in NanoPU.



(a) 1 μs average task service time. **(b)** 10 μs average task service time. **(c)** 100 μs average task service time.

Figure 5: Combinations of each core-allocation policy with a load-balancing policy at 50% load with static-allocation curves for each load-balancing policy for reference. Each policy uses the canonical configurations from §4.1. Points for each combination of policies are the color of their load-balancing curve and have the shape type (squares, circles, stars, or triangles) of their core-allocation scheme.

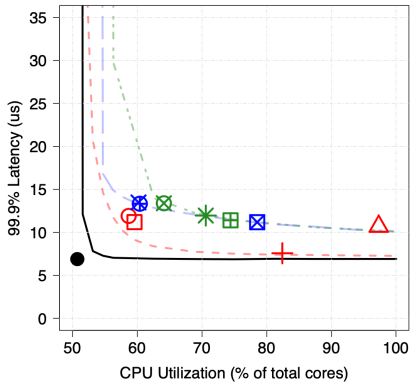


Figure 6: Core-allocation and load-balancing policy combinations for 64 cores and 1 μs tasks. Refer to the legend in Figure 5.

erence. Comparing each core-allocation policy (shape type) across load-balancing policies (colors), we see that work stealing always performs best in terms of proximity to the single queue curve. Note that this graph only shows one choice of parameters for each core-allocation policy, but some can be configured to make different latency vs. efficiency tradeoffs. We generally chose the configuration closest to the bottom-left of the graph, though we will discuss configurability broadly in §4.3.

While adding dynamic core allocations makes the individual performance of each load-balancing policy less clear, overall work stealing still consistently performs better than other load-balancing approaches (absent new hardware).

4.3 Core Allocation

In this section, we compare the performance of different core-allocation policies. Since core-allocation policies are designed to react to changes in load, their performance tends to be tightly coupled with the load-balancing policy employed. Better load-balancing policies will more effectively use the available cycles, allowing the core-allocation policy to be more conservative in granting cores. Therefore, we evaluate each core-allocation policy across each load-balancing policy and seek to find patterns in the tradeoffs between efficiency and latency that each core-allocation policy makes.

We note that some existing systems use an additional dedicated core (such as Shenango's IOKernel) to perform core allocations [26, 60, 67]. We do not count these cores as we are focusing on policy rather than the implementation of that policy. If we were to include these cores, all efficiency results for these policies would incur an additional 3% CPU utilization for a 32-core system.

We began by asking the question: does reallocating cores yield better performance than sticking with a constant number of cores? One might expect that even with constant average load, being able to react to bursts in load over small time scales would yield significant performance benefits. Surprisingly, we found that the answer to this question is often 'no'.

Finding 4: *For short tasks, none of the core-allocation policies we tried achieved better latency (median or tail) for a given average efficiency than static core allocations (with the same load-balancing policy). However, this becomes possible with longer tasks.*

In Figure 5a, none of the core-allocation policies achieve better tail latency for the same efficiency as a static allocation (the points fall up and to the right of their corresponding static core-allocation curves). As shown in Figures 5b and 5c, when the average task service time is longer (*e.g.,* 10 μs or 100 μs), some policy combinations (points) can achieve better performance than their static-allocation curves. With work stealing and 10 μs service times, delay range, utilization range, and Caladan all beat the static curve for 99.9% tail latency, but

not for the median (omitted for space). This is true for 100 μs service times as well, with per-task also beating the work-stealing curve. As the average task duration increases, the relative importance of the core-allocation overhead decreases and allocating new cores for additional tasks becomes reasonably efficient. The only policy combinations which beat their respective curve include work stealing as the load-balancing policy. Work stealing leverages extra cycles to distribute load while enqueue choice and work shedding are limited in impact since newly added cores will spin idly, unable to handle tasks until a new task arrives or they are flagged by another core.

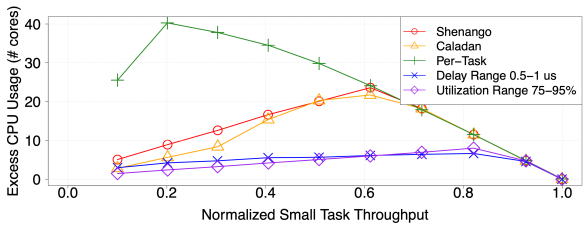
The only method we have found that can outperform the static curve with tasks as small as 1 μs requires the core-allocation system to be extremely reactive, making core-allocation decisions more frequently than 5 μs and giving the new cores to the application faster than in 5 μs. More frequent allocations are challenging in a real-world implementation because of the overheads of checking state and initiating core reallocations. For example, in Shenango, these actions take roughly 2.1 μs or 3.4 μs with 32 or 64 application cores, respectively [61]. Completing each core reallocation in less than a few microseconds is similarly challenging (§3.3).

Even though core allocations may not provide performance benefits with short tasks, one may employ a core-allocation policy to ensure that the application can adapt to changes in load. Average load in datacenters tends to vary over time [11], so allocating a static number of cores for a constant load would require provisioning for the peak load, wasting CPU cycles over time as load varies. Reacting more slowly to changes in load is also unlikely to perform well; prior work has shown that reactions at 50 ms timescales can cause significant tail latency spikes [60].

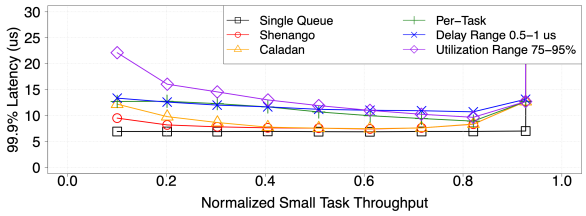
Assuming that achieving better performance than the static-allocation curves is unlikely for small tasks, we evaluate the different core-allocation policies in terms of the consistency of their performance and their ability to achieve high CPU efficiency. For some core-allocation policies, the placement relative to a static curve can vary significantly depending on the workload and load, making it difficult for an operator to configure the policy to achieve their goals (e.g., a specific tail latency or CPU efficiency target). By comparing the tradeoffs core-allocation policies make across workloads and loads, we find the policies that exhibit consistent performance.

Finding 5: *Policies that explicitly optimize for an end-to-end user-visible metric (e.g., delay range and utilization range) have more consistent performance, as measured by those metrics, across different configurations.*

For example, Figure 5 illustrates that for Caladan and per-task, the operating point changes with different service times. In contrast, utilization range and delay range specify a range on the x and y axes of the graphs, respectively, that the system should not leave. This generally forces the points to specific



(a) Efficiency measured in excess cores.



(b) 99.9% latency.

Figure 7: Performance of core-allocation policies paired with work stealing across loads for 64 cores. Efficiency is measured in the excess cores used compared to the single queue simulation.

regions of their static-allocation curves (when the curves cannot be crossed). For example, utilization range points achieve close to 60% CPU utilization across all service times in Figure 5.

Delay range and utilization range also have more predicatable performance across different loads. In Figure 7, we illustrate how performance of different core-allocation policies varies with load (when paired with work stealing). We use 64 cores instead of 32 in order to sweep a wider range of loads. Figure 7a shows the efficiency measured as excess cores in comparison to the single queue ideal simulation (i.e., total number of cores used by a given policy minus those used in the ideal case) while Figure 7b shows the tail latency. Caladan, Shenango, and per-task have inconsistent efficiency and tail latency across loads, while delay range and utilization range each keep their respective end-to-end metric relatively constant. Overall, we found that policies such as delay range and utilization range have consistent performance across workloads and configurations, enabling the operator to directly tune the policy's parameters to achieve a specific end-to-end performance objective.

Next, we consider whether each core-allocation policy can be configured to operate near the bend of each static-allocation curve, achieving high CPU efficiency while only minimally compromising in tail latency.

Finding 6: *Yielding cores only when no work is found (when there is no queued work or work stealing fails) makes it challenging to achieve good efficiency with small tasks, especially with many cores.*

The policies that yield cores only when no work is found (Caladan, Shenango, and per-task) cannot always achieve good CPU efficiency, especially with many cores. Here we focus on analyzing each core-allocation policy when paired

with work-stealing, as it performs best. Figure 5a illustrates that per-task achieves poor CPU efficiency with 32 cores, while Figure 6 shows that per-task and Caladan both achieve poor CPU efficiency with 64 cores, using more than 80% of CPU cores for a workload that only requires 50% of cores. In contrast, delay range and utilization range both operate near the bend of the static-allocation curves for 32 and 64 cores. Figure 7 illustrates that utilization range and delay range can save up to 15 cores for similar tail latency across loads.

The efficiency of the Shenango, Caladan, and per-task policies is limited because these policies are slow to yield excess cores. With per-task core allocations, before all cores are allocated the efficiency cannot reach higher than $T/(T+R)$ where T is the average task time and R is the core-allocation overhead, because a core is allocated for every task. Similarly, in policies that yield cores only when work stealing fails (Shenango and Caladan), a significant amount of cycles can be wasted searching through all other cores to never find work or only to find it late in the search. As the number of cores increases, this effect gets worse. Neither Shenango/Caladan nor per-task can be configured to avoid these inefficiencies. Therefore, to achieve high efficiency across workloads and configurations, a core-allocation policy must revoke cores proactively, even when there is or may be some queued work.

We did assess other core-allocation policies such as maintaining a buffer of idle (or work-stealing) cores of a certain size (similar to PerfISO [35]) and enforcing this buffer at every allocation interval. However, this approach tended to be too noisy with short core-allocation intervals and performed significantly worse than other policies.

All together, we found that it is difficult to outperform static core allocations with small tasks, and if the average load is constant and known *a priori*, then statically allocating cores is the best option. However, when load is unknown or changes over time, dynamic allocation policies that proactively revoke cores perform best.

4.4 Policy Takeaways

Overall, our factor analysis found that without new hardware, the best approach is to use work stealing as the load-balancing policy with delay range or utilization range for core allocations, depending on which end-to-end metric is more important to specify and stabilize. Both of these policies are able to operate close to the work-stealing static curve with short tasks or better than the curve with long tasks. Both are robust in the face of service time variability, different service time distributions, load changes, and changes in number of cores. Lastly, both are configurable, allowing the operator to choose whether they prefer CPU efficiency or tail latency (and to what extent). These approaches are intuitive; since core-allocation policies make a tradeoff between CPU efficiency and tail latency, using either parameter effectively as a signal for reallocating cores and controlling where to operate in the space of tradeoffs makes sense.

5 Implementation

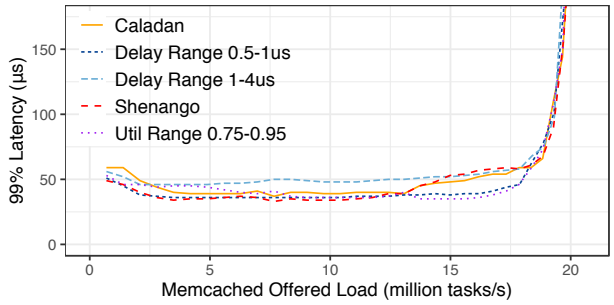
We implement our policies in a real system by extending Caladan [75]; our source code is available at https://github.com/shenango/caladan-policies. Like its predecessor Shenango [60], Caladan's key components are its application runtime and its dedicated scheduler core, which implements the core-allocation policy. Caladan provides lightweight user-level threading, a high-performance network stack, and load balancing via work stealing. It also enables higher network throughput and its core-allocation mechanisms are more scalable compared to those of Shenango.

We implement both delay range and utilization range atop Caladan. This requires small modifications to both the runtime (50 LOC) and to the scheduler (125 LOC). The Caladan runtime already exposes information about the queueing delay of threads and packets to the scheduler core; we augment this with information about CPU utilization (time spent executing the application vs. in the runtime scheduler) as well. We also add the ability for application cores to yield voluntarily when notified by the scheduler core to do so. When application cores enter the runtime scheduler between tasks, they check if they should yield; for efficiency we do not preempt cores while they are handling tasks. In the scheduler core, we simply add logic for polling the utilization information exposed by applications, and use this or the delay information (depending on the current policy) to decide whether to add or revoke cores. When a core revocation is necessary, the scheduler revokes the core that currently has the least amount of queued work.

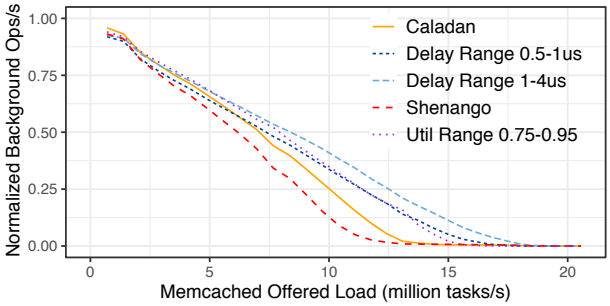
Measuring the CPU utilization of application cores over fine timescales is more challenging in practice than in simulation. This is because we do not interrupt running tasks to record CPU usage and only record how CPU time is spent whenever a task starts or finishes. Thus if a task runs for the entirety of a 5 μs core-allocation interval, the scheduler core will observe 0 cycles spent in both the application and the runtime scheduler for that core. The scheduler core handles this by assuming that when application cores report no CPU usage for a core-allocation interval, their utilization is 100%, and it adds a core. In the case when an application has zero allocated CPU cores, CPU utilization is not a useful metric for deciding if an application needs more cores. Thus regardless of the core-allocation policy, the scheduler core always uses the arrival of packets to decide when to grant an application its first core, as in Shenango and Caladan.

6 Evaluation

The goal of our evaluation is to verify that the high-performing policies we identified above can actually yield performance improvements in practice for a real system. Unfortunately, varying the load-balancing policy within a single system would likely involve significant system changes, making a fair comparison difficult (§4). Thus we focus on evaluating the core-allocation policies. We start with a system (Caladan)



(a) Tail latency for memcached.



(b) Normalized throughput of the background application.

Figure 8: Performance of two applications under different core-allocation policies, when implemented atop Caladan. The x-axis varies the load of memcached.

that uses the best-performing load-balancing policy (work stealing) and evaluate its performance with different core-allocation policies. We evaluate four policies: Shenango [60], Caladan [26], delay range, and utilization range.

Experimental setup. We conduct experiments using two dual-socket servers with 28-core Intel Xeon Platinum 8176 CPUs operating at 2.1 GHz. Our server machine is equipped with a 40 Gbits/s Mellanox Connect X-5 Bluefield NIC (we do not use the SmartNIC features) and our client machine is equipped with an Intel E810C 100 Gbits/s NIC.[7] We enable hyperthreads and disable TurboBoost and frequency scaling. We use 32 hyperthreads on the second socket (to which our NICs are attached). We use Ubuntu 20.04 with kernel version 5.4.0.

Applications. We evaluate the different policies using *memcached* (v1.5.6) [49], a popular key-value store, as our latency-sensitive application. We use *loadgen*, Caladan's open-loop load generator, to generate requests with Poisson arrivals over UDP [75]. Our workload consists of a mixture of read and write requests according to Facebook's `USR` request distribution [8]; requests have service times of about 1 μs. We run the *swaptions* workload from the PARSEC benchmark suite [13] as a background application and allow it to use all CPU cycles not used by memcached.

[7] We run Caladan in "queue steering mode" in which we reconfigure the mappings between NIC queues and cores when core allocations change [61] because our NICs do not support Caladan's default "flow steering mode."

6.1 Policy Comparisons

Our experimental results show that different policies yield different latency vs. CPU efficiency tradeoffs, but that delay range and utilization range generally outperform Shenango and Caladan, confirming the findings of our simulation-based factor analysis. Figure 8a shows the tail latency of memcached while Figure 8b shows the throughput achieved by the background application, both as we vary the load offered to memcached (x-axis). We show results for two different configurations of delay range to illustrate the impact of tuning the target range.

In Figure 8, utilization range and delay range (0.5-1 μs) achieve similar tail latency for memcached as Caladan and Shenango, while achieving higher CPU efficiency for the background application. In addition, all of these policies yield similar median latency for memcached (not shown). Shenango is least efficient overall, and these two new policies achieve up to 22% more of the total possible throughput for the background application (7 hyperthreads worth) than Shenango. Compared to Caladan, these policies achieve up to 13% more throughput for the background application (4 hyperthreads worth). This is because with Shenango and Caladan's policies, memcached spends much more time in the runtime scheduler, primarily work stealing (up to 26% and 21% of its time, respectively). In contrast, with the other policies, CPU time in the scheduler is much lower. For example, with utilization range, memcached spends less than 14% of its time in the scheduler at all except the lowest loads. By proactively revoking unused cores rather than waiting for work stealing to fail to find tasks to handle, delay range and utilization range can achieve higher CPU efficiency without degrading the performance for memcached.

Both the delay range and utilization range policies take as input a target range, and these ranges can be adjusted to make different tradeoffs between tail latency and CPU efficiency. As an example, Figure 8 shows two different ranges for delay range. Delay range 1-4 μs achieves about 2 hyperthreads worth of additional throughput for the batch application compared to delay range 0.5-1 μs, at the cost of 10-15 μs of tail latency.

7 Related Work

Load-balancing policies. Load-balancing policies have been studied extensively, both theoretically and in the context of real systems. Several systems have adopted the ideal policy of maintaining a single shared queue [36, 62], though they experience throughput bottlenecks as a result. Others take the opposite approach and perform no load balancing in software, leaving it to the NIC [12, 64] or storage device [42] to randomly distribute work across cores; these approaches suffer from load imbalances.

Work stealing was originally proposed as a way of efficiently scheduling multithreaded computations across multi-

ple cores [14,39,71]. Variants of work stealing have been studied thoroughly [54, 57] and adopted in task-parallel platforms such as OpenMP [59], IntelTBB [68], Cilk [47], Habanero [9, 16], X10 [17], Java Fork/Join [45], and the Go runtime [76]. More recently, work stealing has been adopted in datacenter systems as a way to provide low tail latency [26,44,60,66,81]. Similarly, past work has analyzed the power-of-two choices load-balancing policy [55] as well as variants of it, such as those that consider known service times [31] or general service time distributions [15]. Arachne [67], SKQ [81], and many other systems [29, 33, 63, 82] leverage power-of-two or the more general power-of-k choices for load balancing. Others have studied work shedding approaches [73] and compared them to other policies [22, 77]. Finally, several recent proposals implement more advanced load-balancing policies such as JBSQ [43] in NIC hardware [18, 34, 70, 74].

Our findings are consistent with past comparisons of load-balancing policies. For example, we confirm that "work-first" load-balancing policies such as work stealing have better performance [21, 22, 27]. However, our analysis differs in two key ways. First, we are not aware of any prior work that compares load-balancing policies in the presence of realistic load-balancing overheads; prior work either assumes no overhead or analyzes a single system and its policy and overheads. Second, prior work evaluates metrics such as delay, throughput, and communication rate, but does not consider CPU efficiency. In contrast, we compare the tradeoffs that different policies make in terms of latency and efficiency, in the presence of load-balancing overheads.

Core-allocation policies. Existing systems adopt a variety of different policies for deciding when to reallocate cores, either across different applications or between cores available for applications and those designated for network processing or a file system. These approaches make decisions based on task arrivals [40], queueing delay [12, 20, 26, 52, 53, 60, 67], CPU utilization [12, 20, 35, 41, 67], or failure to find work [4, 7, 21, 27, 40, 76]. None of these systems compare different policies in the presence of the same overheads, so it is not possible to determine from these works which policies provide the best combination of latency and efficiency. Some past work points out that work-stealing cores can waste considerable CPU cycles, and proposes policies for yielding cores to mitigate this [4, 7, 21]. However, these policies target throughput and fairness for longer tasks (e.g., hundreds of microseconds or more); in contrast, our analysis focuses on which policies provide the best efficiency and latency for microsecond-scale tasks, and thus yields different conclusions.

Implementing policies. The systems Syrup [37] and ghOSt [32] enable users to control scheduling policies in the kernel scheduler, network stack, and network card from code written in userspace. These systems are complementary to our work; they make it easier to express scheduling policies but do not specify which policies users should implement.

8 Conclusion

Numerous systems have been designed to support latency-sensitive datacenter applications while dynamically allocating cores to react to changes in load. However, these systems often come with a significant efficiency penalty with short tasks. In this paper, we systematically evaluated the effects of different policy choices on efficiency and latency to determine which load-balancing and core-allocating schemes achieve the best performance when considering realistic overheads. Work stealing is the definitive best policy option in today's hardware while the core-allocation space is more complex. We designed and implemented two core-allocation policies which provide consistent and configurable performance on the Pareto frontier when paired with work stealing and demonstrated how they can improve efficiency without sacrificing latency.

9 Acknowledgments

We thank our shepherd Ana Klimovic, the anonymous reviewers, John Ousterhout, and the members of NetSys for their useful feedback. We thank Daniel Grier for assistance with the NP-hardness proof. This work was funded in part by NSF Grants 1817116 and 1704941, and by grants from Intel, VMware, Ericsson, Futurewei, and Cisco.

References

[1] Dpdk. https://www.dpdk.org/.

[2] redis. https://redis.io/.

[3] Rss. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[4] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–32, 2008.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[6] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[7] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001.

[8] 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, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[9] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, et al. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736, 2009.

[10] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[11] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[12] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems (TOCS)*, 34(4):1–39, 2016.

[13] C. Bienia. *Benchmarking modern multiprocessors*. Princeton University, 2011.

[14] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[15] M. Bramson, Y. Lu, and B. Prabhakar. Randomized load balancing with general service time distributions. *ACM SIGMETRICS performance evaluation review*, 38(1):275–286, 2010.

[16] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61, 2011.

[17] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.

[18] A. Daglis, M. Sutherland, and B. Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[19] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[20] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 621–637, 2021.

[21] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. Bws: balanced work stealing for time-sharing multicores. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 365–378, 2012.

[22] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance evaluation*, 6(1):53–68, 1986.

[23] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[24] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang. Measuring microarchitectural details of multi-and many-core memory systems through microbenchmarking. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, 2015.

[25] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, 2017.

[26] 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)*, pages 281–297, 2020.

[27] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[28] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[29] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[30] T. Hellemans, T. Bodas, and B. Van Houdt. Performance analysis of workload dependent load balancing policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–35, 2019.

[31] T. Hellemans and B. Van Houdt. On the power-of-d-choices with least loaded server selection. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–22, 2018.

[32] 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)*, pages 588–604, 2021.

[33] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting linux storage stack for μs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128. USENIX Association, July 2021.

[34] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256, 2021.

[35] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, 2018.

[36] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[37] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 605–620, 2021.

[38] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[39] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM (JACM)*, 40(3):765–789, 1993.

[40] M. Karsten and S. Barghi. User-level threading: Have your cake and eat it too. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–30, 2020.

[41] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[42] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash $\approx$ local flash. *ACM SIGARCH Computer Architecture News*, 45(1):345–359, 2017.

[43] 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)*, pages 863–880, 2019.

[44] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, 2018.

[45] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

[46] C. Lee and J. Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 149–154, 2019.

[47] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[48] D. Lemire. Code used on daniel lemire's blog. https://github.com/lemire/Code-used-on-Daniel-Lemire-s-blog/tree/master/2019/01/01.

[49] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[50] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.

[51] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[52] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 819–835, 2021.

[53] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, 2019.

[54] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 212–221, 1998.

[55] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[56] M. Nandagopal, K. Gokulnath, and V. R. Uthariaraj. Sender initiated decentralized dynamic load balancing for multi cluster computational grid environment. In *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, pages 1–4. 2010.

[57] D. Neill and A. Wierman. On the benefits of work stealing in shared-memory multiprocessors. *Department of Computer Science, Carnegie Mellon University, Tech. Rep*, 2009.

[58] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[59] OpenMP. Openmp application programming interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, 2018.

[60] 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)*, pages 361–378, 2019.

[61] A. E. Ousterhout. *Achieving high CPU efficiency and low tail latency in datacenters*. PhD thesis, Massachusetts Institute of Technology, 2019.

[62] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[63] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[64] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, 2014.

[65] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the rpc tax in datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2021.

[66] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[67] 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)*, pages 145–160, 2018.

[68] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. 2007.

[69] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[70] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun. Elastic RSS: Co-scheduling packets and cores using programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 71–77, 2019.

[71] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 237–245, 1991.

[72] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. https://arxiv.org/pdf/2010.09852.pdf.

[73] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[74] 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)*, pages 199–212. IEEE, 2020.

[75] The Caladan Authors. Caladan's open-source release. https://github.com/shenango/caladan.

[76] The Go Community. The go programming language. https://golang.org.

[77] B. Van Houdt. Randomized work stealing versus sharing in large-scale systems with non-exponential job sizes. *IEEE/ACM Transactions on Networking*, 27(5):2137–2149, 2019.

[78] R. V. Van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, 2001.

[79] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[80] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391, 2013.

[81] S. Zhao, H. Gu, and A. J. Mashtizadeh. Skq: Event scheduling for optimizing tail latency in a traditional os kernel. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 759–772, 2021.

[82] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. RackSched: A Microsecond-Scale scheduler for Rack-Scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, Nov. 2020.

A Appendix

A.1 Proof of NP-Hardness for Optimal Core Allocations

In this appendix, we prove that finding the optimal tail latency for a given CPU usage bound or vice versa is NP-hard, assuming a finite number of cores and non-constant service times. We show this using a reduction from the multiprocessor scheduling decision problem.

Multiprocessor Scheduling Problem [28]

Input: A non-zero number of cores c, a set of tasks T where each task t has a positive integer service time (or length) $l(t)$, and an overall deadline D for completing all tasks.

Question: Is there a schedule of the tasks T over the c cores that meets the overall deadline D? Such a schedule assigns a start time to each task t such that there are never more than c tasks being handled simultaneously and for each task, its start time plus $l(t)$ is at most D.

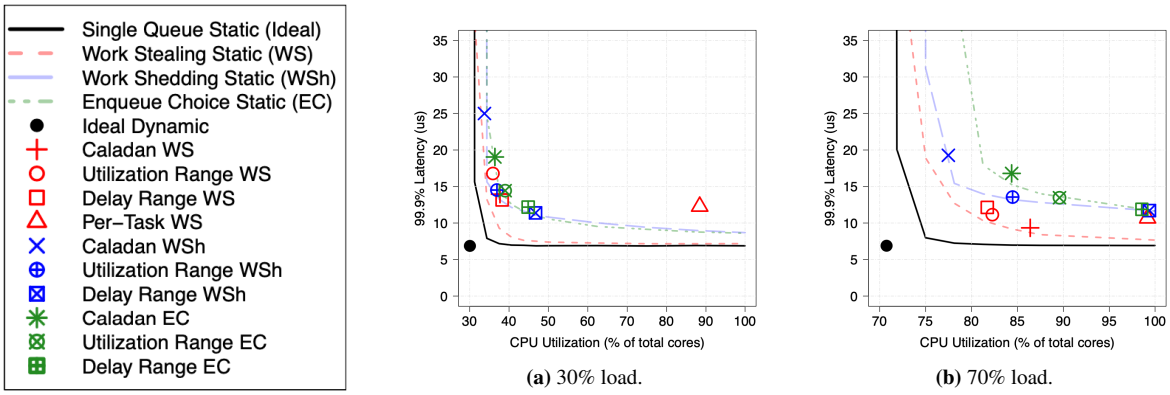
The Multiprocessor Scheduling Problem is NP-complete, assuming that all tasks do not have the same service time; with constant service times, this problem is trivial [28].

Optimal Core-Allocation Problem

Input: A non-zero number of cores c where each core can be either *on* or *off*, and transitioning from off to on requires a start-up time of S; a set of tasks T where each task t has an arrival time $a(t)$ and a positive integer service time $l(t)$; the total "wasted" CPU time W, or time spent by cores while they are starting up or on but not handling a task; a tail latency percentile $P < 1$ (e.g., 99.9^{th} percentile); and a tail latency target L.

Question: Is there a schedule for the c cores and the tasks T such that tasks are only scheduled on cores that are *on*, the wasted CPU time is at most W, and the latency at percentile P is at most L? A schedule for the cores assigns periods of *on* and *off* time to each, noting that it takes S time to transition from *off* to *on*. A schedule for the tasks assigns a start time to each task t such that the start time for t is at least $a(t)$ and the number of tasks being handled simultaneously never exceeds the number of cores that are *on*. Finally, for P percent of the tasks, their start time plus $l(t)$ is at most L.

We can reduce the multiprocessor scheduling problem to the optimal core allocation problem as follows. The number of cores in the core allocation problem matches that in the multiprocessor scheduling problem and we set $L = D$. We construct the set of tasks for the core allocation problem by replicating the tasks and their service times from the multiprocessor scheduling problem and setting them to all arrive at the beginning (i.e., $a(t) = 0$ for all $t \in T$). In addition, we add additional dummy tasks with $l(t) > D$ so that the tasks in the multiprocessor scheduling problem constitute P percent of the total tasks in the core allocation problem; because the dummy tasks cannot possibly meet the latency bound, the problem is only solvable by having all non-dummy tasks meet the latency bound. Finally we set the start-up time S to be zero and the



(a) 30% load. **(b)** 70% load.

Figure 9: Performance of core-allocation and load-balancing policies from Figure 5 at additional loads.

wasted CPU time bound W to be high enough to be irrelevant (e.g., $W \geq c \cdot D$). We leave it as a simple exercise to show that there exists a polynomial time algorithm for such an instance of the optimal core allocation problem if and only if there is a polynomial time algorithm for the corresponding instance of the multiprocessor scheduling problem. In addition, the optimal core allocation problem is clearly in NP; thus it is NP-complete.

Because the optimal core allocation decision problem is NP-complete, the optimization problem of finding the optimal tail latency for a given efficiency bound or vice versa is NP-hard. This proof assumes that the service time distribution $l(t)$ is not constant; the optimization problem with constant service times may also be NP-hard but this cannot be shown using the proof above.

A.2 Extended Factor Analysis

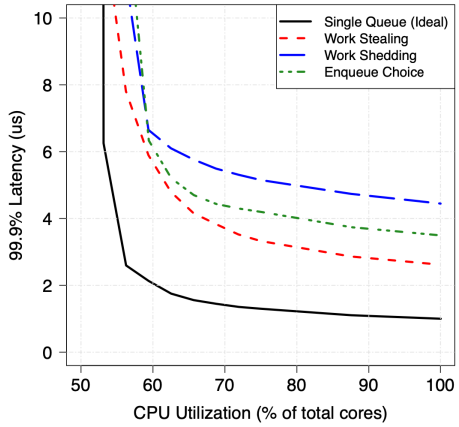
In this appendix we include additional data omitted for space in the factor analysis.

A.2.1 Additional Loads for Static Curves

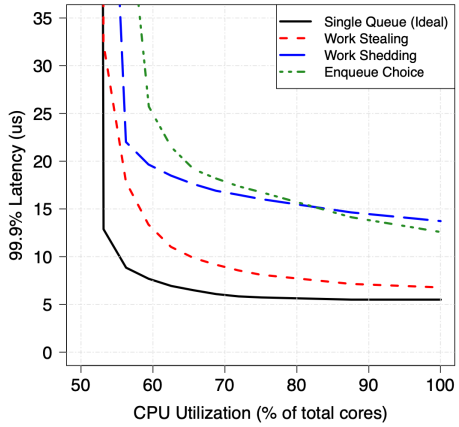
In Figure 5, we compared the performance of different load-balancing policies across different average service times to demonstrate that beating static allocations is more difficult with short tasks. The graphs look at both efficiency and latency simultaneously by keeping load constant. In Figure 9, we vary the offered load to 30% and 70%. To see a complete view of efficiency and latency (without static load-balancing curves for reference) across load, see Figure 7.

A.2.2 Additional Service Time Distributions

We compared the load-balancing policies across different service time distributions. Specifically, we created static allocation performance curves for each load-balancing policy for both constant service times of 1 μs and a bimodal distribution with 500 ns service times for 90% of requests and 5.5 μs for the remaining 10% (average service time of 1 μs). In Figure 10, we see that across these different service time distributions, work stealing consistently outperforms the other load-balancing policies. Since load-balancing choices are less significant to end-to-end performance when service times are constant (Figure 10a), work stealing provides smaller benefits.



(a) Constant service times (1 μs).



(b) Bimodal service time distribution.

Figure 10: Performance of load-balancing policies with static core allocations for different service time distributions.