ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks

George Prekas* EPFL, Switzerland

Marios Kogias* EPFL, Switzerland

Edouard Bugnion EPFL, Switzerland

ABSTRACT

This paper focuses on the efficient scheduling on multicore systems of very fine-grain networked tasks, which are the typical building block of online data-intensive applications. The explicit goal is to deliver high throughput (millions of remote procedure calls per second) for tail latency service-level objectives that are a small multiple of the task size.

We present ZYGOS, a system optimized for μs-scale, in-memory computing on multicore servers. It implements a work-conserving scheduler within a specialized operating system designed for high request rates and a large number of network connections. ZYGOS uses a combination of shared-memory data structures, multi-queue NICs, and inter-processor interrupts to rebalance work across cores.

For an aggressive service-level objective expressed at the 99th percentile, ZYGOS achieves 75% of the maximum possible load determined by a theoretical, zero-overhead model (centralized queueing with FCFS) for 10μs tasks, and 88% for 25μs tasks.

We evaluate ZYGOS with a networked version of Silo, a state-of-the-art in-memory transactional database, running TPC-C. For a service-level objective of 1000μs latency at the 99th percentile, ZYGOS can deliver a 1.63× speedup over Linux (because of its dataplane architecture) and a 1.26× speedup over IX, a state-of-the-art dataplane (because of its work-conserving scheduler).

KEYWORDS

Tail latency, Microsecond-scale computing

ACM Reference Format:
George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of SOSP '17*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3132747.3132780

*co-equal first author.

SOSP '17, October 28, 2017, Shanghai, China
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132780

1 INTRODUCTION

To meet service-level objectives (SLO), web-scale online data-intensive applications such as search, e-commerce, and social applications rely on the scale-out architectures of modern, warehouse-scale datacenters [2]. In such deployments, a single application can comprise of hundreds of software components, deployed on thousands of servers organized in multiple tiers and interconnected by commodity Ethernet switches. Such applications must support high concurrent connection counts and operate with user-facing SLO, often defined in terms of tail latency to meet business objectives [1, 12, 47]. To meet these objectives, most such applications distribute all critical data (*e.g.,* the social graph) in the memory of hundreds of data services, such as memory-resident transactional databases [20, 62, 65, 67, 68], NoSQL databases [44, 55], key-value stores [15, 39, 43, 74], or specialized graph stores [6].

These in-memory data services typically service requests from hundreds of application servers (high fan-in). Because each user request often involves hundreds of data services (high fan-out) and must wait for the laggard for completion, the SLO of the data services must consider the long tail of the latency distributions of requests [12]. Individual task sizes often require only a handful of μs of user-level execution each. These services would therefore ideally execute at the highest throughput, efficiently use all system resources (CPU, NIC, and memory), and deliver a tail-latency SLO that is only a small multiple of the typical task service time [3].

This hunt for the killer microseconds [3] requires researchers to revisit assumptions across the network and compute stacks, whose policies and implementations play a significant role in exacerbating the problem [34].

Our work focuses on the efficient scheduling on multicore systems of these very fine-grain in-memory services. The theoretical answer is well understood: (a) single-queue, multiple-processor models deliver lower tail latency than parallel single-queue, single-processor models and (b) FCFS delivers the best tail latency for low-dispersion tasks while processor sharing delivers superior results in high dispersion service time distributions [70].

The systems answer is, unfortunately, a lot less obvious, in particular when considering high request rates consisting of short messages and small processing times. In such situations, the state-of-the-art uses multi-queue NICs (*e.g.,* RSS [57]) to scale the networking stack across the multiple cores of the system. Current designs force users to choose between conventional operating systems (*i.e.,* typically Linux), and more specialized kernel-bypass approaches. The former can efficiently schedule the resources of a multi-core server and prioritize latency-sensitive tasks [8] but suffers from high overheads for µs-scale tasks. The latter improves throughput substantially (by up to 6× for key-value stores [5]) through sweeping simplifications such as separation of control from the dataplane execution, polling, run-to-completion, and synchronization-free, flow-consistent mapping of requests to cores [5, 26, 27, 39, 42, 51].

These sweeping simplifications lead to two related forms of inefficiencies: (a) the dataplane is not a work conserving scheduler, *i.e.,* a core may be idle while there are pending requests; and (b) the dataplane suffers from head-of-line blocking, *i.e.,* a request may be blocked until the previous tasks complete execution. While these limitations might be acceptable to workloads with near-deterministic task execution time and relatively loose SLO (*e.g.,* some widely-studied memcached workloads [1, 43] with an SLO at $> 100\times$ the mean service time [5]), such assumptions break down when considering more complex workloads, *e.g.,* in-memory transaction processing with a TPC-C-like mix of requests or with more aggressive SLO targets.

We present ZYGOS[1], a new approach to system software optimized for µs-scale, in-memory computing. ZYGOS implements a work-conserving scheduler free of any head-of-line blocking. While the design decisions voluntarily deviate from dataplane principles, ZYGOS retains the bulk of their performance advantages. The design, implementation, and evaluation of ZYGOS makes the following contributions:

(1) The design of ZYGOS, which leverages many conventional operating system building blocks such as the use of symmetric multiprocessing networking stacks, alternate use of polling and interrupts, inter-processor interrupts (IPI), and task stealing with the overall goal of delivering a work-conserving schedule. ZYGOS is architected into three distinct layers: (a) a lower networking layer, which runs in strict isolation on each core, (b) a middle *shuffle layer* which allows idle cores to aggressively steal pending events, and (c) an upper execution layer, which exposes a commutative API to applications for scalability [10]. The shuffle layer eliminates head-of-line-blocking while also offering strong ordering semantics of events associated with the same connection.

[1]The Greek word for balancing scales.

(2) The implementation of ZYGOS, which includes an idle loop logic designed to aggressively identify task stealing opportunities throughout the operating system and down to the NIC hardware queues. Our implementation leverages hardware virtualization and the Dune framework [4] and handles IPIs in an exit-less manner similar to ELI [22].

(3) A methodology using microbenchmarks with synthetic service times to identify system overheads as a function of task size and distribution. This methodology allows us to identify both design limitations and implementation overheads. We apply this approach to Linux for event-driven execution models (using both partitioned and floating connections among threads), IX and ZYGOS and show that all converge as the task granularity increases, but at noticeably different rates, to distinct, well-understood models. For an SLO of 10× the mean service time at the 99^{th} percentile, ZYGOS achieves 75% of the maximum possible theoretical load for 10µs tasks, and 88% of the equivalent load for 25µs tasks (§6.1).

(4) We compare ZYGOS to IX, a state-of-the-art dataplane with strict run-to-completion that partitions flows onto cores [5]. While ZYGOS's scheduler introduces some necessary buffering, communication and synchronization (which are measurable for extremely small tasks), it eliminates head-of-line blocking and clearly outperforms IX for tasks ≥10µs (§6.1). IX does outperform ZYGOS for workloads with very small task sizes such as memcached. The difference is primarily due to IX's adaptive bounded batching, which is not currently supported in ZYGOS. (§6.2)

(5) Last but not least, we evaluate the benefits of ZYGOS for an in-memory, transactional database running the TPC-C workload. Our setup uses Silo [65], a state-of-the-art, in-memory transactional database prototype. As Silo is only a library, we added client/server support to Silo, ported it to Linux, IX, and ZYGOS, and benchmarked it using an open-loop load generator for an SLO of 1000µs at the 99^{th} percentile tail latency. ZYGOS can deliver a 1.63× speedup over Linux and a 1.26× speedup over IX. The speedup over Linux is explained by the use of many dataplane implementation principles in ZYGOS. The speedup over IX is explained by ZYGOS's work-conserving scheduler, which rebalances tasks to deliver consistently low tail latency nearly up to the point of saturation (§6.3).

The source code of ZYGOS, along with benchmarks, scripts and simulation models, is available in open source [75].

The rest of the paper is organized as follows: §2 provides background on the problem and the theory. §3 describes the experimental methodology and characterizes existing systems. We describe the design (§4), implementation (§5) and evaluation of ZYGOS (§6). We discuss a key tradeoff (§7), related work (§8), and conclude.

2 BACKGROUND

2.1 Scaling remote procedure calls

In-memory data services are typically exposed by remote procedure calls (RPC). The problem of efficiently handling incoming RPCs dates back to the original C10k problem [7] when socket scalability was the primary bottleneck. Today, fine-tuned commodity operating systems can serve millions of requests per second and over a million of concurrent connections on a commodity server [50, 69, 72].

The initial approaches to building scalable applications allocated a kernel thread or process per connection; servicing a new request required a scheduling decision. However, despite the sophistication of modern operating system schedulers such as Completely Fair Scheduler (CFS) [8] and Borrowed Virtual Time (BVT) [16], context switch and stack management overheads made developers move to more performant designs to serve incoming requests.

Today's scalable designs fall into two main event-oriented patterns: symmetrical and asymmetrical ones. Symmetrical designs split connections onto threads, and each thread interacts with the operating system using non-blocking system calls. This pattern is used by the popular `libevent` and `libuv` frameworks [37, 38]. On Linux, this pattern typically relies on the `epoll` system call, which has long provided a way to statically map connections to threads. To avoid cases of load imbalance across cores because of imbalance across connections, developers tried sharing the same connection among multiple threads. However, this led to thundering herd problems [32]. The recent addition in Linux 4.5 of `EPOLLEXCLUSIVE` avoids such problems since in most cases only one thread is woken up to serve `epoll` [19].

In the asymmetrical pattern, a small number of threads handle all network I/O, identify RPC boundaries and add RPC requests to a centralized queue from which other tasks pull requests. This pattern is used by frameworks such as `gRPC` [23] and applications such as recent versions of `nginx` [46] and Apache Lucene [41]. While this pattern may increase the latency of an individual request and impact throughput when the tasks are small, it provides for an elegant separation of concerns and enables the efficient use of all worker cores.

2.2 Kernel bypass and sweeping simplifications

Data plane approaches such as IX [5], Arrakis [51] and user-level stacks [14, 26, 52, 56, 59, 63] bypass the kernel and rely on I/O polling to both increase throughput and reduce latency jitter [34, 36]. For example, IX increases the throughput of memcached by up to 6.4× over Linux [5].

While these sweeping simplifications provide substantial throughput improvements, they come at a key cost when it comes to resource efficiency: the synchronization-free nature of dataplanes forces each thread to process *only* the packets that were directed to it by the NIC hardware. Assuming a balanced, high-connection count fan-in pattern, such a design does not substantially impact throughput or even mean latency as all cores would get *on average* the same amount of work. It, however, has a dramatic impact on tail latency when the load is below saturation as some cores may be idle while others have a backlog. Dataplanes that rely on historical information to rebalance future traffic from the NIC can only address persistent imbalances and resource allocations problems [5]. The same limitation exists for applications that are explicitly designed to statistically distribute the load on all cores such as MICA in its CREW and CRCW execution models [39]. While such a design prevents any sustained imbalance, the randomized selection process of mapping requests to cores does nothing to prevent temporary imbalance between cores.

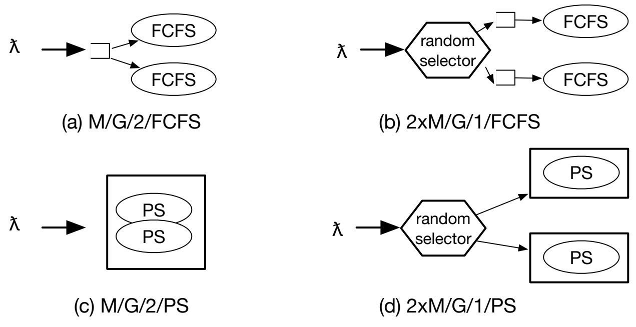


Figure 1: Queuing models for $n = 2$ processors.

2.3 Just enough queuing theory

There are at least three distinct forms of imbalance which impact tail latency that can be observed in systems:

(1) Persistent imbalance occurs when different NIC queues observe different packet arrival rates over long intervals. Unless the system can share the load dynamically, some cores will be busier than others. This situation can occur if there is connection skew when some clients request substantially more data than the average, or if there is data access skew (*e.g.,* the CREW protocol in MICA balances reads but not writes across cores [39]).
(2) Arrival bursts cause temporary queuing even when the system is not saturated. The well-known Poisson arrival process has such bursts which cause the gradual increase in tail latency as a function of load, even if the time to process each request is fixed. In a multi-queue system, the Poisson arrival process generates bursts on different cores at different points in time. This creates a form of *temporary* imbalance that also impacts tail latency.

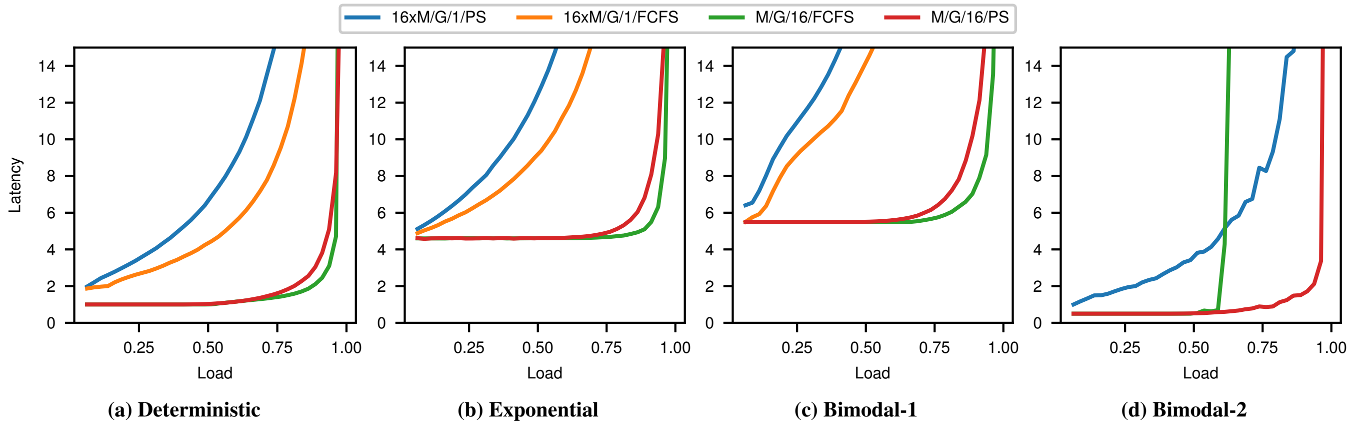


Figure 2: Simulation results for the 99^{th} percentile tail latency for four service time distributions with $\bar{S} = 1$.

(3) Service time variability will also create backlog and head-of-line blocking. A long request can occupy the processor for a long time, thus leading to a backlog of pending requests and a severe increase in tail latency.

We use four open-loop queuing models to build an intuition for the impact of arrival bursts and service time variability on tail latency. We use Kendall's notation to describe the models, where in the following expression `A/S/n/K`, `A` is the inter-arrival distribution, `S` is the service time distribution, `n` is the number of workers and `K` is the policy implemented, *i.e.,* first-come-first-serve (FCFS) or processor sharing (PS). For simplicity of the analysis, all models assume a Poisson inter-arrival time of requests (A=M). This is expected of many open-queuing models and representative of datacenter traffic with high fan-in connection counts. The Poisson process will generate arrival bursts and temporary imbalance in the multi-queue models, but no persistent imbalance.

Figure 1 illustrates the four different modes. Each delivers the same maximum throughput at saturation ($\lambda = n/\bar{S}$), but with different tail latencies. The models idealize the implementations of the systems of §2.

- The *centralized-FCFS* model (formally M/G/n/FCFS) idealizes event-driven applications that process events from a single queue or that float connections across cores (*e.g.,* using the `epoll` exclusive flag).
- The *partitioned-FCFS* model (formally n×M/G/1/FCFS) idealizes event-driven applications that partition connections among cores (*e.g.,* `libevent`-based applications) and associate each core with its own private work queue. This model can be deployed on conventional operating systems or shared-nothing dataplanes
- M/G/n/PS idealizes the thread-oriented pattern (1 thread per connection) deployed on time-sharing operating systems. In practice, the task size granularity must be a multiple of the operating system time quantum.
- n×M/G/1/PS similarly idealizes the thread-oriented pattern when the operating system does not rebalance threads among cores.

Figure 2 illustrates simulation results for these idealized queueing models for a system with $n = 16$ processors. The figure shows the result for four well-known distributions [40]:

- deterministic $P[X = \bar{S}] = 1$
- exponential with mean service time $\bar{S}$
- bimodal-1:$P[X = \bar{S}/2] = .9; P[X = 5.5 \times \bar{S}] = .1$
- bimodal-2:$P[X = \bar{S}/2] = .999; P[X = 500.5\times\bar{S}] = .001$

For each distribution, Figure 2 shows the tail request latency (queuing delay + service time) at the 99^{th} percentile as a function of the load. Intuitively we understand that as the system load increases and approaches the system's limits, the number of requests in the queues also increases. That leads to an increase in the queueing time and tail latency. As expected, the minimum 99th-percentile latency is 1 for the deterministic distribution and 4.6 for the exponential distribution. As for the two bimodal distribution, b1 has a minimum tail latency of 5.5, which corresponds to the slow requests and b2 has a minimum tail latency of 0.5, which corresponds to its fast requests.

We make two observations that inform our system design:

Observation 1: Single-queue systems (*i.e.,* `M/G/n/*`) perform better compared to systems with multiple queues (*i.e.,* n`×M/G/1/*`): Systems with multiple queues, even with random assignment of events to queues, suffer from temporary load imbalance. This imbalance can create a backlog on some processors while other queues are empty. The lack of work conservation in such models limits performance. In contrast, single-queue models with a work-conserving scheduler (whether FCFS or PS) can immediately schedule the next task on any available processor.

Observation 2: FCFS performs better in regards to tail latency for distributions with low dispersion: This result has also been theoretically analyzed by Wierman et al. [70]. In Figure 2, FCFS outperforms PS for the deterministic, exponential and bimodal-1 distribution. PS only outperforms FCFS when the variance in service times increases, as in the case for bimodal-2. Note that partitioned-FCFS performs that poorly in bimodal-2 that is not obvious in these axis scales.

3 EXPERIMENTAL METHODOLOGY

We now describe the experimental methodology used to evaluate existing low-latency systems. The challenge is to define metrics that help determine (a) the inherent design tradeoffs by comparing real-life systems with the idealized models of §2.3; and (b) the sweet spot, in terms of mean service time and distribution, of each system. We use synthetic microbenchmarks to compare analytical results with experimental baseline results for three OS configurations.

3.1 Approach and metrics

We rely on microbenchmarks with synthetic execution times to systematically compare different systems approaches for different task granularities. From the perspective of user-level execution, the applications are trivial: for each request, the application spins for an amount of time randomly selected to match both service time ($\bar{S}$) and distribution. From a systems perspective, the application follows the event-driven model to accept remote procedure calls sent over TCP/IP socket by client machines. The clients approximate an open-loop load-generator where incoming requests follow a Poisson inter-arrival time on randomly-selected connections [58]. All throughputs (requests per second) and 99^{th} percentile tail latencies are measured on the client-side.

We use two metrics to compare systems: (a) the conventional "tail latency vs. throughput" is used to compare the efficiency of different systems for a given task granularity and distribution; (b) the "maximum load @ SLO" is used to compare the efficiency across timescales, for a given SLO expressed as a multiple of the mean service time.

This second metric is used to determine how fast different systems converge (or not) to the expected behavior of their idealized model, as the service time increases. For example, consider an SLO that requires 99% of requests to complete within $\leq 10 \times \bar{S}$. Queuing theory predicts a maximum load for each configuration, *e.g.,* for the exponential distribution a load of 53.7% for the partitioned-FCFS model and of 96.3% for centralized-FCFS.

3.2 Experimental Environment

Our experimental setup consists of a cluster of 11 clients and one server connected by a Quanta/Cumulus 48x10GbE switch with a Broadcom Trident+ ASIC. The client machines are a mix of Xeon E5-2637 @ 3.5 GHz and Xeon E5-2650 @ 2.6 GHz. The server is a Xeon E5-2665 @ 2.4 GHz with 8 cores (16 hyperthreads) and 256 GB of DRAM. All machines are configured with Intel x520 10GbE NICs (82599EB chipset). We connect the clients and the server to the switch through a single NIC port each. The client machines run mutilate [34] as a load generator: 10 machines generate load and the 11th one measures latency. The machines connect to the server over a total of 2752 TCP/IP connections. To minimize client latencies, we modified the latency-measurement agent of mutilate to use a DPDK-based, simple TCP/IP stack.

The machines run an Ubuntu LTS 16.04 distribution running Linux kernel version 4.11. Systems are tuned to reduce jitter: all power management features, including CPU frequency governors and TurboBoost, and support for transparent huge pages, are disabled.

3.3 Evaluated Systems

The synthetic microbenchmark models an event-oriented, scalable RPC server. During the setup phase, it accepts all connections from the client machines. During the benchmark, it simply receives request messages from the open connections, spins during the requested amount of time and returns a response. The server is setup as a 16-way multi-threaded application that uses all cores (and hyperthreads) and memory of the CPU socket connected to the NIC. We deliberately leave the other socket unused to eliminate the potential impact of NUMA policies in this study. We compare three configurations designed to support a large number of incoming connections:

- **Linux-partitioned:** This mode minimizes communication and application logic at the expense of load-imbalance: each thread accepts its set of connections (as directed by the RSS in the NIC [57]) and then polls on that same set during the benchmark. *Partitioned-FCFS* models the performance upper bound.
- **Linux-floating:** In this mode, all open connections are put into a single pool from which all threads may poll. Our implementation uses a simple locking protocol to serialize access to the same socket. *Centralized-FCFS* models the upper bound of performance.
- **IX:** The application uses the native dataplane ABI to receive socket events and respond correspondingly. This is also modeled as *centralized-FCFS*.

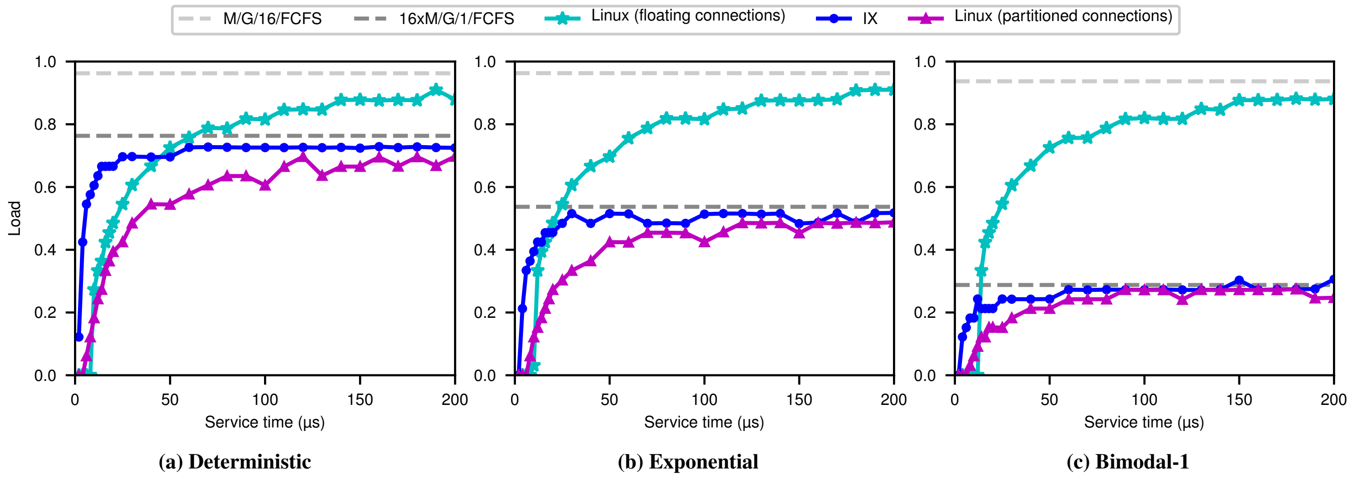


Figure 3: Maximum load that meets the SLO as a function of the mean service time $\bar{S}$. The SLO is set at $\leq (10 \times \bar{S})$ at the 99^{th} percentile. The grey lines correspond to the ideal upper bounds determined by the centralized-FCFS and partitioned-FCFS models.

Linux configuration: The Linux systems were tuned to minimize latency and maximize throughput, by settling them on a configuration that limits the number of returned events by `epoll` to 1. We did observe that some of these settings had a surprisingly small, or even negative impact on either latency or throughput (*e.g.,* the `EPOLLEXCLUSIVE` commit evaluated the impact on thundering herds on a 250-thread setup whereas we only use one per core [19]). We attribute this to the fact that we pinned each application thread to a distinct core, thereby avoiding many of the subtle interactions associated with CPU scheduling.

IX configuration: IX can process bounded batches of packets to completion, which improves throughput only for very small task sizes. Unless when explicitly mentioned, we disabled it in our experiments as disabling batching noticeably improves tail latency. We also disabled the control plane and configured IX to use all 16 hardware threads of the socket and use the CPU at its nominal frequency of 2.4GHz.

3.4 Baseline results

Figure 3 shows the maximum load that meets the SLO of the 99^{th} percentile $\leq 10 \times \bar{S}$ for three baseline operating system configurations described in §3.3. We include in greyscale two horizontal lines that correspond to the upper bound in performance, as predicted by the *partitioned-FCFS* and *centralized-FCFS*, respectively. These upper bounds assume zero operating system overheads, no scheduling overheads, no propagation delays, no head-of-line blocking, no interrupt delays, *etc.* In addition, the centralized model assumes a perfect, global FCFS order of the allocation of requests to idle processors.

Figure 3 shows the result for three of the four distributions studied analytically in Figure 2. We omit the bimodal-2 results as the analysis of §2.3 showed that multi-queue systems have pathological tail latency with an FCFS scheduler. The figure shows clearly that:

(a) IX and `Linux-partitioned` both converge asymptotically to the expected 16×M/G/1 level of performance. Intuitively, we understand that as the service time increases, the overhead of the operating system becomes less prevalent. IX, which is optimized for small tasks, reaches 90% efficiency with tasks ≥25µs, ≥25µs, and ≥60µs for the deterministic, exponential, and bimodal-1 distributions. Larger tasks are required for Linux-partitioned to reach the same level of efficiency, *i.e.,* ≥120µs, ≥120µs, and ≥90µs, respectively.

(b) Yet, `Linux-floating` actually provides the best performance for larger tasks and slowly converges to the upper bound predicted by the centralized-FCFS model. The ability to rebalance tasks across cores allows it to outperform IX for tasks that are ≥50µs, ≥20µs, and ≥14µs for the deterministic, exponential and bimodal-1 distributions.

4 DESIGN

4.1 Requirements

The theoretical analysis suggests, and in fact proves, that synchronization-free dataplane approaches cannot provide a robust solution to the tail latency problem, in particular when the service time distribution has a high dispersion. Yet,

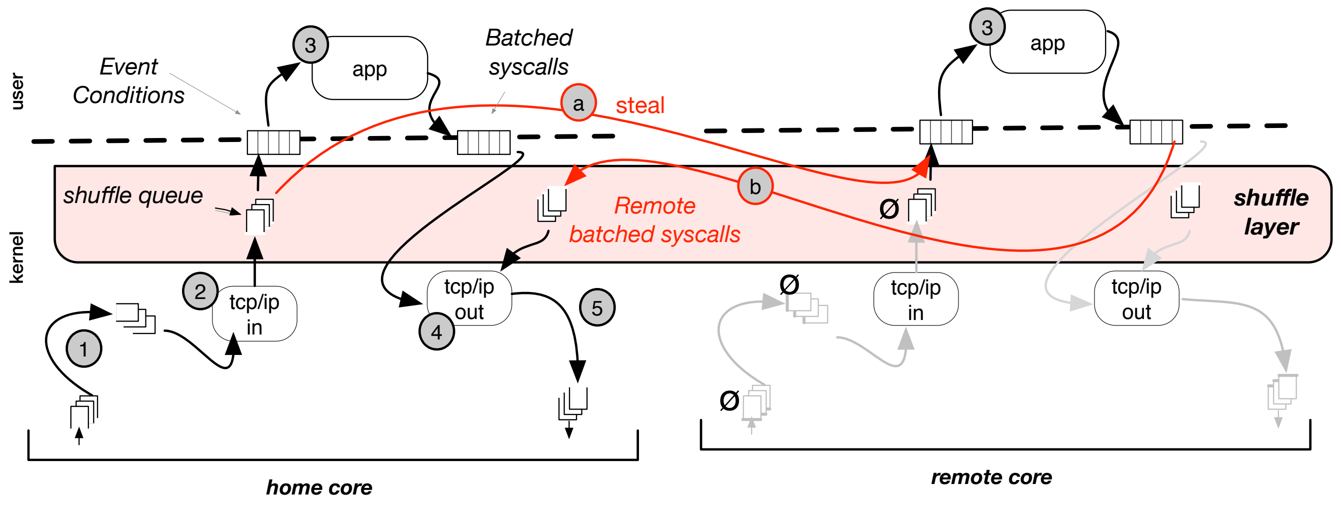


Figure 4: Dataflow in the ZYGOS operating system. Steps (1) – (6) correspond to the normal execution on the home core. Steps (a)-(b) occur during stealing and involve the home and remote cores.

synchronization-free dataplanes provide substantial throughput improvements over conventional operating systems.

We design ZYGOS, a single-address space operating system for the latency-sensitive data services, components of large-scale, online, data-intensive applications. Our design does not make any client-side assumptions or require any changes to the network protocol stack. We set the following hard requirements for our system design:

(1) Designed for current-generation datacenter architectures: Xeon multicore processors, 10GbE+ NICs with stateless offloads, Ethernet connectivity.

(2) Build a robust, multi-core, work-conserving scheduler free of head-of-line blocking for event-driven applications.

(3) Provide clean, ordering semantics of task-stealing operations to multi-threaded applications when handling back-to-back events for the same socket.

(4) Minimally degrade the throughput of short tasks when compared with state-of-the-art, shared-nothing dataplanes.

These hard requirements constrain the design space. While commodity operating systems such as Linux meet requirements #1 and #2, they provide only partial support for #3, which we will discuss in §4.3. As discussed in §3, the strict run-to-completion approach of dataplanes and their shared-nothing design is not an appropriate architectural foundation. We also rule out asymmetrical approaches which dedicate some cores to specific purposes (such as network processing) as the partitioning of resources is highly sensitive to assumptions on task granularities.

4.2 ZYGOS High-level Design

ZYGOS shares a number of architectural and implementation building blocks with IX [5]: each ZYGOS instance runs a single application in a single address space, and accesses the network through its dedicated NIC (physical or virtual function) with a dedicated IP address; each ZYGOS instance runs on top of the Dune framework [4]; a separate control plane can adjust resource allocations among instances.

Despite the lineage, ZYGOS is designed with radically different scheduling and communication principles than IX: IX is designed around a coherency-free execution model, *i.e.,* no cache-coherence traffic among cores is necessary, in the common case, to receive packets, open connections, or execute application tasks; ZYGOS is optimized for task stealing which has intrinsic communication requirements. IX achieves high throughput through adaptive batching, an approach that ensures that a batch of packets is first carried through the networking stack and then —without further buffering— processed by the application; ZYGOS uses intermediate buffering to enable stealing. Finally, IX is also designed around a run-to-completion model where it alternates execution between network processing and application execution, which cannot be interrupted; ZYGOS relies on intermediate buffering and IPIs to eliminate head-of-line blocking.

ZYGOS achieves work-conservation with minimal throughput impact by architecturally separating the execution stack into three distinct layers, illustrated in Figure 4:

(1) the lower **networking layer** executes independently on each core, in a coherency-free manner. This includes the hardware/software driver layer, which relies on RSS to dispatch flow-consistent traffic to one receive queue per core. This also

includes the layer-4 TCP/IP and UDP/IP layer, all of their associated data structures, intermingled queues, and timers. This design eliminates the need for any locking within the networking stack and ensures good cache locality.

(2) the intermediate **shuffle layer** introduces a new data structure per core: the *shuffle queue* is a single-producer, multiple-consumer queue which contains the list of *ready* connections originating from a given core. Connections in the shuffle queue contain at least one outstanding event and can be consumed by the core that produced it —the *home core*—, or atomically stolen by another *remote* core.

(3) the **application execution layer** manages the interactions between the kernel and the application through event conditions and batched system calls [61]. Each core has its own data structures and also operates in a coherency-free execution manner within that layer. Obviously, the application itself may have synchronization or shared-memory communication between cores and does not, in the general case, execute in a coherency-free manner.

Figure 4 shows the typical flow of events. Events numbered (1) – (5) occur when the packet is processed on its home core (*i.e.,* when no stealing occurs): (1) the driver dequeues packets from the hardware ring into a software queue; (2) the TCP/IP stack processes a batch of packets and enqueues ready connections into the shuffle queue; (3) the application execution layer dequeues the top entry, generates corresponding event conditions for the application and transfers execution to it. This, in turn, generates batched, system calls; (4) some system calls may call back into the network stack leading to execution of timers and/or (5) packet transmits. While the control flow resembles that of IX, the data flow is distinct as the shuffle queue breaks the run-to-completion assumptions as data is asynchronously produced into it and consumed from it.

Figure 4 also shows the interactions during a steal as the steps (a)-(b) in red. Consider the case where the remote core has no pending packets in the hardware queue, no pending packets in the software queue and no pending events in its shuffle queue. In step (a), it can then steal from another shuffle queue, which leads to the normal execution of the events in userspace, as step (3). The resulting batched system calls that relate to the networking stack are then enqueued for processing back at the home core in a multiple-producer, single-consumer queue, shown in step (b). Similar to the TCP input path, the TCP output path therefore also executes in a coherency-free manner on the home core.

Figure 4 is only a high-level illustration of the system. In ZYGOS, each core is the home core of a set of flow-groups, as defined by the NIC RSS configuration and can act as the remote core for any other flow whenever it is idle. We now describe the ordering semantics that enable stealing (§4.3) and the data structures of the shuffle queue that eliminates head-of-line blocking (§4.4).

4.3 Ordering semantics in multi-threaded applications

When TCP sockets are statically assigned to threads, applications can rely on intuitive ordering and concurrency semantics [33]. The situation changes dramatically when sockets can float across cores as the `read` system call is not commutative when two threads access the same socket. Even though the Linux system call `epoll` allows it, and was even recently optimized for this use case [19], the implications on applications are far from trivial. Consider the case of back-to-back messages sent to the same socket (*e.g.,* two distinct RPC of the memcached protocol) for a multi-threaded application that uses the Linux-floating model of §3.3. Unless the application takes additional steps at user-level to synchronize across requests, race conditions lead to broken parsing of requests, out-of-order responses, or worse, intermingled responses on the wire. As a practical manner, applications or frameworks must, therefore, build their own synchronization and locking layer to eliminate these system races. This is sufficiently non-trivial that no known popular applications have done it to date, to the best of our knowledge. A related approach is the recent KCM kernel patch that provides a multiplexing layer of messages to TCP connections [28, 29].

With its goal to ensure very fine-grain work-stealing, we designed ZYGOS to free the application layer from the burden of synchronizing access to connection-oriented TCP/IP sockets. In this case, ZYGOS has an ownership model that ensures the events that relate to the same socket are implicitly ordered without the need for synchronization: whenever the home core or a remote core grabs an event for processing at the application layer, it grabs the exclusive access to the socket until the event execution has completed, including sending the replies on the TCP socket.

4.4 Eliminating Head-of-Line Blocking

The ordering semantics of §4.3 introduce a substantial complication to the design of the shuffle queue. ZYGOS eliminates head-of-line blocking by grouping events in the home core by socket. The shuffle queue has the ordered subset of sockets that are (a) not currently being processed on a core and (b) have pending data. The event queues are held in the per-socket protocol control block (PCB). While it offers strong ordering semantics to applications, this pre-sorting step does have an implication on the global order of packets, which is no longer guaranteed to be FCFS.

Figure 5 shows the state machine diagram that controls the decisions for each socket. Changes to the state machine and to the shuffle queue are atomic.

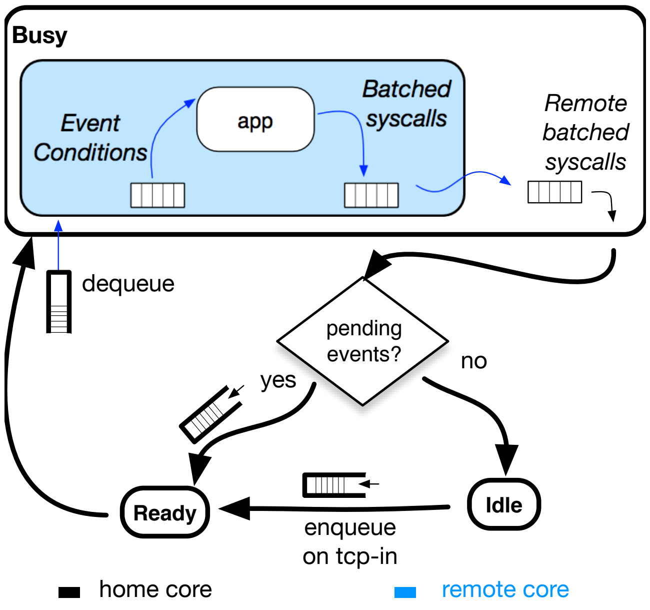


Figure 5: Connection state machine transitions for the general case where an event is executed on a remote core (in blue). The connection is present in the shuffle queue exactly once when it is in the "ready" state, and never otherwise.

- *idle:* Sockets in this state have no pending incoming events, events currently processed by the application, or outgoing batched system calls.
- *ready:* The socket has pending incoming events, but is not currently being processed by the application and has no pending system calls.
- *busy:* The socket is associated with an execution core, which is either the home or remote core.

The execution core dequeues the first ready connection and creates the event conditions for the application. As previously discussed, system calls are returned back to the home core for processing. System calls may each generate asynchronous responses for that socket. After the execution of all system calls, the socket transitions either into the *idle* state if there is nothing further to process or into the *ready* state otherwise. In the latter case, the PCB is once-again enqueued into the shuffle queue.

4.5 Inter-processor Interrupts

The design in §4.4 eliminates head-of-line blocking concerns from the shuffle queue itself. In a purely cooperative implementation of ZYGOS, the cores poll on each other's data structures, which causes head-of-line blocking situations both before network processing as well as after application execution, since network processing explicitly takes place in the home core.

First, consider the case where packets are available for network processing in the hardware NIC queue but the shuffle queue is empty. This is the queue shown around step (1) in Figure 4. As long as that core is executing application code, no remote core can steal the task. Idle cores poll both software and hardware remote packets queues. If pending packets exist, it sends an IPI to the remote core can force the execution of the networking stack, which replenishes the shuffle queue.

Second, remote batched system calls are enqueued by the remote core for execution on the home core (shown as step (b) of Fig 4). In a cooperative model, these system calls are only executed after the completion of application code, which unfortunately directly impacts RPC latency as some of these system calls write responses on the socket. Here also, an IPI ensures the timely execution of these remote system calls.

The shared IPI handler, therefore, performs two simple tasks when interrupting user-level execution: (1) process incoming packets if the shuffle queue is empty and (2) execute all remote system calls and transmit outgoing packets on the wire. The IPI interrupts only user-level execution since kernel processing is short and bounded. The kernel executes with interrupts disabled, thus avoiding starvation or reentrancy issues in the TCP/IP stack.

5 IMPLEMENTATION

The system architecture of ZYGOS is derived from the IX open-source release v1.0 [53]: it relies on hardware virtualization and the Dune framework [4] to host a protected operating system with direct access to VMX non-root mode ring 0 in the x86-64 architecture [66]. The kernel links in with DPDK [14] for NIC drivers and lwIP for TCP/IP [17]. The modifications to the application libraries are minor, but the kernel changes are extensive. Specifically, we modified ~2000 LOC of the IX kernel and ~200 LOC of Dune. While we retain the tight code base of IX, we revisit many of its fundamental design assumptions and principles.

The shuffle layer: We chose a simple implementation to ensure the atomic transitions described in §4.4. There is one spinlock per core which protects the shuffle queue of that core as well as the state machine transitions for sockets that call that core home. The lightweight nature of the operations that access it makes such a coarse-grain approach possible. Remote cores rely on `trylock` for their steal attempts to further reduce contention. Each PCB maintains a distinct event queue of pending events. This is a single-producer (the home core) and single consumer (the execution core) queue, implemented with one spinlock per PCB. The transitions from the *busy* state must test whether the PCB queue is empty and must first grab that lock.

Idle loop polling logic: The core design principle of ZYGOS is to ensure that an idle core will aggressively identify pending work. A core is idle when its shuffle queue, remote batch system call queue, and software raw packet queue are all empty. When it enters its idle mode, it starts to poll a sequence of memory locations, all of which are reads from cacheable locations. These locations include, in order of priority (a) the head of its own NIC hardware descriptor ring, (b) the shuffle queue of all other cores, (c) the head of all unprocessed software packet queues of all other cores, and (d) the head of the NIC hardware descriptor of all other cores. For steps (b-c-d), the order of access is randomized. While heuristics could tradeoff a reduction of interrupts for a slight degree of non-work conservation, our current implementation aggressively sends interrupts as soon as a remote core detects a pending packet in the hardware queue and the home core is executing at user-level.

Exit-less Inter-processor Interrupts: ZYGOS relies on inter-processor interrupts to force a home core to process pending packets identified in steps (c) and (d) of the idle loop and to execute remote system calls back on the home core. Using an approach similar to ELI [22], we added support in Dune for exit-less interrupts in non-root mode, based on the assumption that ZYGOS kernel's interrupt handler will redirect to the Linux host operating system the interrupts that are destined to it. There is, however, no guarantee that the destination CPU will be VMX non-root mode when it receives the interrupt. We use interrupt 242, which is also used by KVM [30]. Interrupts received in root-mode are simply ignored by the KVM handler. As interrupts are used exclusively as hints, the unreliability of delivery impacts tail latency, but not correctness.

Control plane interactions: The IX control plane implement energy proportionality or workload consolidation by dynamically adjusting processor frequency and core allocation [54]. It operates in conjunction with the IX dataplane, which reprograms the NIC RSS settings. In principle, ZYGOS is compatible with these RSS settings changes, although policies and mechanisms would have to be adjusted as ZYGOS introduces new forms of buffering. We leave the evaluation of these interactions to future work.

6 EVALUATION

We use the same experimental setup explained in Section 3 to evaluate ZYGOS in a series of microbenchmarks, use memcached [43] to evaluate overheads on tiny tasks, and with a real application running TPC-C [64].

6.1 Synthetic micro-benchmarks

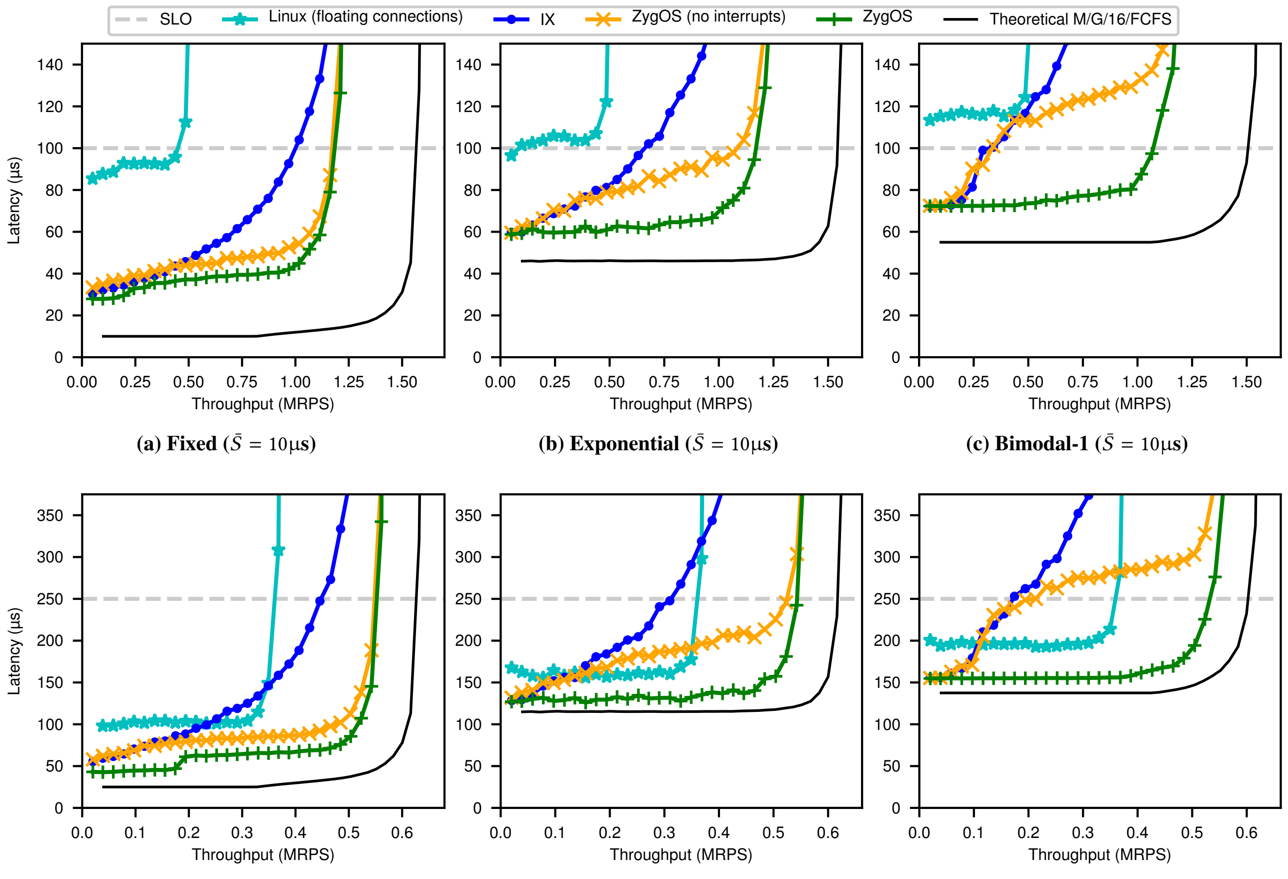
Figure 6 shows the latency vs. throughput of the three synthetic micro-benchmarks of §3. We compare ZYGOS with existing systems (IX and Linux) as well as the theoretical performance of a zero-overhead M/G/16/FCFS model for two granularities of interest, namely 10µs and 25µs. We observe that:

- ZYGOS and `Linux-floating` both approximate the theoretical model, with ZYGOS substantially reducing tail latency over IX;
- ZYGOS and IX have comparable throughput, even for tasks as small as 10µs; both clearly outperform Linux;
- for the exponential distribution, ZYGOS achieves 75% throughput efficiency at the SLO at $10 \times \bar{S}$ for $\bar{S} = 10\mu s$ (Figure 6b) and 88% for $\bar{S} = 25\mu s$ (Figure 6e);
- interrupts are necessary to eliminate head-of-line blocking with medium and high dispersion workloads, and the cooperative model of `Zygos-no-interrupts`, which is typical of pure user-level application, visibly impacts tail latency.

Efficiency for the $10 \times \bar{S}$ tail latency SLO: Figure 7 reports the efficiency (in terms of max load at SLO) as a function of task size. We compare ZYGOS with the baseline shown in Figure 3. We note the reduced X-axis truncated to 50µs for visibility; efficiency is stable beyond that point. ZYGOS clearly outperforms IX and Linux for any tasks sizes ≥5 µs and all three distributions for such a tight SLO. ZYGOS reaches 90% of the maximum possible load as determined by the zero-overhead *centralized-FCFS* theoretical model for tasks ≥30µs for the deterministic distribution, ≥40µs for exponential and ≥40µs for bimodal-1.

How much task stealing occurs?: Figure 8 provides an insight into the rate of stealing events as a function of load. The results are for the exponential distribution of Figure 6e but are remarkably similar for other distributions and timescales. As expected, there are few steals at low loads as more cores are near idle, and no steals at saturation as all cores are busy processing their own queue.

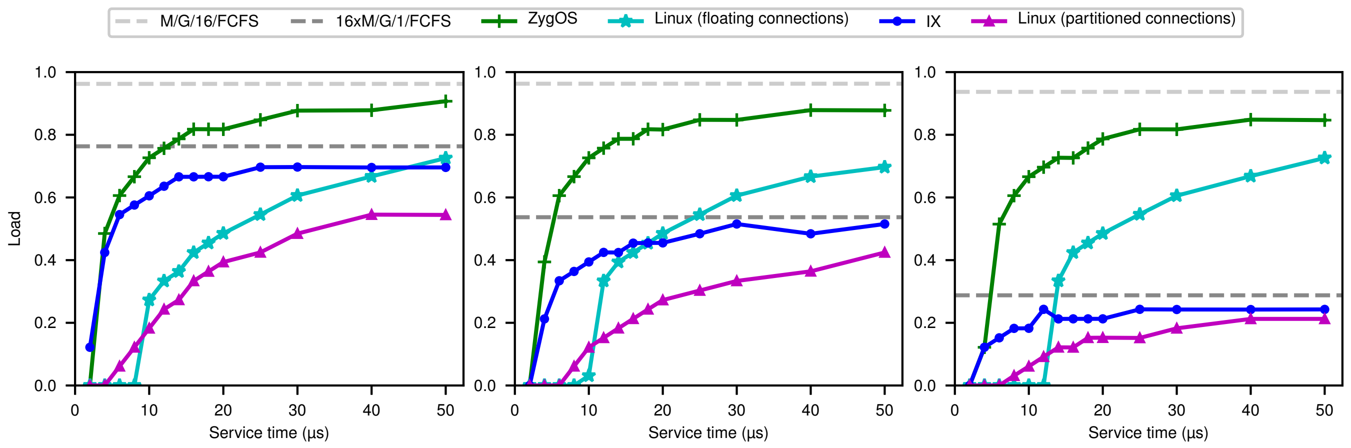
Without interrupts, temporary imbalances lead to a steal rate that peaks at ~33%. This rate is consistent with the peak of ~35% measured in a discrete event simulator that emulates the shuffle queue in a cooperative model without interrupts. Interrupts, —which are necessary to eliminate head-of-line blocking—, substantially increase the steal rate. At the peak, which corresponds to 77% of saturation, steals, and therefore interrupts are very frequent. Stealing opportunities become less frequent as the load further increases.



(a) Fixed ($\bar{S}$ = 10µs) **(b) Exponential** ($\bar{S}$ = 10µs) **(c) Bimodal-1** ($\bar{S}$ = 10µs)

(d) Fixed ($\bar{S}$ = 25µs) **(e) Exponential** ($\bar{S}$ = 25µs) **(f) Bimodal-1** ($\bar{S}$ = 25µs)

Figure 6: 99^{th} percentile tail latency according to throughput for three distributions, each with 10µs and 25µs mean task granularity. The horizontal line corresponds to the SLO of $\leq 10 \times \bar{S}$.



(a) Deterministic **(b) Exponential** **(c) Bimodal-1**

Figure 7: Maximum load that meets the SLO of the 99^{th} percentile $\leq 10 \times \bar{S}$. The grey lines correspond to the ideal upper bounds of the two theoretical, zero-overheads, models (centralized-FCFS and partitioned-FCFS).

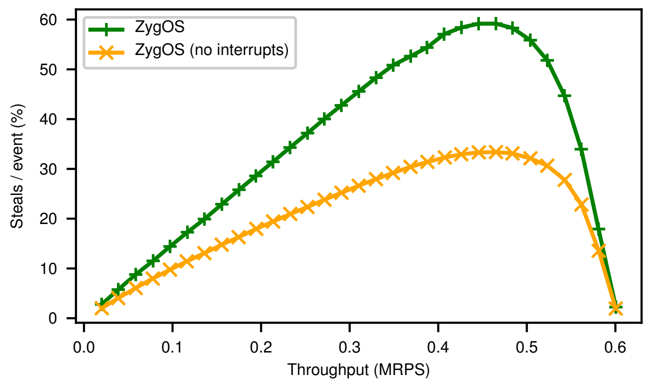


Figure 8: Normalized rate of stealing vs throughput for exponential service time with mean 25 µs

6.2 Overheads of ZYGOS on tiny tasks: memcached

We compare the overheads of ZYGOS to IX for tiny tasks with the goal of identifying the task granularity where the sweeping simplifications of shared-nothing dataplanes such as IX noticeably improve throughput. We use memcached as an application ($< 2\mu s$ mean task size), and use the methodology and reproduce the results from IX [5]. We consider memcached a near worst case for ZYGOS as the application has very small task size with a small dispersion best approximated by a deterministic distribution.

Figure 9 shows the latency vs. throughput for the USR and ETC workloads, [1], as modelled by mutilate [34]. We compare Linux, ZYGOS, and IX. For IX, we choose two configurations: with adaptive batching disabled (B=1) and with adaptive batching enabled with the default setting (B=64).

First, we observe that ZYGOS and IX both clearly outperform Linux. We then note that for this particular SLO (500 µs), ZYGOS outperforms IX with batching disabled but lags behind IX with adaptive bounded batching. IX implements a strict run-to-completion model bounded by the batch size (B). ZYGOS currently implements adaptive bounded batching only on the receive path. It then processes events individually, interleaving between user and kernel code. While this hurts cache locality, it avoids head-of-line blocking. Similarly, it eagerly sends packets through the TX TCP/IP path and the NIC, also to avoid head-of-line blocking.

Of note, ZYGOS has a differently shaped latency vs throughput curve for this workload. As described in §4.3 and §4.4, ZYGOS does not respect strict FIFO ordering on servicing packets across different connections. For this workload configuration, up to four distinct memcached requests can be pipelined onto the same connection. The resulting reordering leads to a form of implicit batching of events, but only for those corresponding to the same flow. This implicit batching improves throughput but at an increase in tail latency. Such a behaviour is hard to restrict since ZYGOS doesn't know the boundaries of the requests in the TCP byte stream. Linux applications which use KCM sockets [28] can potentially handle this situation.

6.3 A real application: Silo running TPC-C

We validate the tail latency benefits of ZYGOS using Silo [65], a state-of-the-art in-memory database optimized for multicore scalability.

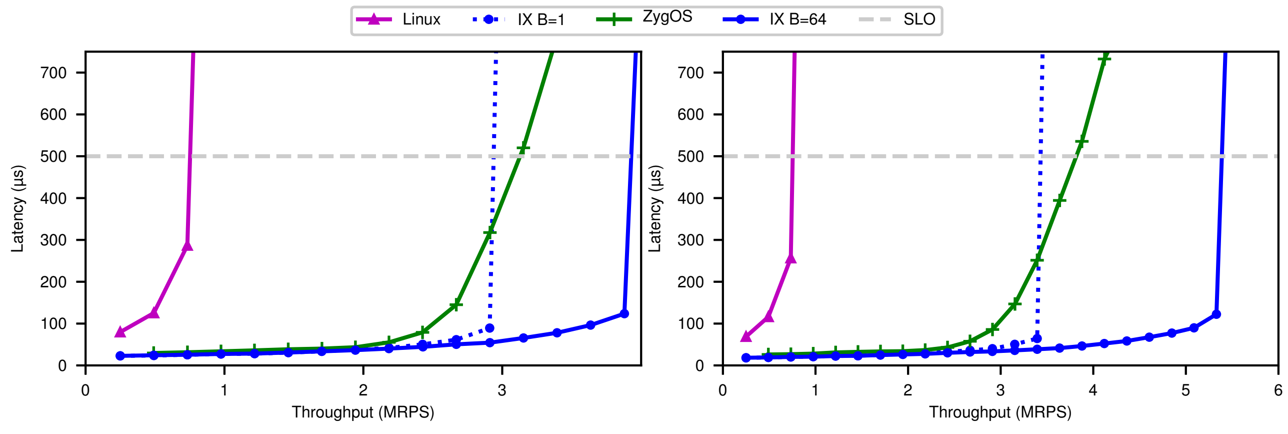
6.3.1 Application setup. Silo was originally implemented and evaluated as a library linked in with the benchmark. In the original evaluation, each thread runs as a closed loop issuing transaction requests, and in particular the TPC-C mix.

We ported Silo to run as a networked server accepting requests over sockets. We replaced the main loop of Silo with an event loop, which we used to run the workload on top of Linux, IX, and ZYGOS. The workload uses mutilate [34] with the same setup described in §3.2 to initiate transactions that then execute totally within the database server. Each remote procedure call generates one transaction from the TPC-C mix of requests.

We did not attempt to implement a marshalling of the full SQL queries and their responses, *e.g.,* over a JDBC-like protocol, as this falls outside the scope of the research question. We also note that Silo has a garbage-collection phase tied to its epoch-based commit protocol, which introduces a periodic barrier for all threads, with transaction latencies exceeding $1ms$. We disabled garbage collection for our measurements as it adds experimental variability, especially at the 99^{th} percentile, depending on the experiment (and that taming the tail latency impact of Silo's GC also falls clearly outside the scope of this work)

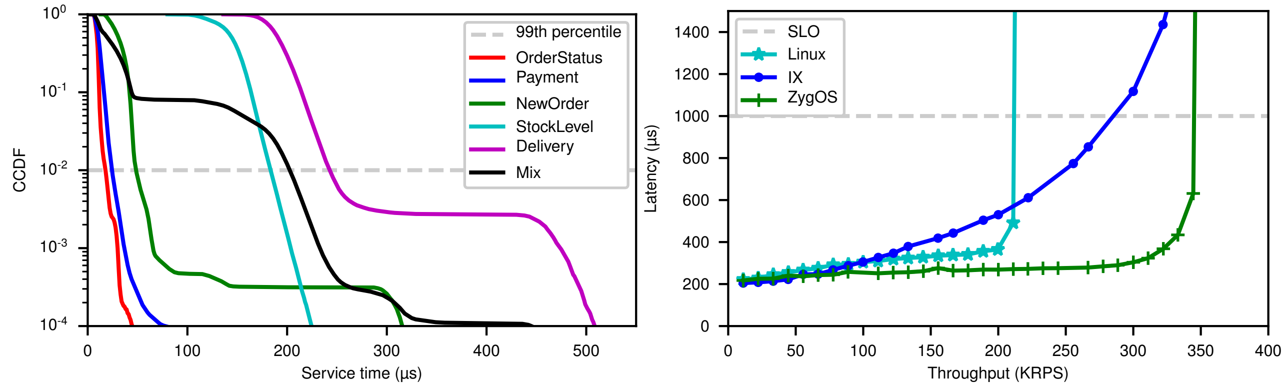
6.3.2 Results. Figure 10a shows the complementary cumulative distribution of service time for the TPC-C benchmark for each of the five transaction types of the benchmark as well as the mix. The results were computed using Silo's master branch [60], with Silo locally driving the TPC-C benchmark. There is, therefore, no network activity, and indeed nearly no operating system activity. We run with GC disabled across all 16 hardware threads of a single CPU socket. The Figure reports the service time rather than that the end-to-end latency (*i.e.,* it excludes any queueing delays).

In this setup, the achieved transaction rate was 460 KTPS, which corresponds to the maximal throughput of the application, excluding any SLO and operating system overheads. Note that this TPS is consistent with the reported results in [65], given the differences in thread counts and processors. For the full mix, the average service time is 33µs, the median is 20µs, and the 99^{th} percentile is 203µs. The figure



(a) ETC **(b) USR**

Figure 9: 99th **percentile tail latency vs. throughput for two memcached workloads for Linux, IX and, ZYGOS.**



(a) Complementary CDF of task execution time (Linux) **(b) 99th percentile end-to-end latency vs. throughput**

Figure 10: Silo running the TPC-C benchmark.

System	Max load@SLO	Speedup	Tail Lat.@50%	Tail Lat.@75%	Tail Lat.@90%
Linux	211 KTPS	1.00×	310µs (1.5×) @111 KTPS	335µs (1.6×)@156 KTPS	356µs (1.8×) @189 KTPS
IX	267 KTPS	1.26×	379µs (1.9×) @133 KTPS	530µs (2.6×)@200 KTPS	774µs (3.8×) @256 KTPS
Zygos	344 KTPS	1.63×	265µs (1.3×) @178 KTPS	279µs (1.4×)@266 KTPS	323µs (1.6×) @311 KTPS

Table 1: Maximum throughput under the SLO of 1000 µs and respective latencies at approximately 50%, 75%, and 90% of that load for each Silo running the TPC-C benchmark. The number in the parentheses is the ratio of the 99th percentile end-to-end latency to Silo's 99th percentile service time (203µs).

clearly shows that Silo's service time distribution is overall multi-modal with small task granularity in the µs-scale.

Figure 10b shows the tail latency at the 99th percentile for Silo as a function of the load. To compare maximum loads, we selected a stringent SLO of 1000µs, which corresponds to ~33× the average and ~5× the 99th percentile tail latency. We observe:

- ZYGOS can support 344 KTPS without violating the SLO; this corresponds to a speedup of 1.63× over Linux. This demonstrates the benefits of our approach for real-life in-memory applications. The achieved transaction rate corresponds to 75% of the ideal, zero-overhead load with no SLO restrictions.

- This rate also corresponds to a speedup of 1.26× over IX. ZYGOS's work-conserving scheduler and its ability to rebalance requests across cores avoids SLO violations until the system becomes CPU bound on all cores.

Table 1 further quantifies the benefits of ZYGOS in terms of throughput at SLO and tail latency at a specific fraction of their respective maximum load. ZYGOS and Linux both deliver low end-to-end tail latencies for up to 90% of their respective capacity: 1.6× the 99^{th} percentile service time for ZYGOS and 1.8× for Linux. This is anticipated by the *centralized-FCFS* model. In contrast, as anticipated by the *partitioned-FCFS* model, IX delivers substantially higher tail latencies, *e.g.,* 1.9× when operating at half capacity, 2.6× at 75% capacity, and 3.8× at 90% capacity.

7 DISCUSSION: THE IMPACT OF SLO ON SYSTEMS

The choice of an SLO is driven by application requirements and scale, with the intuitive understanding that a more stringent SLO reduces the delivery capacity of the system. We show that the choice of an SLO also informs on the choice of the underlying operating system and scheduling strategy.

Figure 11 illustrates the tradeoff through the latency vs. throughput curves for the synthetic benchmark of §6.1 with an exponential service time of $\bar{S}$ =10 µs. Figure 11a and 11b actually show the results of the same experiment but on two different Y-axis corresponding to two different SLO. ZYGOS consistently shines on the more stringent SLO of 100µs (Figure 11a, $10 \times \bar{S}$) as the work-conserving scheduler tames the tail latency, followed by IX with batching disabled. For this SLO, IX (with batching enabled) consistently delivers the highest tail latency and violates the SLO with the lowest throughput.

However, for a more lenient SLO (Figure 11b, $100 \times \bar{S}$), IX's adaptive batching delivers marginally higher throughput than ZYGOS before violating the SLO.

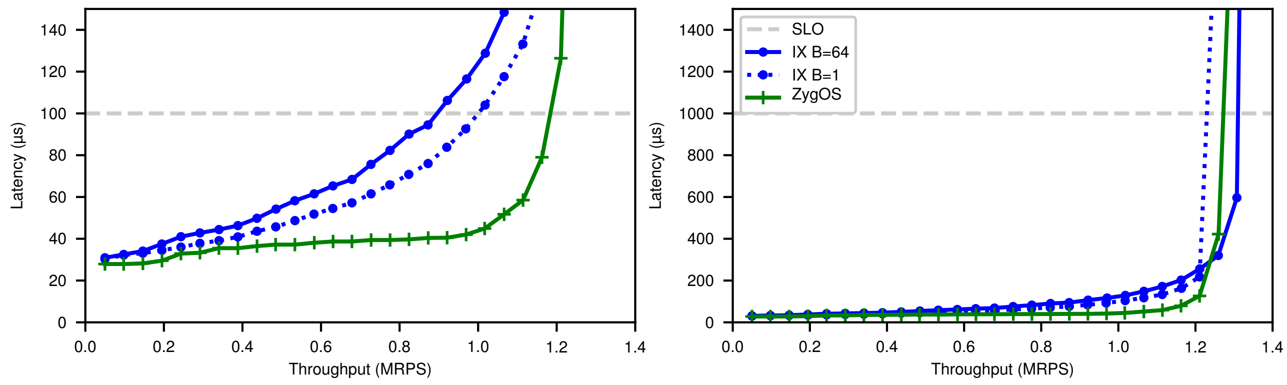
8 RELATED WORK

Traditional event-driven models: This is the de-facto standard approach for online data-intensive services with high connection fan-in. On Linux, the use of the `epoll` has substantially improved system scalability. While `epoll` can be used in a floating model, and the recent `epoll-exclusive` eliminates thundering herds [19], applications must still rely on additional, complex synchronization to take advantage of the feature. ZYGOS delivers built-in, ordered semantics that guarantee that the replies from back-to-back remote procedure calls on the same socket will be returned in order. However, unlike the case of Affinity-accept [50] where each connection remains local to the core that accepted it, ZYGOS enables a connection to be served by any available core. Hanford et al. [24] investigated the impact of affinity on application throughput and proposed to distribute packet processing tasks across multiple CPU cores to improve CPU cache hit ratio. Although our work does not consider cache effects, we also conclude that strict request affinity can harm performance.

Traditional multi-threading model: Operating systems preemptive schedulers such as CFS [8], BVT [16] favor latency-sensitive tasks. Applications can benefit from multithreading to lower tail latency of completion of tasks when the granularity is a multiple of the scheduling quantum and the distribution has a high dispersion.

Shared-nothing dataplanes architectures: Systems such as Arrakis [51], IX [5], mTCP [26], MICA [39], Seastar [59] and Sandstorm [42] bypass the kernel(via frameworks such as DPDK [14] or netmap [56]) and rely on NIC RSS to partition flows among cores. These shared-nothing architectures (at the system-level) with run-to-completion approaches completely eliminate the need to make scheduling decisions. These sweeping simplifications noticeably increase throughput but are oblivious to temporary imbalances across cores. MICA uses a client-side randomizing protocol (CREW or CRCW) to eliminate some causes for persistent imbalances among cores but does not address temporary imbalances. Decibel [45] and Reflex [31] are designed for storage disaggregation, depend on the shared-nothing assumption and similarly do not handle imbalance. ZYGOS is designed to eliminate such cases of imbalance though work-stealing. RAMcloud clients leverage RDMA hardware to bypass the kernel and communicate with a cluster of RAMcloud servers, with an asymmetric, push-based approach to task scheduling [48]. ZYGOS works with commodity Ethernet NICs and handles I/O and protocol processing symmetrically on all cores, with a pull-based, work-stealing scheme for task execution.

Work-stealing within applications: This commonly-used technique that has been mostly implemented either within the application or in a userspace run-time that runs on top of the operating system. Run-times such as Intel's Cilk++, Intel's C++ Threading Building Blocks (TBB), Java's Fork/Join Framework and OpenMP implement work-stealing schemes. Optimizing or building such run-times has also been studied intensely academically, *e.g.,* [9, 11, 13]. Statically mapping connections to cores can result in load imbalance in event-based programs and requires a solution at the library level [21, 73]. Recent focus on work stealing for latency-critical applications is at coarser timescales. [25, 35, 71]. The prior work largely targets applications with millisecond-scale task granularities that are easily accommodated by conventional operating systems. ZYGOS implements work-stealing



(a) 100 µs SLO **(b) 1000 µs SLO**

Figure 11: Comparison of IX (batch size 1 and 64) and ZYGOS for a deterministic service time of 10 µs and 2 different SLOs.

within the operating system itself for network-driven to eliminate both persistent and temporary imbalances and is suitable for µs-scale tasks. As an operating system, ZYGOS's use of IPIs eliminates all cooperative multitasking assumptions between the threads.

Cluster-level work-stealing: Finally, load imbalance has been extensively studied at cluster-scale. Lu et al. [40] proposed a 2-level load balancing scheme based on the power of two to load balance traffic towards the front-end of cloud services. Sparrow [49] also relies on power-of-two choices for batch job scheduling. Google's Maglev [18] is a generic distributed network load balancer that leverages consistent hashing to load balance packets across the corresponding services.

9 CONCLUSION

We presented ZYGOS, a work-conserving operating system designed for latency-critical, in-memory applications with high connection fan-in, high requests rates, and short individual task execution times. ZYGOS applies some well-proven work-stealing ideas within the framework of an execution environment but avoids the fundamental limitations of dataplane designs with static partitioning of connections. We validate our ideas on a series of synthetic microbenchmarks (with known theoretical bounds) and with a state-of-the-art, in-memory transactional database. ZYGOS demonstrates that it is possible to schedule µs-scale tasks on multicore systems to deliver high throughout together with low tail latency, nearly up to the point of saturation.

ACKNOWLEDGEMENTS

We thank Katerina Argyraki, Ashvin Goel, James Larus, the SOSP anonymous reviewers and our shepherd Ion Stoica for their detailed comments. Moreover, we would like to thank Christos Kozyrakis and Mia Primorac for participating in many early discussions that lead to ZYGOS. Finally, we gratefully acknowledge Adrien Ghosn and Utku Sirin that helped understanding Silo and setting up the related experiments.

This work was funded in part by the Microsoft-EPFL Joint Research Center, the NanoTera YINS project, and a VMware Research Grant. George Prekas is supported by a Google Graduate Research Fellowship.

REFERENCES

[1] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[2] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[3] BARROSO, L. A., MARTY, M., PATTERSON, D., AND RANGANATHAN, P. Attack of the killer microseconds. *Commun. ACM 60*, 4 (2017), 48–54.

[4] BELAY, A., BITTAU, A., MASHTIZADEH, A. J., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)* (2012), pp. 335–348.

[5] BELAY, A., PREKAS, G., PRIMORAC, M., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst. 34*, 4 (2017), 11:1–11:39.

[6] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI,

H. C., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)* (2013), pp. 49–60.

[7] 10 thousand connections problem. http://www.kegel.com/c10k.html, 1999.

[8] Linux cfs scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.

[9] Chase, D., and Lev, Y. Dynamic circular work-stealing deque. In *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2005), pp. 21–28.

[10] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Trans. Comput. Syst. 32*, 4 (2015), 10:1–10:47.

[11] Contreras, G., and Martonosi, M. Characterizing and improving the performance of Intel Threading Building Blocks. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization (IISWC)* (2008), pp. 57–66.

[12] Dean, J., and Barroso, L. A. The tail at scale. *Commun. ACM 56*, 2 (2013), 74–80.

[13] Dinan, J., Larkins, D. B., Sadayappan, P., Krishnamoorthy, S., and Nieplocha, J. Scalable work stealing. In *Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC)* (2009).

[14] Data plane development kit. http://www.dpdk.org/.

[15] Dragojevic, A., Narayanan, D., Castro, M., and Hodson, O. FaRM: Fast Remote Memory. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)* (2014), pp. 401–414.

[16] Duda, K. J., and Cheriton, D. R. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose schedular. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)* (1999), pp. 261–276.

[17] Dunkels, A. Design and Implementation of the lwIP TCP/IP Stack. *Swedish Institute of Computer Science 2* (2001), 77.

[18] Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W., and Hosein, J. D. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)* (2016), pp. 523–535.

[19] Epollexclusive kernel patch. https://lwn.net/Articles/667087/, 2015.

[20] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W. SAP HANA database: data management for modern business applications. *SIGMOD Record 40*, 4 (2011), 45–51.

[21] Gaud, F., Geneves, S., Lachaize, R., Lepers, B., Mottet, F., Muller, G., and Quéma, V. Efficient Workstealing for Multicore Event-Driven Systems. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS)* (2010), pp. 516–525.

[22] Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A., and Tsafrir, D. ELI: bare-metal performance for I/O virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVII)* (2012), pp. 411–422.

[23] gRPC. http://www.grpc.io/.

[24] Hanford, N., Ahuja, V., Balman, M., Farrens, M. K., Ghosal, D., Pouyoul, E., and Tierney, B. Characterizing the impact of end-system affinities on the end-to-end performance of high-speed flows. In *Proceedings of the Third International Workshop on Network-Aware Data Management* (2013), pp. 1:1–1:10.

[25] Haque, M. E., Eom, Y. H., He, Y., Elnikety, S., Bianchini, R., and McKinley, K. S. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)* (2015), pp. 161–175.

[26] Jeong, E., Woo, S., Jamshed, M. A., Jeong, H., Ihm, S., Han, D., and Park, K. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)* (2014), pp. 489–502.

[27] Kapoor, R., Porter, G., Tewari, M., Voelker, G. M., and Vahdat, A. Chronos: predictable low latency for data center applications. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC)* (2012), p. 9.

[28] Kernel connection multiplexer. https://lwn.net/Articles/657999/, 2015.

[29] Kernel connection multiplexer patch. https://lwn.net/Articles/657970/, 2015.

[30] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium* (2007), vol. 1, pp. 225–230.

[31] Klimovic, A., Litz, H., and Kozyrakis, C. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)* (2017), pp. 345–359.

[32] Laadan, O., Nieh, J., and Viennot, N. Structured linux kernel projects for teaching operating systems concepts. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE)* (2011), pp. 287–292.

[33] Leung, K.-C., Li, V. O. K., and Yang, D. An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges. *IEEE Trans. Parallel Distrib. Syst. 18*, 4 (2007), 522–535.

[34] Leverich, J., and Kozyrakis, C. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 2014 EuroSys Conference* (2014), pp. 4:1–4:14.

[35] Li, J., Agrawal, K., Elnikety, S., He, Y., Lee, I.-T. A., Lu, C., and McKinley, K. S. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2016), pp. 14:1–14:13.

[36] Li, J., Sharma, N. K., Ports, D. R. K., and Gribble, S. D. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC)* (2014), pp. 9:1–9:14.

[37] libevent. http://libevent.org/.

[38] libuv. http://libuv.org/.

[39] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)* (2014), pp. 429–444.

[40] Lu, Y., Xie, Q., Kliot, G., Geller, A., Larus, J. R., and Greenberg, A. G. Join-Idle-Queue: A novel load balancing algorithm for dynamically scalable web services. *Perform. Eval. 68*, 11 (2011), 1056–1071.

[41] Apache lucene. https://lucene.apache.org/.

[42] Marinos, I., Watson, R. N. M., and Handley, M. Network stack specialization for performance. In *Proceedings of the ACM SIGCOMM 2014 Conference* (2014), pp. 175–186.

[43] Memcached. https://memcached.org/.

[44] In-memory mongodb. https://docs.mongodb.com/manual/core/inmemory/.

[45] NANAVATI, M., WIRES, J., AND WARFIELD, A. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)* (2017), pp. 17–33.

[46] Nginx thread pool usage. https://www.nginx.com/blog/thread-pools-boost-performance-9x/.

[47] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)* (2013), pp. 385–398.

[48] OUSTERHOUT, J. K., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S. M., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst. 33*, 3 (2015), 7:1–7:55.

[49] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2013), pp. 69–84.

[50] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving network connection locality on multicore systems. In *Proceedings of the 2012 EuroSys Conference* (2012), pp. 337–350.

[51] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T. E., AND ROSCOE, T. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst. 33*, 4 (2016), 11:1–11:30.

[52] Pf_ring. http://www.ntop.org/products/packet-capture/pf_ring/.

[53] PREKAS, G., BELAY, A., PRIMORAC, M., KLIMOVIC, A., GROSSMAN, S., KOGIAS, M., GÜTERMANN, B., KOZYRAKIS, C., AND BUGNION, E. IX Open-source version 1.0 – Deployment and Evaluation Guide. Tech. rep., EPFL Technical Report 218568, 2016.

[54] PREKAS, G., PRIMORAC, M., BELAY, A., KOZYRAKIS, C., AND BUGNION, E. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)* (2015), pp. 342–355.

[55] Redis. https://redis.io/.

[56] RIZZO, L. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)* (2012), pp. 101–112.

[57] Microsoft corp. receive side scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx.

[58] SCHROEDER, B., WIERMAN, A., AND HARCHOL-BALTER, M. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)* (2006).

[59] SCILLADB PROJECT. Seastar – high-performance service-application framework. https://github.com/scylladb/seastar/.

[60] Silo: Multicore in-memory storage engine. https://github.com/stephentu/silo.

[61] SOARES, L., AND STUMM, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th Symposium on Operating System Design and Implementation (OSDI)* (2010), pp. 33–46.

[62] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large DataBases (VLDB)* (2007), pp. 1150–1160.

[63] THEKKATH, C. A., NGUYEN, T. D., MOY, E., AND LAZOWSKA, E. D. Implementing Network Protocols at User Level. In *Proceedings of the ACM SIGCOMM 1993 Conference* (1993), pp. 64–73.

[64] TPC-C Benchmark. http://www.tpc.org/tpcc/, 2010.

[65] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (2013), pp. 18–32.

[66] UHLIG, R., NEIGER, G., RODGERS, D., SANTONI, A. L., MARTINS, F. C. M., ANDERSON, A. V., BENNETT, S. M., KÄGI, A., LEUNG, F. H., AND SMITH, L. Intel Virtualization Technology. *IEEE Computer 38*, 5 (2005), 48–56.

[67] Voltdb. https://www.voltdb.com/.

[68] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)* (2015), pp. 87–104.

[69] What'sapp 2m connections. https://https://blog.whatsapp.com/196/1-million-is-so-2011.

[70] WIERMAN, A., AND ZWART, B. Is Tail-Optimal Scheduling Possible? *Operations Research 60*, 5 (2012), 1249–1257.

[71] YANG, X., BLACKBURN, S. M., AND MCKINLEY, K. S. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)* (2016), pp. 309–322.

[72] YASUKATA, K., HONDA, M., SANTRY, D., AND EGGERT, L. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)* (2016), pp. 43–56.

[73] ZELDOVICH, N., YIP, A., DABEK, F., MORRIS, R., MAZIÈRES, D., AND KAASHOEK, M. F. Multiprocessor Support for Event-Driven Programs. In *USENIX Annual Technical Conference* (2003), pp. 239–252.

[74] ZHANG, H., DONG, M., AND CHEN, H. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologie (FAST)* (2016), pp. 167–180.

[75] Zygos kernel. https://github.com/ix-project/zygos, 2017.