Bridging the gap between dataplanes and commodity operating systems

THÈSE N^{O} 8673 (2018)

PRÉSENTÉE LE 1ER JUIN 2018
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DES SYSTÈMES DE CENTRES DE CALCUL
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Georgios PREKAS

acceptée sur proposition du jury:

Prof. W. Zwaenepoel, président du jury
Prof. E. Bugnion, directeur de thèse
Prof. A. Belay, rapporteur
Prof. R. Stutsman, rapporteur
Prof. J. Larus, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2018

Acknowledgements

First, I would like to thank my advisor, Edouard Bugnion. Ed has great technical knowledge and a profound understanding of computer systems. I was extremely lucky that I had the opportunity to work with him, understand how he thinks, how he approaches and solves problems. Most importantly, I am grateful that I had the chance to learn from him. He helped me see the big picture of problems and be more systematic when solving them. With his support and guidance, I improved not only my technical skills but also my interpersonal skills.

I would also like to thank the members of my thesis committee, Adam Belay, James Larus, Ryan Stutsman, and Willy Zwaenepoel both for finding the time to serve in it, as well as for their constructive comments.

I am also fortunate for my collaborations with many people on our various research efforts: Adam Belay, Adrien Ghosn, Ana Klimovic, Christos Kozyrakis, Marios Kogias, Mia Primorac, and Samuel Grossman. Without them, most of this research would not have been possible.

Special thanks to all the members of my lab and our neighboring lab: Adrien, Bogdan, David, Dmitrii, Jonas, Maggy, Marios, Mia, Sahand, Sam, Stanko, and Stuart. Their diverse set of interests made life at work always more interesting. I would like to thank all the people who have given me feedback on my research on numerous occasions including, Alex, Manos, and Matt.

I would like also to thank all my friends, both in Greece and in Switzerland. The first for encouraging me to go abroad for my PhD and the second for making life in Switzerland entertaining.

Of course, I would like to thank my family, especially my parents, who have always supported me during many years of studies. Their support has been invaluable for me and has significantly contributed to allowing me to reach this point in my life.

Last but certainly not least, I would like to thank Joëlle. In the past two years, Joëlle has had to endure my irregular work schedule and was constantly helping me to find the balance between life and work.

This research has been supported by a Google Graduate Research Fellowship.

Lausanne, 15 May 2018 G. P.

Abstract

The conventional wisdom is that aggressive networking requirements, such as high packet rates for small messages and μs-scale tail latency, are best addressed outside the kernel, in a user-level networking stack. In particular, dataplanes borrow design elements from network middleboxes to run tasks to completion in tight loops. In its basic form, the dataplane design leverages sweeping simplifications such as the elimination of any resource management and any task scheduling to improve throughput and lower latency. As a result, dataplanes perform best when the request rate is predictable (since there is no resource management) and the service time of each task has a low execution time and a low dispersion. On the other hand, they exhibit poor energy proportionality and workload consolidation, and suffer from head-of-line blocking.

This thesis proposes the introduction of resource management to dataplanes. Current dataplanes decrease latency by constantly polling for incoming network packets. This approach trades energy usage for latency. We argue that it is possible to introduce a control plane, which manages the resources in the most optimal way in terms of power usage without affecting the performance of the dataplane.

Additionally, this thesis proposes the introduction of scheduling to dataplanes. Current designs operate in a strict FIFO and run-to-completion manner. This method is effective only when the incoming request requires a minimal amount of processing in the order of a few microseconds. When the processing time of requests is (a) longer or (b) follows a distribution with higher dispersion, the transient load imbalances and head-of-line blocking deteriorate the performance of the dataplane. We claim that it is possible to introduce a scheduler to dataplanes, which routes requests to the appropriate core and effectively reduce the tail latency of the system while at the same time support a wider range of workloads.

Keywords: web-scale application, datacenter, scale-out, virtualization, networking, operating system, energy proportionality, resource management, scheduling, work stealing, work conservation, head-of-line blocking, dataplane

Résumé

Dans le milieu académique, il est communément accepté qu'afin de satisfaire des exigences élevées en termes de performance des réseaux, telles que de hauts débits pour des messages courts, ou une latence (tail-latency) à l'échelle de quelques microseconds, le stack réseau du noyau (kernel) doit être remplacé par des implémentations plus efficaces et spécialisées, au niveau utilisateur. En particulier, les dataplanes empruntent des éléments de design aux middleboxes réseaux et exécutent les tâches suivant un modèle « run-to-completion ». Dans leur forme la plus simple, les dataplanes reposent sur des simplifications telles que l'élimination de toutes gestions des ressources et de tout « scheduling » (ordonnancement) de tâches. Ces simplifications permettent de réduire la logique des dataplanes à une boucle d'exécution courte, et d'ainsi améliorer le débit tout en réduisant les latences. Cependant, ces simplifications radicales ne fonctionnent vraiment que lorsque les taux de requêtes sont prévisibles (dû au manque de gestion des ressources) et lorsque les temps d'exécution individuels des requêtes sont courts et homogènes. D'autre part, les dataplanes ont en général une mauvaise gestion de leur consommation énergétique (energy-proportionality), une mauvaise exploitation des ressources (workload consolidation) et sont sujets au « head-of-line blocking ».

Cette thèse propose de réintroduire une forme de gestion des ressources dans les dataplanes. Les implémentations actuelles reposent sur un « polling » constant afin de traiter au plus vite les paquets entrants et de réduire les temps de latence. Cette approche sacrifie la consommation énergétique au profit de meilleures performances. Nous soutenons qu'il est possible d'introduire un « control plane », responsable de la gestion optimale de la consommation d'énergie, sans pour autant impacter les performances du dataplane.

De plus, cette thèse propose d'ajouter une forme de scheduling aux dataplanes. Les implémentations existantes reposent sur un modèle "FIFO" et traitent, entièrement, tour-à-tour chaque paquet (run-to-completion). Cette méthode n'est efficace que lorsque les requêtes entrantes correspondent à des temps d'exécution faibles, de l'ordre de quelques micro-seconds. Lorsque les temps d'exécution sont (a) plus longs ou (b) suivent une distribution avec une dispersion plus élevée, les déséquilibres sporadiques en terme de charge de travail, ainsi que les éventuelles situations de head-of-line blocking, détériorent les performances des dataplanes. Nous affirmons qu'il est possible d'introduire un « scheduler » dans les dataplanes, responsable de la répartition des tâches parmi les différents cœurs du processeur, et d'ainsi réduire la tail-latency du système tout en supportant des charges de travail hétérogènes.

Mots clefs : application à l'échelle du Web, centre de données, scale-out, virtualisation, réseaux, système d'exploitation, proportionnalité énergétique, gestion des ressources, scheduling, work stealing, conservation du travail/workload conservation, head-of-line blocking, dataplane

Contents

Acknowledgements **i**

Abstract (English/Français) **iii**

List of figures **ix**

List of tables **xi**

1 Introduction **1**
- 1.1 Web-scale applications 1
  - 1.1.1 The tail at scale 2
  - 1.1.2 Microsecond computing 3
  - 1.1.3 Energy-proportional computing 4
- 1.2 Resource management for web-scale applications 4
- 1.3 Microsecond-scale scheduling for web-scale applications 5
- 1.4 Thesis Statement 6
- 1.5 Thesis Contributions 7
  - 1.5.1 Resource management control plane for dataplane operating systems . 7
  - 1.5.2 Work conserving scheduler for dataplane operating systems 8
- 1.6 Thesis Roadmap 9
- 1.7 Bibliographic Notes 9

2 Dataplane OS and resource management **11**
- 2.1 Introduction 11
- 2.2 Background and Motivation 14
  - 2.2.1 Challenges for Datacenter Applications 15
  - 2.2.2 The Hardware – OS Mismatch 15
  - 2.2.3 Alternative Approaches 16
- 2.3 IX Design Approach 17
- 2.4 IX Implementation 20
  - 2.4.1 Overview 20
  - 2.4.2 The IX Dataplane 22
  - 2.4.3 Dataplane API and Operation 23
  - 2.4.4 Multi-core Scalability 25
  - 2.4.5 Flow group migration 26

2.4.6 The IXCP Control Loop . . . 26
2.4.7 Security Model . . . 27
2.5 Evaluation of the dataplane . . . 28
2.5.1 Experimental Methodology . . . 28
2.5.2 Dataplane performance . . . 30
2.5.3 Pareto-Optimal Static Configurations . . . 36
2.6 Evaluation . . . 39
2.7 Discussion . . . 45
2.8 Related Work . . . 48
2.9 Conclusion . . . 49

3 Scheduling for dataplane OS **51**
3.1 Introduction . . . 51
3.2 Background . . . 54
3.2.1 Scaling remote procedure calls . . . 54
3.2.2 Kernel bypass and sweeping simplifications . . . 55
3.2.3 Just enough queuing theory . . . 55
3.3 Experimental Methodology . . . 58
3.3.1 Approach and metrics . . . 60
3.3.2 Experimental Environment . . . 60
3.3.3 Evaluated Systems . . . 61
3.3.4 Baseline results . . . 61
3.4 Design . . . 62
3.4.1 Requirements . . . 62
3.4.2 ZYGOS High-level Design . . . 63
3.4.3 Ordering semantics in multi-threaded applications . . . 65
3.4.4 Eliminating Head-of-Line Blocking . . . 65
3.4.5 Inter-processor Interrupts . . . 66
3.5 Implementation . . . 67
3.6 Evaluation . . . 68
3.6.1 Synthetic micro-benchmarks . . . 68
3.6.2 Overheads of ZYGOS on tiny tasks: `memcached` . . . 73
3.6.3 A real application: Silo running TPC-C . . . 73
3.7 Discussion: the impact of SLO on systems . . . 77
3.8 Related Work . . . 78
3.9 Conclusion . . . 79

4 Conclusion **81**
4.1 What we did . . . 81
4.2 Future work . . . 82

Bibliography **85**

Curriculum Vitae **97**

List of Figures

1.1 Dynamic resource controls with IX for a workload consolidation scenario with a latency-sensitive application (e.g., `memcached`) and a background batch task (e.g., analytics). The controller, `ixcp`, partitions cores among the applications and adjusts the processor's DVFS settings. 5

2.1 Protection and separation of control and data plane in IX. 19
2.2 Interleaving of protocol processing and application execution in the IX dataplane. 21
2.3 Flow-group Migration Algorithm 26
2.4 NetPIPE performance for varying message sizes and system software configurations. 29
2.5 Multi-core scalability and high connection churn for 10GbE and 4x10GbE setups. In (a), half steps indicate hyperthreads. 31
2.6 Connection scalability of IX. 32
2.7 Average and 99^{th} percentile latency as a function of throughput for the ETC and USR `memcached` workloads. 34
2.8 Pareto efficiency for energy proportionality and workload consolidation for IX. The Pareto efficiency is in red while the various static configurations are color-coded according to their distinctive characteristics. 37
2.9 Energy-proportionality comparison between the Pareto-optimal frontier considering only DVFS adjustments, and the full Pareto frontier considering core allocation, hyperthread allocations, and frequency. 37
2.10 Energy proportionality (left) and workload consolidation (right) for the *slope* pattern 40
2.11 Energy proportionality (left) and workload consolidation (right) for the *step* pattern 41
2.12 Energy proportionality (left) and workload consolidation (right) for the *sin+noise* pattern 42
2.13 99^{th} percentile latency as a function of throughput for USR workload from Fig. 2.7b, for different values of the batch bound B. 46

3.1 Queuing models for $n = 2$ processors. 55
3.2 Simulation results for the 99^{th} percentile tail latency for four service time distributions with $\bar{S} = 1$. 56

3.3 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. 59

3.4 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. 62

3.5 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. 66

3.6 99^{th} percentile tail latency according to throughput for three distributions with 10µs mean task granularity. The horizontal line corresponds to the SLO of $\leq 10 \times \bar{S}$. 69

3.7 99^{th} percentile tail latency according to throughput for three distributions with 25µs mean task granularity. The horizontal line corresponds to the SLO of $\leq 10 \times \bar{S}$. 70

3.8 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). 71

3.9 Normalized rate of stealing vs. throughput for exponential service time with mean 25 µs 72

3.10 99^{th} percentile tail latency vs. throughput for two `memcached` workloads for Linux, IX and, ZYGOS. 74

3.11 Complementary CDF of task execution time for Silo running the TPC-C benchmark under Linux. 75

3.12 99^{th} percentile end-to-end latency vs. throughput for Silo running the TPC-C benchmark 76

3.13 Comparison of IX (batch size 1 and 64) and ZYGOS for a deterministic service time of 10 µs and 2 different SLOs. 78

List of Tables

2.1 Lines of code (in thousands). 23
2.2 The IX dataplane system call and event condition API. 23
2.3 Unloaded latency and maximum RPS for a given service-level agreement for the `memcached` workloads ETC and USR. 36
2.4 Energy Proportionality and Consolidation gains. 43
2.5 Breakdown of flow group migration measured during the six benchmarks. 45

3.1 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 99^{th} percentile end-to-end latency to Silo's 99^{th} percentile service time (203µs). 76

1 Introduction

1.1 Web-scale applications

Contemporary computer applications operate on a datacenter-scale [6]. Web search, social networking, e-commerce platforms, and ad serving are examples of applications that fall in this category. In order to support this massive scale, these applications are designed, implemented, and deployed as datacenter-wide distributed applications. Researchers have identified three major problems for the next generation of web-scale applications:

- **Tail latency.** User studies at Amazon, Bing, and Google have shown that even a minor increase of end-user latency in the orders of a couple of milliseconds negatively affects revenue [78, 118]. End-user latency depends in part on the internal deployment of these applications. It turns out that some of them are internally organized as a set of micro-services, which communicate in various high fan-in / fan-out patterns. The nature of this type of communication reveals the importance of tail latency [25].

- **Microsecond computing.** Datacenter operators deploy such latency-critical applications in memory in order to avoid the increased latency of accessing secondary storage. In-memory applications operate in the order of microseconds, which means that most of the commodity operating systems abstractions and mechanisms, such as schedulers, threads, and interrupt driven I/O are currently inadequate in terms of performance [8].

- **Energy proportional computing.** Another important aspect of datacenter-scale latency critical applications is resource efficiency. Most datacenter operators run their equipment at a deliberately low CPU utilization [7, 27, 4, 111] in order to accommodate for user requests spikes and respect the service level objectives. Of course, this practice leads to higher capital expenses. At the same time, datacenter operators face higher operational expenses, because modern CPUs do not exhibit a proportional energy-load relationship, by default. It is an elaborate task to configure the CPU so that it is more energy proportional [7].

Researchers and datacenter operators have been increasingly interested over the last few years in network dataplanes. Network dataplanes replace the traditional networking stack of commodity operating systems with a specialized and optimized stack. The goal is to provide stricter guarantees for the tail latency and at the same time support serving requests which require a couple of microseconds each. Dataplanes take advantage of the fact that existing networking stacks that are integrated in the operating systems are evolving very slowly for reasons of backward compatibility and complexity, which renders them unsuitable for modern workloads and/or hardware [121].

This thesis extends dataplanes in two ways: energy efficiency and work scheduling. The energy efficiency improvements rely on the introduction of an external control plane agent, which observes key latency metrics of the dataplane and takes coarse-grain decision in the order of milliseconds. The work scheduling improvements rely on the design and implementation of a low overhead work stealing mechanism, which is able to operate in the order of microseconds. This mechanism provides work conservation and reduces head-of-line blocking. Both mechanisms increase the potential impact of dataplanes and simplify their adoption as an architecture for serving low-latency microsecond-scale workloads with energy efficiency.

The following subsections summarize the three key issues for the next generation of web-scale applications: §1.1.1 summarizes the tail at scale problem as described by Dean et al. [25], §1.1.2 summarizes the microsecond scale computing problem as described by Barroso et al. [8], and §1.1.3 summarizes the energy proportionality computing problem as described by Barroso et al. [7].

1.1.1 The tail at scale

The tail at scale problem describes the complexity of taming the tail of the latency distribution for interactive datacenter-scale applications. The basic assumption is that variability unavoidably exists in computer systems. Shared resources, daemons, global resource sharing, maintenance activities, queueing, power limits, garbage collection, and energy management can all contribute to variability.

The first class of actions to mitigate variability include differentiating service classes, keeping low-level queues short, reducing head-of-line blocking by splitting long requests into a series of short ones, managing and synchronizing background activities. Despite these efforts, variability in latency cannot be completely eliminated. Moreover, fanning out a request from a root to a number of leaf servers, which is necessary in order to scale an application, amplifies the latency variability of the root request.

The solution to the tail at scale problem takes advantages of the fact that systems are usually already designed with fault-tolerance in mind, thus most of the data and services are replicated. Hedged requests and tied requests rely on issuing the same request multiple times; when the fastest response arrives, the remaining requests are cancelled. These techniques lower tail latency because the source of latency is often not inherent in the request itself but rather due to other forms of interference and because factors that cause variability do not tend to affect multiple nodes at the same time.

In this thesis, we will discuss dataplanes, which have the potential to help with the tail at scale problem because they are designed primarily to reduce latency and jitter, thus also reducing tail latency. Moreover, they reduce the end-to-end latency of network communication, which reduces the cost of implementing the techniques mentioned above, such as the tied requests.

1.1.2 Microsecond computing

Existing hardware and software mechanisms are designed to mitigate latencies in the order of nanoseconds or milliseconds. Hardware techniques, such as prefetching, out-of-order execution, and branch prediction, can hide nanosecond-scale latencies. On the other hand, software is capable of dealing with millisecond latencies incurred by standard I/O devices, for example.

The emergence of new low-latency I/O, such as datacenter networking, raw flash devices, and non-volatile memory, makes microsecond operations much more common than before, as all these new I/O have (or are expected to have) latencies in the order of microseconds. Moreover, in-memory systems (such as RAMCloud [99]) also have latencies in the same range.

In order to utilize the full potential of this new hardware, system designers need to rethink the software stack. An RDMA operation can have a latency of 2 µs but will take more than 50 µs if it is used with a feature-filled RPC stack or even TCP/IP. The same behavior will be observed if a raw flash device is used with the standard storage subsystem of any operating system.

Another important aspect when dealing with microsecond-scale computing is the absolute need to reduce overheads, such as interrupts, data copies, and context switches. It is obvious that when the service time is short, an overhead of a few microseconds can lead to a reduction of the overall system efficiency.

Finally, it is worth noting that the system design for datacenter-scale computing systems must solve the microsecond computing challenge while at the same time optimize for high utilization in order to reduce the total cost of ownership of the equipment.

As the most interesting new datacenter technologies start to operate at that time scale, dataplanes become increasingly crucial as a component that can deal with these technologies. Dataplanes have been designed from the ground up in order to provide a microsecond-optimized system stack. Their layers and abstractions are tailored to the new workloads and their time scales. In this thesis, we will demonstrate how dataplanes can deal with a wide range of microsecond computations and at the same time remain resource efficient.

1.1.3 Energy-proportional computing

A major effort in system design for datacenters is the case for energy-proportional computing. Energy efficiency has always been a major driver in the mobile and embedded computing areas where it maximized the battery life of devices. Lately, it is gaining significance also in general-purpose computing, because thermal constraints limit further CPU performance improvements.

Additionally, energy management is a key issue for datacenters because of cost and environmental reasons. Servers in datacenters cannot be completely turned off because each server holds a part of the datacenter's replicated data. Moreover, servers operate most of the time at between 10 and 50 percent of their maximum utilization [7, 27, 4, 111]. This behavior is by design so that servers can meet throughput and latency service level objectives even in the presence of load spikes and unexpected software or hardware events. Unfortunately, this range of utilization corresponds to the lowest energy-efficiency region of the server; essentially, a server still consumes about half of its full power when doing virtually no work.

CPU hardware improvements may improve this situation by widening the dynamic power range of the CPU. At the same time, software is responsible to use the hardware in the most energy-efficient way.

A naive implementation of a dataplane suffers from poor energy-proportionality because of constantly polling for network packets even at low load. In this thesis, we will demonstrate the design and implementation of a control agent to monitor load and adjust the resource allocation to the dataplane in the most energy efficient way.

1.2 Resource management for web-scale applications

It is obvious that managing and minimizing energy consumption is necessary to sustain a datacenter-scale application [7]. Datacenter operators prefer to run fully utilized servers in order to minimize their capital expenses. In addition, they would rather improve the proportionality of load vs. power to cut down on operational expenses. There are two goals towards this direction: (a) *energy proportionality*, which minimizes the energy consumed to deliver a workload and (b) *workload consolidation*, which raises server utilization and minimizes the number of servers needed for a set of workloads.

As we have seen before, a parallel trend in datacenter application is the effort to guarantee strict microscale-scale response latencies. This is necessary in order to support the large fan-out patterns that exist in modern datacenter applications, which are deployed as a set of microservices. Such latency-sensitive services are challenging to run in the presence of concurrent tasks on the same server, thus preventing workload consolidation. Additionally, these services must be able to respond to load spikes, so datacenter operators deploy them on

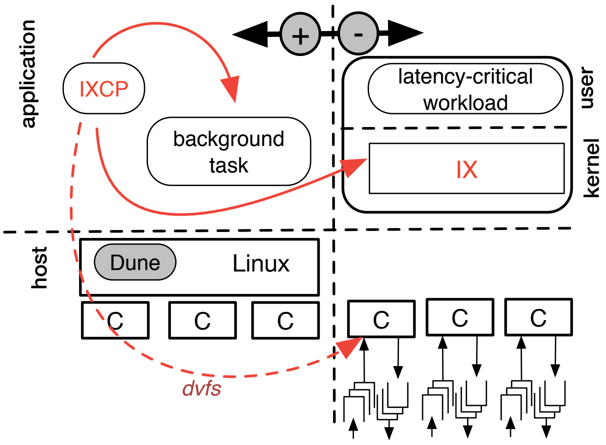


Figure 1.1 – Dynamic resource controls with IX for a workload consolidation scenario with a latency-sensitive application (e.g., `memcached`) and a background batch task (e.g., analytics). The controller, `ixcp`, partitions cores among the applications and adjusts the processor's DVFS settings.

dedicated servers running at low utilization. We can understand that it is difficult to combine the need for minimizing energy consumption and the trend for microsecond-scale computing.

To understand the challenges for resource management for latency-critical services, we performed a broad set of experiments under various configurations, such as core allocations, CPU frequency, use of hyperthreads, and existence of background tasks. Based on the results of these experiments and the Pareto methodology, we derived optimal strategies for achieving energy proportionality and/or workload consolidation. We integrate those strategies in IX, a state-of-the-art dataplane operating system that optimizes both throughput and latency for latency-critical workloads [13].

Fig. 1.1 illustrates our approach: the dynamic controller (`ixcp`) adjusts the number of cores allocated to a latency-sensitive application running on top of IX and the DVFS settings for these cores. The remaining cores can be placed in idle modes to reduce power consumption or can be safely used to run background tasks.

In §2.4.5 and §2.4.6, we introduce the design and the implementation of the control plane for IX. We present the methodology and the results of the exhaustive analysis of the configurations in §2.5.3. And, finally, in §2.6, we present the evaluation of the control plane.

1.3 Microsecond-scale scheduling for web-scale applications

Conventional operating system abstractions and mechanisms, such as schedulers, threads, and interrupt driven I/O, are not suitable for microsecond-scale microservices for performance reasons [8]. This observation led many researchers and engineers to develop solutions that bypass the operating system in order to achieve maximum performance [13, 103, 77, 57, 84, 59].

Most of these kernel-bypass approaches abandon the complicated schedulers [19] in favor for a much simpler execution model that involves polling, run-to-completion, and synchronization-free, flow-consistent mapping of requests to cores e.g., via RSS [90], or similar hardware mechanism.

The performance of this dataplane model relies on the elimination of system overheads that traditional operating systems face because of their complicated abstractions and layers. When the service time of the application is comparable to these overheads (e.g., for key-value stores), then the dataplane model improves throughput substantially (by up to 6× [13]). The limitations of this model appear for different applications or workloads where the service time is (a) either higher or (b) follows a distribution with high dispersion. In these cases, the lack of a proper scheduling mechanism reduces (and in some cases completely eliminates) the performance benefit of dataplanes vs. conventional operating systems.

The theoretical justification 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 [143]. Traditional operating systems follow more closely the first paradigm, while dataplanes adhere to the second one. Unfortunately, this leads to two inefficiencies: (a) the dataplane is not a work conserving scheduler, i.e., a core may be idle while there are pending requests, and (b) it suffers from head-of-line blocking, i.e., a request may be blocked until the previous tasks complete execution.

In chapter 3, we present ZYGOS, 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.

1.4 Thesis Statement

The “dataplane operating system” approach bypasses general-purpose operating systems and rely on sweeping simplifications such as the use of polling, run-to-completion, coherency-free execution, and in general the elimination of all forms of scheduling to increase throughput and/or reduce tail latency in a narrow set of conditions. These sweeping simplifications come with multiple hidden tradeoffs, such as loss of energy proportionality or head of line blocking, which limit their current applicability to a narrow set of workloads that consist of extremely small tasks with low dispersion of task service time.

This thesis demonstrates that energy management, resource allocation and request scheduling can be reintroduced within dataplane operating systems in order to eliminate these hidden tradeoffs. Energy proportionality (and workload consolidation) require a sophisticated control plane that interacts with the dataplane. Scheduling and elimination of head-of-line blocking *require the design and implementation, within the dataplane, of a work-conserving scheduler suitable for microsecond-scale network tasks.*

1.5 Thesis Contributions

This thesis makes the following key contributions:

1.5.1 Resource management control plane for dataplane operating systems

- We design and develop an external agent (control plane) which manages CPU resources (cores and frequency) while running a dataplane operating system. Certain metrics are exported in real time from the dataplane to the control plane. Then, the control plane aggregates the metrics and employs a control loop to make decisions regarding the CPU frequency and the number of dedicated CPU cores for the dataplane. The operator can configure the control plane in two modes: (a) maximum power efficiency and (b) maximum work consolidation. Under the maximum power efficiency mode, the control plane will use the minimum number of CPU cores and the minimum CPU frequency as required by the real time load of the dataplane without violating a defined service level objective (SLO). Under the maximum work consolidation mode, the control plane will schedule a configured background process to execute on as few cores as possible without violating the SLO of the dataplane.

- We develop techniques for fine-grain resource management for latency-critical workloads. This includes mechanisms for detection of load changes in sub-second timescales and for rebalancing flow-groups between cores without causing packet drops or reordered deliveries. In our experiments, this mechanism completes in less than 2 ms 95% of the time, and in at most 3.5 ms.

- We provide a methodology that uses the Pareto frontier of a set of static configurations to derive resource allocation policies. We derive two policies for `memcached` that respect the SLO constraints of a latency-critical in-memory key-value store. These policies lead to 42%–51% energy savings for a variety of load patterns, and enable workload consolidation with background jobs executing at 31%–44% of their peak throughput on a standalone machine. These gains are close to the Pareto-optimal bounds for the server used, within 91% and 81%–92% respectively.

- We demonstrate that precise and fine-grain control of cores, hyperthreads, and DVFS has a huge impact on both energy proportionality and workload consolidation, especially when load is highly variable. DVFS is a necessary but insufficient mechanism to control latency-critical applications that rely on polling.

1.5.2 Work conserving scheduler for dataplane operating systems

- We design and implement a scheduler for a dataplane operating system. Originally, each CPU core of a dataplane process requests independently from the other cores in order to minimize contention and cache-coherence traffic. This operation model resembles the *nxM/G/1* queueing model, which is inferior to the *M/G/n* model according to queueing theory. Based on this theoretical observation, we designed a work stealing system where idle CPU cores steal connections from busy CPU cores. Additionally, we addressed the head-of-line blocking problem, which happens when a short network request follows a long network request. To mitigate this situation, we used inter-processor interrupts to notify a busy CPU core that it must perform a minimal amount of network processing in order to allow idle cores to steal requests from its queue.

- We design a new system (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 [22]. The shuffle layer eliminates head-of-line-blocking while also offering strong ordering semantics of events associated with the same connection.

- We implement 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 [11] and handles IPIs in an exit-less manner similar to ELI [42].

- We develop a methodology using microbenchmarks with synthetic service times to identify system overheads as a function of task duration 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 (§3.6.1).

- We compare ZYGOS to IX, a state-of-the-art dataplane with strict run-to-completion that partitions flows onto cores [13]. 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 (§3.6.1). IX does outperform ZYGOS for workloads with very small task durations such as `memcached`. The difference is primarily due to IX's adaptive bounded batching, which is not currently supported in ZYGOS. (§3.6.2)

- 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 [130], 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 (§3.6.3).

1.6 Thesis Roadmap

This thesis is organized as follows:

- Chapter 2 presents the design, implementation, and evaluation of the IX dataplane operating system. Additionally, it includes a study of the impact of various power management features of a modern CPU to the performance of a dataplane operating system in terms of throughput and latency. Finally, it introduces the resource management control plane for the IX dataplane operating system.
- Chapter 3 includes a detailed analysis of the throughput and latency performance of various queueing models and various service time distributions. Additionally, it introduces a methodology to identify system overheads as a function of task duration for different systems. Finally, it describes the design and implementation of ZYGOS, which extends the IX dataplane operating system with a work conserving scheduler.
- Chapter 4 concludes the thesis and presents future directions.

1.7 Bibliographic Notes

Portions of this thesis are based on the work I have previously published with my advisor and my colleagues. Chapter 2 is based on a journal article published in the ACM Transactions on Computer Systems (TOCS) in 2017 [13]; the article itself is based on a conference paper published in the Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI) in 2014 [12] and a conference paper published in the Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC) in 2015 [107]. Chapter 3 is based on a conference paper published in the Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP) in 2016 [106].

It is important to note that from chapter 2 the contribution of this thesis is the Pareto analysis, and the design, implementation, and evaluation of the control plane. From chapter 3 this thesis' contribution does not include §3.2.3.

2 Dataplane OS and resource management

The conventional wisdom is that aggressive networking requirements, such as high packet rates for small messages and µs-scale tail latency, are best addressed outside the kernel, in a user-level networking stack. We present IX, a dataplane operating system that provides high I/O performance and high resource efficiency while maintaining the protection and isolation benefits of existing kernels.

IX uses hardware virtualization to separate management and scheduling functions of the kernel (control plane) from network processing (dataplane). The dataplane architecture builds upon a native, zero-copy API and optimizes for both bandwidth and latency by dedicating hardware threads and networking queues to dataplane instances, processing bounded batches of packets to completion, and by eliminating coherence traffic and multi-core synchronization. The control plane dynamically adjusts core allocations and voltage/frequency settings to meet service-level objectives.

We demonstrate that IX outperforms Linux and a user-space network stack significantly in both throughput and end-to-end latency. Moreover, IX improves the throughput of a widely deployed, key-value store by up to 6.1× and reduces tail latency by more than 1.9×. With three varying load patterns, the control plane saves 42%–51% of processor energy, and it allows background jobs to run at 31%–44% of their standalone throughput.

2.1 Introduction

Datacenter applications have evolved with the advent of web-scale services. User-facing, large-scale applications such as search, social networking, and e-commerce now rely extensively on high fan-out patterns between low-latency services. Such services exhibit low per-request service times (a handful of µs for a key-value store), have strict service-level objectives (SLO, e.g. $< 500\mu s$ at the 99^{th} percentile), and must sustain massive request rates for short messages with high client fan-in connection counts and churn [3, 25, 98].

The conventional wisdom is that there is a basic mismatch between these requirements and existing networking stacks in commodity operating systems. To address the performance concern, some systems bypass the kernel and implement the networking stack in user-space [57, 59, 84, 125, 128]. While kernel bypass eliminates privilege level crossing overheads, on its own it does not eliminate the difficult tradeoffs between high packet rates and low latency (see §2.5.2). Moreover, user-level networking suffers from lack of protection. Application bugs and crashes can corrupt the networking stack and impact other workloads. Other systems go a step further by also replacing TCP/IP with RDMA in order to offload network processing to specialized adapters [31, 58, 91, 99]. However, such adapters must be present at both ends of the connection and can only be used within the datacenter.

Such latency-critical services are also challenging to run in a shared infrastructure environment. They are particularly sensitive to resource allocation and frequency settings, and they suffer frequent tail latency violations when common power management or consolidation approaches are used [71, 74]. As a result, operators typically deploy them on dedicated servers running in polling mode, forgoing opportunities for workload consolidation and reduced power consumption at below-peak utilization levels. Since these services are deployed on thousands of servers in large-scale datacenters, this deployment practice represents a huge waster in resource use.

Ideally, we want these services to achieve *energy proportionality*, so that their energy consumption scales with observed load [7, 79]. Hardware enhancements, primarily in dynamic voltage/frequency scaling (DVFS) and idle modes in modern processors [62, 114] provide a foundation for energy proportionality. Moreover, we want these services to allow for *workload consolidation*, so that any spare resources during periods of low load can be used by workloads such as background analytics in order to raise server utilization [136, 135]. The two goals map to distinct economic objectives: energy proportionality reduces operational expenses (*opex*), whereas workload consolidation reduces capital expenses (*capex*). Since capital costs often dominate the datacenter's total cost of ownership (*TCO*), consolidation is highly desirable. Nevertheless, it is not always possible, e.g., when one application consumes the entirety of a given resource, e.g., memory. In such cases, energy proportionality is a necessity.

We propose IX, an operating system designed to break the 4-way tradeoff between high throughput, low latency, strong protection, and resource efficiency. Its architecture builds upon lessons from high performance middleboxes, such as firewalls, load-balancers, and software routers [29, 66]. IX separates the control plane, which is responsible for system configuration and coarse-grain resource provisioning between applications, from the dataplanes, which run the networking stack and application logic. IX leverages Dune and virtualization hardware to run the dataplane kernel and the application at distinct protection levels and to isolate the control plane from the dataplane [11]. In our implementation, the control plane leverages mechanisms of the full Linux kernel to dynamically reallocate resources, and the dataplanes run as protected, library-based operating systems on dedicated hardware threads.

The IX dataplane allows for networking stacks that optimize for both bandwidth and latency. It is designed around a native, zero-copy API that supports processing of bounded batches of packets to completion. Each dataplane executes all network processing stages for a batch of packets in the dataplane kernel, followed by the associated application processing in user mode. This approach amortizes API overheads and improves both instruction and data locality. We set the batch size adaptively based on load. The IX dataplane also optimizes for multi-core scalability. The network adapters (NICs) perform flow-consistent hashing of incoming traffic to distinct queues. Each dataplane instance exclusively controls a set of these queues and runs the networking stack and a single application without the need for synchronization or coherence traffic during common case operation. The IX API departs from the POSIX API, and its design is guided by the commutativity rule [22]. However, the `libix` user-level library includes an event-based API similar to the popular `libevent` library [108], providing compatibility with a wide range of existing applications.

The core of the IX control plane is a dynamic controller that adjusts the number of cores allocated to a latency-sensitive application running on top of IX and the DVFS settings for these cores. The remaining cores can be placed in idle modes to reduce power consumption or can be safely used to run background tasks. The controller builds upon two key mechanisms. The first mechanism detects backlog and increases in queuing delays that exceed the allowable upper bound for the specific latency-critical application. It monitors CPU utilization and signals required adjustments in resource allocation. The second mechanism, implemented in coordination with the dataplane, quickly migrates both network and application processing between cores transparently and without dropping or reordering packets.

To evaluate the dataplane, we compare IX with a TCP/IP dataplane against Linux 4.8 and mTCP, a state-of-the-art user-level TCP/IP stack [57]. On a 10GbE experiment using short messages, IX outperforms Linux and mTCP by up to 6.3× and 1.8× respectively for throughput. IX further scales to a 4x10GbE configuration using a single multi-core socket. The unloaded uni-directional latency for two IX servers is 5.8μs, which is 2.7× better than between standard Linux kernels and an order of magnitude better than mTCP, as both trade-off latency for throughput. Our evaluation with `memcached`, a widely deployed key-value store, shows that IX improves upon Linux by up to 6.1× in terms of throughput at a given 99^{th} percentile latency bound, as it can reduce kernel time, due essentially to network processing, from ~ 80% with Linux to 60% with IX.

Before evaluating the control plane, we performed an exhaustive analysis of static configurations for a latency-critical service (`memcached` [89]) running on a modern server to gain a principled understanding of the challenges for resource management in the presence of latency-critical services. We explored up to 224 possible settings for core allocation, use of hyperthreads, DVFS frequencies, and Turbo Boost. While our experiments use a single application, the implications have broad applicability because `memcached` has aggressive latency requirements, short service times, and a large number of independent clients that are common among many latency-critical applications. Our experiments reveal that there is an inherent tradeoff for any given static configuration between the maximum throughput and the overall efficiency when operating below peak load. Furthermore, the experiments reveal a Pareto-optimal frontier in the efficiency of static configurations at any given load level, which allows for close to linear improvements in energy-proportionality and workload consolidation factors.

We then evaluated our control plane with two control policies that optimize for energy proportionality and workload consolidation, respectively. A policy determines how resources (cores, hyperthreads, and DVFS settings) are adjusted to reduce underutilization or to restore violated SLO. The two policies are derived from the exhaustive analysis of the 224 static configurations. For the platform studied (a Xeon E5-2665), we conclude that: for *best energy proportionality*, (i) we start with the lowest clock rate and allocate additional cores to the latency-critical task as its load grows, using at first only one hyperthread per core; (ii) we enable the second hyperthread only when all cores are in use; and finally (iii) we increase the clock rate for the cores running the latency-critical task. For *best consolidation*, (i) we start at the nominal clock rate and add cores with both hyperthreads enabled as load increases; and (ii) finally enable Turbo Boost as a last resort.

IX demonstrates that, by revisiting networking APIs and taking advantage of modern NICs and multi-core chips, we can design systems that achieve high throughput, low latency, robust protection, and resource efficiency. It also shows that, by separating the small subset of performance-critical I/O functions from the rest of the kernel, we can architect radically different I/O systems and achieve large performance gains, while retaining compatibility with the huge set of APIs and services provided by a modern OS like Linux. Finally, we also demonstrate that latency-sensitive applications can be deployed efficiently through dynamic resource allocation policies that target a specific tail latency.

This chapter contains the research contributions of two conference papers that focus on the dataplane [12] and the control plane [107], respectively. The evaluation results presented in this chapter have been reproduced with IX v.1.0, which is available in open-source [55]. A corresponding technical report provides detailed instructions to reproduce all the results of this chapter [105].

The rest of the chapter is organized as follows. §2.2 motivates the need for a new OS architecture. §2.3 and §2.4 present the design principles and implementation of IX. §2.5 presents the quantitative evaluation. §2.7 and §2.8 discuss open issues and related work. We conclude in §2.9.

2.2 Background and Motivation

Our work focuses on improving operating systems for applications with aggressive networking requirements running on multi-core servers.

2.2.1 Challenges for Datacenter Applications

Large-scale, datacenter applications pose unique challenges to system software and their networking stacks:

Microsecond tail latency To enable rich interactions between a large number of services without impacting the overall latency experienced by the user, it is essential to reduce the latency for some service requests to a few tens of µs [5, 115]. Because each user request often involves hundreds of servers, we must also consider the long tail of the latency distributions of RPC requests across the datacenter [25]. Although tail-tolerance is actually an end-to-end challenge, the system software stack plays a significant role in exacerbating the problem [71]. Overall, each service node must ideally provide tight bounds on the 99^{th} percentile request latency.

High packet rates The requests and, often times, the replies between the various services that comprise a datacenter application are quite small. In Facebook's `memcached` service, for example, the vast majority of requests use keys shorter than 50 bytes and involve values shorter than 500 bytes [3], and each node can scale to serve millions of requests per second [98].

The high packet rate must also be sustainable under a large number of concurrent connections and high connection churn [43]. If the system software cannot handle large connection counts, there can be significant implications for applications. The large connection count between application and `memcached` servers at Facebook made it impractical to use TCP sockets between these two tiers, resulting in deployments that use UDP datagrams for `get` operations and an aggregation proxy for `put` operations [98].

Protections Since multiple services commonly share servers in both public and private datacenters [25, 49, 119], there is need for isolation between applications. The use of kernel-based or hypervisor-based networking stacks largely addresses the problem. A trusted network stack can firewall applications, enforce access control lists (ACLs), and implement limiters and other policies based on bandwidth metering.

Resource efficiency The load of datacenter applications varies significantly due to diurnal patterns and spikes in user traffic. Ideally, each service node will use the fewest resources (cores, memory, or IOPS) needed to satisfy packet rate and tail latency requirements at any point. Unfortunately, classic operating system schedulers are ill-matched to ensure tail control [71, 74]. Novel dynamic resource management mechanisms and policies are required to improve energy proportionality and workload consolidation in the presence of latency-sensitive applications [79, 80, 73].

2.2.2 The Hardware – OS Mismatch

The wealth of hardware resources in modern servers should allow for low latency and high packet rates for datacenter applications. A typical server includes one or two processor sockets, each with eight or more multithreaded cores and multiple, high-speed channels to DRAM and PCIe devices. Solid-state drives and PCIe-based Flash storage are also increasingly popular. For networking, 10 GbE NICs and switches are widely deployed in datacenters, with 40 GbE and 100 GbE technologies right around the corner. The combination of tens of hardware threads and 10 GbE NICs should allow for rates of 15M packets/sec with minimum sized packets. We should also achieve 10–20µs round-trip latencies given 3µs latency across a pair of 10 GbE NICs, one to five switch crossings with cut-through latencies of a few hundred ns each, and propagation delays of 500ns for 100 meters of distance within a datacenter.

Unfortunately, commodity operating systems have been designed under very different hardware assumptions. Kernel schedulers, networking APIs, and network stacks are based on an assumption of multiple applications sharing a single processing core and packet inter-arrival times being many times higher than the latency of interrupts and system calls. As a result, such operating systems trade off both latency and throughput in favor of fine-grain resource scheduling. Interrupt coalescing (used to reduce processing overheads), queuing latency due to device driver processing intervals, the use of intermediate buffering, and CPU scheduling delays frequently add up to several hundred µs of latency to remote requests. The overheads of buffering and synchronization needed to support flexible, fine-grain scheduling of applications to cores increases CPU and memory system overheads, which limits throughput. As requests between service tiers of datacenter applications often consist of small packets, common NIC hardware optimizations, such as TCP segmentation and receive side coalescing, have a marginal impact on packet rate.

2.2.3 Alternative Approaches

Since the network stacks within commodity kernels cannot take advantage of the abundance of hardware resources, a number of alternative approaches have been suggested. Each alternative addresses a subset, but not all of the requirements for datacenter applications.

User-space networking stacks Systems such as OpenOnload [125], mTCP [57], and Sandstorm [84] run the entire networking stack in user-space in order to eliminate kernel crossing overheads and optimize packet processing without incurring the complexity of kernel modifications. However, there are still tradeoffs between packet rate and latency. For instance, mTCP uses dedicated threads for the TCP stack, which communicate at relatively coarse granularity with application threads. This aggressive batching amortizes switching overheads at the expense of higher latency (see §2.5). It also complicates resource sharing as the network stack must use a large number of hardware threads regardless of the actual load. More importantly, security tradeoffs emerge when networking is lifted into the user-space and application bugs can corrupt the networking stack. For example, an attacker may be able to transmit raw packets (a capability that normally requires root privileges) to exploit weaknesses in network protocols and impact other services [14]. It is difficult to enforce any security or metering policies beyond what is directly supported by the NIC hardware.

Alternatives to TCP In addition to kernel bypass, some low-latency object stores rely on RDMA to offload protocol processing on dedicated Infiniband host channel adapters [31, 58, 91, 99]. RDMA can reduce latency, but requires that specialized adapters be present at both ends of the connection. Using commodity Ethernet networking, Facebook's `memcached` deployment uses UDP to avoid connection scalability limitations [98]. Even though UDP is running in the kernel, reliable communication and congestion management are entrusted to applications.

Alternatives to POSIX API MegaPipe replaces the POSIX API with lightweight sockets implemented with in-memory command rings [45]. This reduces some software overheads and increases packet rates, but retains all other challenges of using an existing, kernel-based networking stack.

OS enhancements Tuning kernel-based stacks provides incremental benefits with superior ease of deployment. Linux `SO_REUSEPORT` allows multi-threaded applications to accept incoming connections in parallel. Affinity-accept reduces overheads by ensuring all processing for a network flow is affinitized to the same core [102]. Recent Linux Kernels support a busy polling driver mode that trades increased CPU utilization for reduced latency [52], but it is not yet compatible with `epoll`. When microsecond latencies are irrelevant, properly tuned stacks can maintain millions of open connections [141].

2.3 IX Design Approach

The first two requirements in §2.2.1 — microsecond latency and high packet rates — are not unique to datacenter applications. These requirements have been addressed in the design of middleboxes such as firewalls, load-balancers, and software routers [29, 66] by integrating the networking stack and the application into a single *dataplane*. The two remaining requirements — protection and resource efficiency — are not addressed in middleboxes because they are single-purpose systems, not exposed directly to users.

Many middlebox dataplanes adopt design principles that differ from traditional OSes. First, they *run each packet to completion*. All network protocol and application processing for a packet is done before moving on to the next packet, and application logic is typically intermingled with the networking stack without any isolation. By contrast, a commodity OS decouples protocol processing from the application itself in order to provide scheduling and flow control flexibility. For example, the kernel relies on device and soft interrupts to context switch from applications to protocol processing. Similarly, the kernel's network stack will generate TCP ACKs and slide its receive window even when the application is not consuming data, up to an extent. Second, middlebox dataplanes optimize for *synchronization-free operation* in order to scale well on many cores. Network flows are distributed into distinct queues via flow-consistent hashing and common case packet processing requires no synchronization or coherence traffic between cores. By contrast, commodity OSes tend to rely heavily on coherence traffic and are structured to make frequent use of locks and other forms of synchronization.

IX extends the dataplane architecture to support untrusted, general-purpose applications and satisfy all requirements in §2.2.1. Its design is based on the following key principles:

Separation and protection of control and data plane IX separates the control function of the kernel, responsible for resource configuration, provisioning, scheduling, and monitoring, from the dataplane, which runs the networking stack and application logic. Like a conventional OS, the control plane multiplexes and schedules resources among dataplanes, but in a coarse-grained manner in space and time. Entire cores are dedicated to dataplanes, memory is allocated at large page granularity, and NIC queues are assigned to dataplane cores. The control plane is also responsible for elastically adjusting the allocation of resources between dataplanes.

The separation of control and data plane also allows us to consider radically different I/O APIs, while permitting other OS functionality, such as file system support, to be passed through to the control plane for compatibility. Similar to the Exokernel [35], each dataplane runs a single application in a single address space. However, we use modern virtualization hardware to provide three-way isolation between the control plane, the dataplane, and untrusted user code [11]. Dataplanes have capabilities similar to guest OSes in virtualized systems. They manage their own address translations, on top of the address space provided by the control plane, and can protect the networking stack from untrusted application logic through the use of privilege rings. Moreover, dataplanes are given direct pass-through access to NIC queues through memory mapped I/O.

Run to completion with adaptive batching IX dataplanes run to completion all stages needed to receive and transmit a packet, interleaving protocol processing (kernel mode) and application logic (user mode) at well-defined transition points. Hence, there is no need for intermediate buffering between protocol stages or between application logic and the networking stack. Unlike previous work that applied a similar approach to eliminate receive livelocks during congestion periods [92], IX uses run to completion during all load conditions. Thus, we are able to use polling and avoid interrupt overhead in the common case by dedicating cores to the dataplane. We still rely on interrupts as a mechanism to regain control, for example, if application logic is slow to respond. Run to completion improves both message throughput and latency because successive stages tend to access many of the same data, leading to better data cache locality.

The IX dataplane also makes extensive use of batching. Previous systems applied batching at the system call boundary [45, 124] and at the network API and hardware queue level [57]. We apply batching in every stage of the network stack, including but not limited to system calls and queues. Moreover, we use batching *adaptively* as follows: (i) we never wait to batch requests and batching only occurs in the presence of congestion; (ii) we set an upper bound on the number of batched packets. Using batching only on congestion allows us to minimize the impact on latency, while bounding the batch size prevents the live set from exceeding cache capacities and avoids transmit queue starvation. Batching improves packet

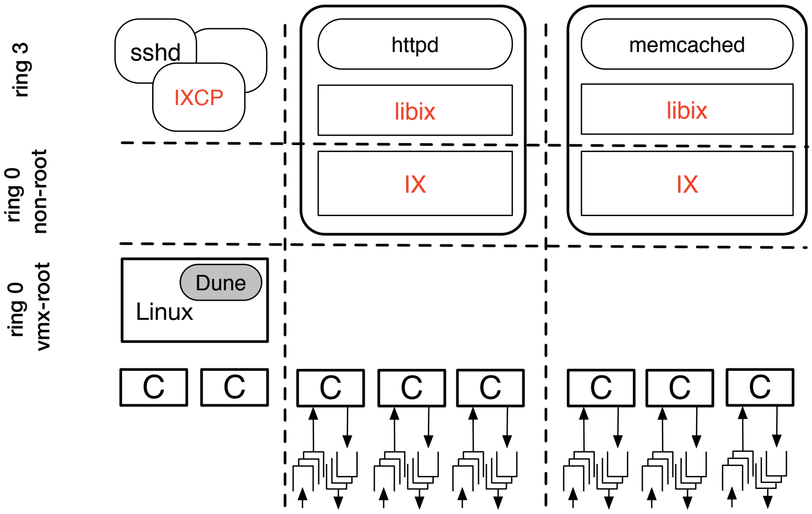


Figure 2.1 – Protection and separation of control and data plane in IX.

rate because it amortizes system call transition overheads and improves instruction cache locality, prefetching effectiveness, and branch prediction accuracy. When applied adaptively, batching also decreases latency because these same efficiencies reduce head-of-line blocking.

The combination of bounded, adaptive batching and run to completion means that queues for incoming packets can build up only at the NIC edge, before packet processing starts in the dataplane. The networking stack sends acknowledgments to peers only as fast as the application can process them. Any slowdown in the application-processing rate quickly leads to shrinking windows in peers. The dataplane can also monitor queue depths at the NIC edge and signal the control plane to allocate additional resources for the dataplane (more hardware threads, increased clock frequency), notify peers explicitly about congestion (e.g., via ECN [109]), and make policy decisions for congestion management (e.g., via RED [40]).

Native, zero-copy API with explicit flow control We do not expose or emulate the POSIX API for networking. Instead, the dataplane kernel and the application communicate at coordinated transition points via messages stored in memory. Our API is designed for true zero-copy operation in both directions, improving both latency and packet rate. The dataplane and application cooperatively manage the message buffer pool. Incoming packets are mapped read-only into the application, which may hold onto message buffers and return them to the dataplane at a later point. The application sends to the dataplane scatter/gather lists of memory locations for transmission but, since contents are not copied, the application must keep the content immutable until the peer acknowledges reception. The dataplane enforces flow control correctness and may trim transmission requests that exceed the available size of the sliding window, but the application controls transmit buffering.

Flow consistent, synchronization-free processing We use multi-queue NICs with receive-side scaling (RSS [90]) to provide flow-consistent hashing of incoming traffic to distinct hardware queues. Each hardware thread (hyperthread) serves a single receive and transmit queue per NIC, eliminating the need for synchronization and coherence traffic between cores in the networking stack. Similarly, memory management is organized in distinct pools for each hardware thread. The absence of a POSIX socket API eliminates the issue of the shared file descriptor namespace in multithreaded applications [22]. Overall, the IX dataplane design scales well with the increasing number of cores in modern servers, which improves both packet rate and latency. This approach does not restrict the memory model for applications, which can take advantage of coherent, shared memory to exchange information and synchronize between cores.

TCP-friendly flow group migration The IX control plane establishes dynamically the mapping of RSS flow groups to queues to balance the traffic among the hardware threads. The IX dataplane implements the actual flow group migration and programs the NIC's RSS Redirection Table [53] to change the mappings. The implementation does not impact the steady state performance of the dataplane and its coherence-free design. The migration algorithm contains distinct phases that ensure that migration does not create network anomalies such as dropping packets or processing them out of order in the networking stack.

Dynamic control loop with user-defined policies At its core, the control plane has a control loop that monitors the queuing delay to detect likely SLO violations and reacts by adding system resources within milliseconds. It monitors the utilization of the IX dataplane to similarly remove unnecessary system resources. The IX control plane relies on the host Linux kernel mechanisms to adjust system resources such as changing the processor frequency or the number of cores allocated to the IX dataplane. It relies on the IX dataplane's TCP-friendly flow group migration mechanism to balance the load among the cores. Although the control loop specifies *when* resources must be adjusted, it does not specify *which* resource must be added or removed, as this policy decision is a function of the platform's characteristics, the application's ability to scale horizontally, and the overall objective (energy proportionality or workload consolidation).

2.4 IX Implementation

2.4.1 Overview

Fig. 2.1 presents the IX architecture, focusing on the separation between the control plane and the multiple dataplanes. The hardware environment is a multi-core server with one or more multi-queue NICs with RSS support. The IX control plane consists of the full Linux kernel and `IXCP`, a user-level program. The Linux kernel initializes PCIe devices, such as the NICs, and provides the basic mechanisms for resource allocation to the dataplanes, including cores, memory, and network queues. Equally important, Linux provides system calls and services that are necessary for compatibility with a wide range of applications, such as file system and

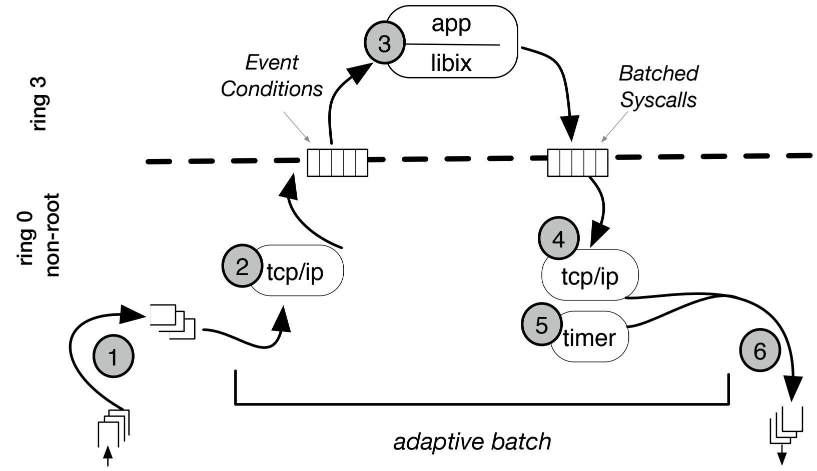


Figure 2.2 – Interleaving of protocol processing and application execution in the IX dataplane.

signal support. `IXCP` monitors resource usage and dataplane performance and implements resource allocation policies.

We run the Linux kernel in VMX root ring 0, the mode typically used to run hypervisors in virtualized systems [131]. We use the Dune module within Linux to enable dataplanes to run as application-specific OSes in VMX non-root ring 0, the mode typically used to run guest kernels in virtualized systems [11]. Applications run in VMX non-root ring 3, as usual. This approach provides dataplanes with direct access to hardware features, such as page tables and exceptions, and pass-through access to NICs. Moreover, it provides full, three-way protection between the control plane, dataplanes, and untrusted application code.

Each IX dataplane supports a single, multithreaded application. For instance, Fig. 2.1 shows one dataplane for a multi-threaded `memcached` server and another dataplane for a multi-threaded `httpd` server. The control plane allocates resources to each dataplane in a coarse-grained manner. Core allocation is controlled through real-time priorities and `cpusets`; memory is allocated in large pages; each NIC hardware queue is assigned to a single dataplane. This approach avoids the overheads and unpredictability of fine-grained time multiplexing of resources between demanding applications [71].

Each IX dataplane operates as a single address-space OS and supports two thread types within a shared, user-level address space: (i) *elastic threads* which interact with the IX dataplane to initiate and consume network I/O and (ii) *background threads*. Both elastic and background threads can issue arbitrary POSIX system calls that are intermediated and validated for security by the dataplane before being forwarded to the Linux kernel. Elastic threads are expected to *not* issue blocking calls because of the adverse impact on network behavior resulting from delayed packet processing. Each elastic thread makes exclusive use of a core or hardware thread allocated to the dataplane in order to achieve high performance with predictable latency. In contrast, multiple background threads may timeshare an allocated hardware thread. For example, if an application were allocated four hardware threads, it could use all of them as elastic threads to serve external requests or it could temporarily transition to three elastic threads and use one background thread to execute tasks such as garbage collection. When the control plane revokes or allocates an additional hardware thread using a protocol similar to the one in Exokernel [35], the dataplane adjusts its number of elastic threads.

2.4.2 The IX Dataplane

We now discuss the IX dataplane in more detail. It differs from a typical kernel in that it is specialized for high performance network I/O and runs only a single application, similar to a library OS but with memory isolation. However, our dataplane still provides many familiar kernel-level services.

For memory management, we accept some internal memory fragmentation in order to reduce complexity and improve efficiency. All hot-path data objects are allocated from per hardware thread memory pools. Each memory pool is structured as arrays of identically sized objects, provisioned in page-sized blocks. Free objects are tracked with a simple free list, and allocation routines are inlined directly into calling functions. *Mbufs*, the storage object for network packets, are stored as contiguous chunks of bookkeeping data and MTU-sized buffers, and are used for both receiving and transmitting packets.

The dataplane also manages its own virtual address translations, supported through nested paging. In contrast to contemporary OSes, it uses exclusively large pages (2MB). We favor large pages due to their reduced address translation overhead [9, 11] and the relative abundance of physical memory resources in modern servers. The dataplane maintains only a single address space; kernel pages are protected with supervisor bits. We deliberately chose not to support swappable memory in order to avoid adding performance variability.

We provide a hierarchical timing wheel implementation for managing network timeouts, such as TCP retransmissions [133]. It is optimized for the common case where most timers are canceled before they expire. We support extremely high-resolution timeouts, as low as 16 µs, which has been shown to improve performance during TCP incast congestion [134].

Our current IX dataplane implementation is based on Dune and requires the VT-x virtualization features available on Intel x86-64 systems [131]. However, it could be ported to any architecture with virtualization support, such as ARM, SPARC, and Power. It also requires one or more Intel 82599 chipset NICs, but it is designed to easily support additional drivers.

Table 2.1 lists the code size (in thousands of SLOC [142]). The rows correspond to the different protection domains of the system while the columns correspond to the different open-source projects involved. The TCP/IP stack uses a highly-modified version of lwIP [33]. We chose lwIP as a starting point for TCP/IP processing because of its modularity and its maturity as a RFC-compliant, feature-rich networking stack. We implemented our own RFC-compliant support for UDP, ARP, and ICMP. Since lwIP was optimized for memory efficiency in embedded

KSLOC	IX	lwIP	Dune	total
Control plane	0.4			0.4
Data plane	9.7	9.4	4.9	24.0
Linux kernel		2.5		2.5
User-level library	1.0			1.0

Table 2.1 – Lines of code (in thousands).

System Calls (batched)		
Type	**Parameters**	**Description**
connect	cookie, dst_IP, dst_port	Opens a connection
accept	handle, cookie	Accepts a connection
sendv	handle, scatter_gather_array	Transmits a scatter-gather array of data
recv_done	handle, bytes_acked	Advances the receive window and frees memory buffers
close	handle	Closes or rejects a connection

Event Conditions		
Type	**Parameters**	**Description**
knock	handle, src_IP, src_port	A remotely initiated connection was opened
connected	cookie, outcome	A locally initiated connection finished opening
recv	cookie, mbuf_ptr, mbuf_len	A message buffer was received
sent	cookie, bytes_sent, window_size	A send completed and/or the window size changed
dead	cookie, reason	A connection was terminated

Table 2.2 – The IX dataplane system call and event condition API.

environments, we had to radically change its internal data structures for multi-core scalability and fine-grained timer management. However, we did not yet optimize the lwIP code for performance. Hence, the results of §2.5 have room for improvement. In addition, the IX dataplane links with an unmodified DPDK library, which is used to initially configure the NIC. DPDK code is not used during datapath operations; instead, IX accesses NIC descriptor rings directly.

2.4.3 Dataplane API and Operation

The elastic threads of an application interact with the IX dataplane through three asynchronous, non-blocking mechanisms summarized in Table 2.2: they issue *batched systems calls* to the dataplane; they consume *event conditions* generated by the dataplane; and they have direct, but safe, access to mbufs containing incoming payloads. The latter allows for zero-copy access to incoming network traffic. The application can hold on to mbufs until it asks the dataplane to release them via the `recv_done` batched system call.

Both batched system calls and event conditions are passed through arrays of shared memory, managed by the user and the kernel respectively. IX provides an unbatched system call (`run_io`) that yields control to the kernel and initiates a new run to completion cycle. As part of the cycle, the kernel overwrites the array of batched system call requests with corresponding return codes and populates the array of event conditions. The handles defined in Table 2.2 are kernel-level flow identifiers. Each handle is associated with a cookie, an opaque value provided by the user at connection establishment to enable efficient user-level state lookup [45].

IX differs from POSIX sockets in that it directly exposes flow control conditions to the application. The `sendv` system call does not return the number of bytes buffered. Instead, it returns the number of bytes that were accepted and sent by the TCP stack, as constrained by correct TCP sliding window operation. When the receiver acknowledges the bytes, a `sent` event condition informs the application that it is possible to send more data. Thus, send window-sizing policy is determined entirely by the application. By contrast, conventional OSes buffer send data beyond raw TCP constraints and apply flow control policy inside the kernel.

We built a user-level library, called `libix`, which abstracts away the complexity of our low-level API. It provides a compatible programming model for legacy applications and significantly simplifies the development of new applications. `libix` currently includes a very similar interface to `libevent` and non-blocking POSIX socket operations. It also includes new interfaces for zero-copy read and write operations that are more efficient, at the expense of requiring changes to existing applications.

`libix` automatically coalesces multiple write requests into single `sendv` system calls during each batching round. This improves locality, simplifies error handling, and ensures correct behavior, as it preserves the data stream order even if a transmit fails. Coalescing also facilitates transmit flow control because we can use the transmit vector (the argument to `sendv`) to keep track of outgoing data buffers and, if necessary, reissue writes when the transmit window has more available space, as notified by the `sent` event condition. Our buffer sizing policy is currently very basic; we enforce a maximum pending send byte limit, but we plan to make this more dynamic in the future [39].

Fig. 2.2 illustrates the run-to-completion operation for an elastic thread in the IX dataplane. NIC receive buffers are mapped in the server's main memory and the NIC's receive descriptor ring is filled with a set of buffer descriptors that allow it to transfer incoming packets using DMA. The elastic thread (1) polls the receive descriptor ring and potentially posts fresh buffer descriptors to the NIC for use with future incoming packets. The elastic thread then (2) processes a bounded number of packets through the TCP/IP networking stack, thereby generating event conditions. Next, the thread (3) switches to the user-space application, which consumes all event conditions. Assuming that the incoming packets include remote requests, the application processes these requests and responds with a batch of system calls. Upon return of control from user-space, the thread (4) processes all batched system calls, and in particular the ones that direct outgoing TCP/IP traffic. The thread also (5) runs all kernel timers in order to ensure compliant TCP behavior. Finally (6), it places outgoing Ethernet frames in the NIC's transmit descriptor ring for transmission, and it notifies the NIC to initiate a DMA transfer for these frames by updating the transmit ring's tail register. In a separate pass, it also informs the protocol stack of any buffers that have finished transmitting, based on the transmit ring's head position. The process repeats in a loop until there is no network activity. In this case, the thread enters a quiescent state which involves either hyperthread-friendly polling or optionally entering a power efficient C-state, at the cost of some additional latency.

2.4.4 Multi-core Scalability

The IX dataplane is optimized for multi-core scalability, as elastic threads operate in a synchronization and coherence free manner in the common case. This is a stronger requirement than lock-free synchronization, which requires expensive atomic instructions even when a single thread is the primary consumer of a particular data structure [24]. This is made possible through a set of conscious design and implementation tradeoffs.

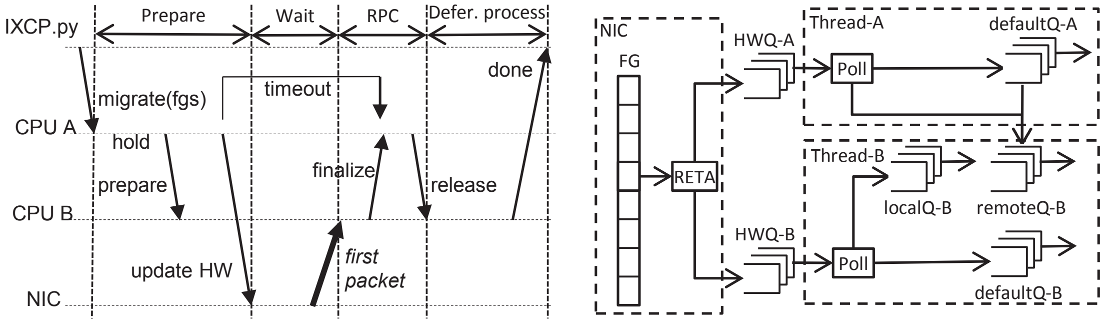
First, system call implementations can only be synchronization-free if the API itself is commutative [22]. The IX API is commutative between elastic threads. Each elastic thread has its own flow identifier namespace, and an elastic thread cannot directly perform operations on flows that it does not own.

Second, the API implementation is carefully optimized. Each elastic thread manages its own memory pools, hardware queues, event condition array, and batched system call array. The implementation of event conditions and batched system calls benefits directly from the explicit, cooperative control transfers between IX and the application. Since there is no concurrent execution by producer and consumer, event conditions and batched system calls are implemented without synchronization primitives based on atomics.

Third, the use of flow-consistent hashing at the NICs ensures that each elastic thread operates on a disjoint subset of TCP flows. Hence, no synchronization or coherence occurs during the processing of incoming requests for a server application. For client applications with outbound connections, we need to ensure that the reply is assigned to the same elastic thread that made the request. Since we cannot reverse the Toeplitz hash used by RSS [90], we simply probe the ephemeral port range to find a port number that would lead to the desired behavior. Note that this implies that two elastic threads in a client cannot share a flow to a server.

IX does have a small number of shared structures, including some that require synchronization on updates. For example, the ARP table is shared by all elastic threads and is protected by RCU locks [86]. Hence, the common case reads are coherence-free but the rare updates are not. RCU objects are garbage collected after a quiescent period that spans the time it takes each elastic thread to finish a run to completion cycle.

Finally, the application code may include inter-thread communication and synchronization. While using IX does not eliminate the need to develop scalable application code, it ensures that there are no scaling bottlenecks in the system and protocol processing code.



(a) Thread-centric view

(b) Packet-centric view

Figure 2.3 – Flow-group Migration Algorithm

2.4.5 Flow group migration

When adding or removing a thread, `IXCP` generates a set of migration requests. Each individual request is for a set of flow groups (`fgs`) currently handled by one elastic thread `A` to be handled by elastic thread `B`. To simplify the implementation, the controller serializes the migration requests and the dataplane assumes that at most one such request is in progress at any point in time. Each thread has three queues that can hold incoming network packets and ensure that packets are delivered in order to the network layer.

Fig. 2.3 illustrates the migration steps in a thread-centric view (Fig. 2.3a) and in a packet-centric view (Fig. 2.3b). The controller and the dataplane threads communicate via lock-free structures in shared memory. First, the controller signals `A` to migrate `fgs` to `B`. `A` first marks each flow group of the set `fgs` with a special tag to hold off normal processing on all threads, moves packets which belong to the flow group set `fgs` from `defaultQ-A` to `remoteQ-B` and stops all timers belonging to the flow group set. `A` then reprograms the NIC's RSS Relocation Table for index `fgs`. Packets still received by `A` will be appended to `remoteQ-B`; packets received by `B` will go to `localQ-B`.

Upon reception of the first packet whose flow group belongs to `fgs` by `B`, `B` signals `A` to initiate the final stage of migration. Then, `B` finalizes the migration by re-enabling `fgs`'s timers, removing all migration tags, and pre-pending to its `defaultQ-B` the packets from `remoteQ-B` and the packets from `localQ-B`. Finally, `B` notifies the control plane that the operation is complete. A migration timer ensures completion of the operation when the NIC does not receive further packets.

2.4.6 The IXCP Control Loop

The IXCP daemon largely relies on Linux host and IX dataplane provided mechanisms. It is implemented in ~500 lines of Python. At its core, the controller adjusts processor resources by suspending and resuming IX elastic threads, specifying the mapping between flow groups and threads, and controlling the processor frequency. For server consolidation scenarios, it may additionally control the resources allocated to background tasks.

The control loop implements a user-specified policy which determines the upper bound on the acceptable queuing delay and the sequence of resource allocation adjustments. For this, it relies on a key side effect of IX's use of adaptive batching: unprocessed packets that form the backlog are queued in a central location, namely in step (1) in the pipeline of Fig. 2.2. Packets are then processed in order, in bounded batches to completion through both the networking stack and the application logic. In other words, each IX core operates like a simple FCFS queuing server, onto which classic queuing and control theory principles can be easily applied. In contrast, conventional operating systems distribute buffers throughout the system: in the NIC (because of coalesced interrupts), in the driver before networking processing, and in the socket layer before being consumed by the application. Furthermore, these conventional systems provide no ordering guarantees across flows, which makes it difficult to pinpoint congestion.

To estimate queuing delays, the controller monitors the iteration time τ and the queue depth Q. With B the maximal batch size, the tail latency is $\sim max(delay) = \lceil Q/B \rceil * \tau$. The dataplane computes each instantaneous metric every $10ms$ for the previous $10ms$ interval. As these metrics are subject to jitter, the dataplane computes the exponential weighted moving averages using multiple smoothing factors (α) in parallel. For example, we track the queue depth as $Q(t,\alpha) = \alpha * Q_{now} + (1-\alpha) * Q(t-1,\alpha)$. The control loop executes at a frequency of 10 Hz, which is sufficient to adapt to load changes.

The control loop is responsible to determine *when* to adjust resources, but not the sequence of resource adjustment steps. For example, adding a core, enabling hyperthread or increasing processor frequency can each increase throughput. In principle, the selection of the resource allocation (and deallocation) sequence can be derived from a Pareto analysis among all possible static configuration. For energy proportionality, the optimization metric is the energy consumption; for workload consolidation, it is the throughput of the background job. We show in §2.5.3 how such a methodology can be applied in practice for a given workload and compute platform.

Deciding when to remove resources is trickier than deciding when to add them, as shallow and near-empty queues do not provide reliable metrics. Instead, the control loop measures idle time and relies on the observation that each change in the configuration adds or removes a predictable level of throughput. The control loop makes resource deallocation decisions when idle time exceeds the throughput ratio.

2.4.7 Security Model

The IX API and implementation has a cooperative flow control model between application code and the network-processing stack. Unlike user-level stacks, where the application is trusted for correct networking behavior, the IX protection model makes few assumptions about the application. A malicious or misbehaving application can only hurt itself. It cannot corrupt the networking stack or affect other applications. All application code in IX runs in user-mode, while dataplane code runs in protected ring 0. Applications cannot access dataplane memory, except for read-only message buffers. No sequence of batched system calls or other user-level actions can be used to violate correct adherence to TCP and other network specifications. Furthermore, the dataplane can be used to enforce network security policies, such as firewalling and access control lists. The IX security model is as strong as conventional kernel-based networking stacks, a feature that is missing from all recently proposed user-level stacks.

The IX dataplane and the application collaboratively manage memory. To enable zero-copy operation, a buffer used for an incoming packet is passed read-only to the application, using virtual memory protection. Applications are encouraged (but not required) to limit the time they hold message buffers, both to improve locality and to reduce fragmentation because of the fixed size of message buffers. In the transmit direction, zero-copy operation requires that the application must not modify outgoing data until reception is acknowledged by the peer, but if the application violates this requirement, it will only result in incorrect data payload.

Since elastic threads in IX execute both the network stack and application code, a long running application can block further network processing for a set of flows. This behavior in no way affects other applications or dataplanes. We use a timeout interrupt to detect elastic threads that spend excessive time in user mode (e.g., in excess of 10ms). We mark such applications as non-responsive and notify the control plane.

The current IX prototype does not yet use an IOMMU. As a result, the IX dataplane is trusted code that has access to descriptor rings with host-physical addresses. This limitation does not affect the security model provided to applications.

2.5 Evaluation of the dataplane

We compared IX to a baseline running Linux kernel version 4.8 and to mTCP [57]. Our evaluation uses both networking microbenchmarks and a widely deployed, event-based application. In all cases, we use TCP as the networking protocol.

2.5.1 Experimental Methodology

Our experimental setup consists of a cluster of 24 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 256 GB of DRAM. Each client and server socket has 8 cores and 16 hyperthreads. All machines are configured with Intel x520 10GbE NICs (82599EB chipset). We connect clients

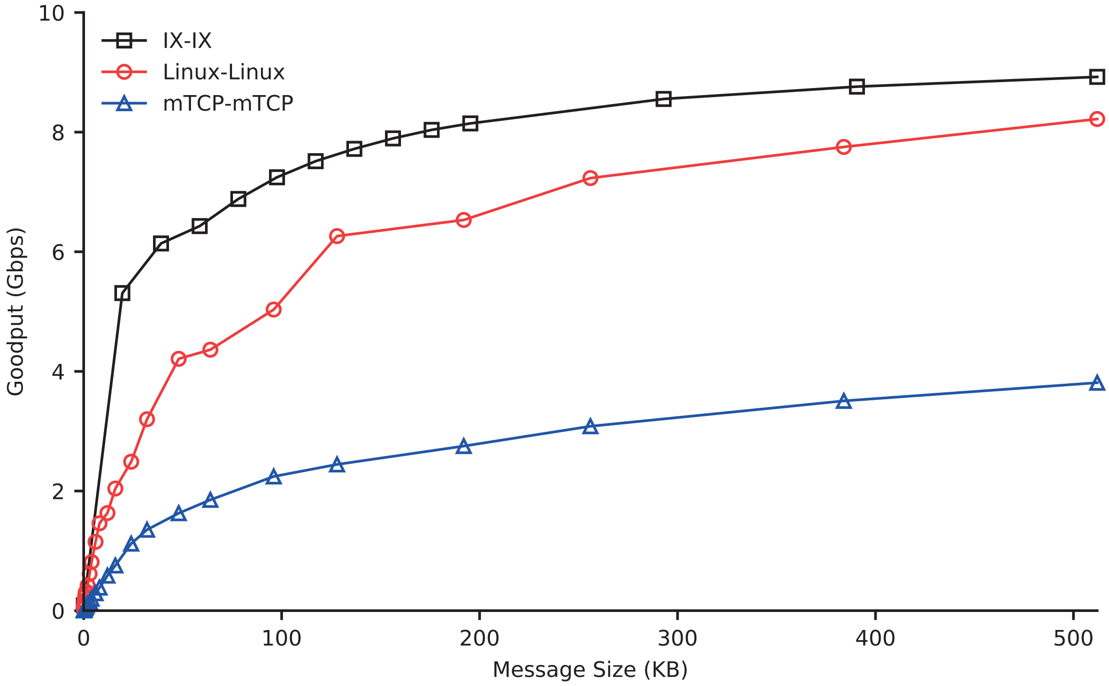


Figure 2.4 – NetPIPE performance for varying message sizes and system software configurations.

to the switch through a single NIC port, while for the server it depends on the experiment. For *10GbE* experiments, we use a single NIC port, and for *4x10GbE* experiments, we use four NIC ports bonded by the switch with a L3+L4 hash.

Our baseline configuration in each machine is an Ubuntu LTS 14.0.4 distribution, updated to the 4.8 Linux kernel, the most recent at time of writing. We enable hyperthreading when it improves performance. Except for §2.5.2, client machines always run Linux. All power management features are disabled for all systems in all experiments. Jumbo frames are never enabled. All Linux workloads are pinned to hardware threads to avoid scheduling jitter, and background tasks are disabled.

The Linux client and server implementations of our benchmarks use the `libevent` framework with the `epoll` system call. We downloaded and installed mTCP from the public-domain release [56], but had to write the benchmarks ourselves using the mTCP API. We run mTCP with the 2.6.36 Linux kernel, as this is the most recent supported kernel version. We report only 10GbE results for mTCP, as it does not support NIC bonding. For IX, we bound the maximum batch size to $B = 64$ packets per iteration, which maximizes throughput on microbenchmarks (see §2.7).

2.5.2 Dataplane performance

Latency and Single-flow Bandwidth

We first evaluated the latency of IX using NetPIPE, a popular ping-pong benchmark, using our 10GbE setup. NetPIPE simply exchanges a fixed-size message between two servers and helps calibrate the latency and bandwidth of a single flow [123]. In all cases, we run the same system on both ends (Linux, mTCP, or IX).

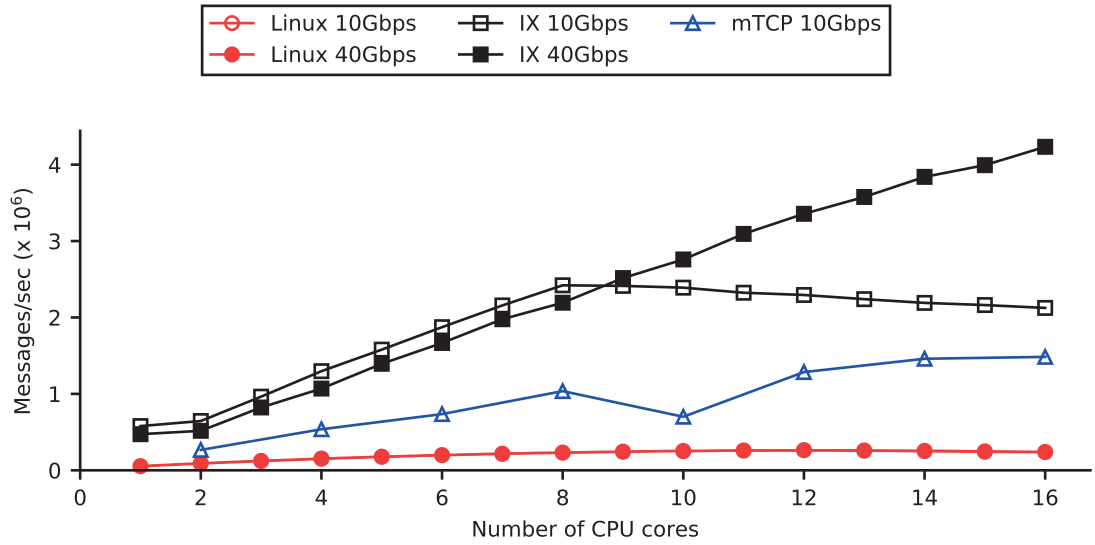
Fig. 2.4 shows the goodput achieved for different message sizes. Two IX servers have a one-way latency of 5.8µs for 64B messages and achieve goodput of 5 Gbps, half of the maximum, with messages as small as 20000 bytes. In contrast, two Linux servers have a one-way latency of 15.5µs and require 96KB messages to achieve 5 Gbps. The differences in system architecture explain the disparity: IX has a dataplane model that polls queues and processes packets to completion whereas Linux has an interrupt model, which wakes up the blocked process. mTCP uses aggressive batching to offset the cost of context switching [57], which comes at the expense of higher latency than both IX and Linux in this particular test.

Throughput and Scalability

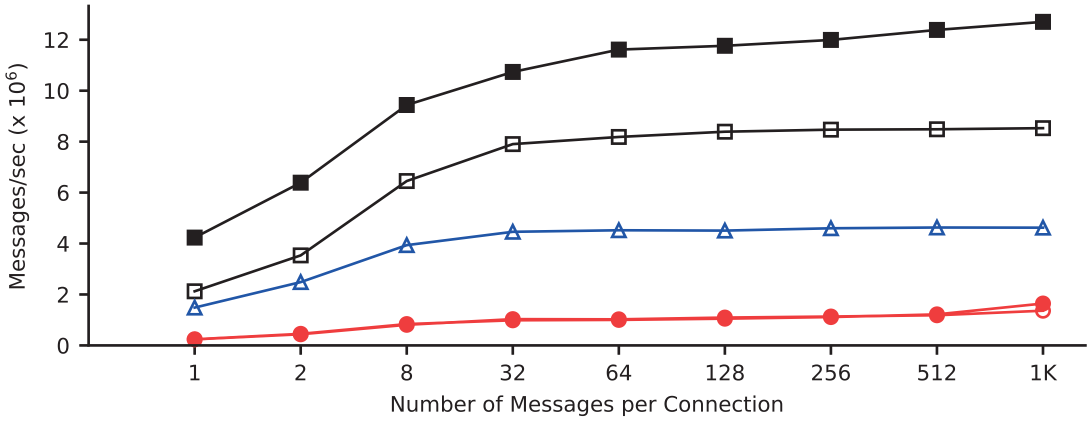
We evaluate IX's throughput and multi-core scalability with the same benchmark used to evaluate MegaPipe [45] and mTCP [57]. 18 clients connect to a single server listening on a single port, send a remote request of size s bytes, and wait for an echo of a message of the same size. Similar to the NetPIPE benchmark, while receiving the message, the server holds off its echo response until the message has been entirely received. Each client performs this synchronous remote procedure call n times before closing the connection. As in [57], clients close the connection using a reset (TCP RST) to avoid exhausting ephemeral ports.

Fig. 2.5 shows the message rate or goodput for both the 10GbE and the 40GbE configurations as we vary the number of cores used, the number of round-trip messages per connection, and the message size respectively. For the 10GbE configuration, the results for Linux and mTCP are consistent with those published in the mTCP paper [57]. For all three tests (core scaling, message count scaling, message size scaling), IX scales more aggressively than mTCP and Linux. Fig. 2.5a shows that IX needs only 4 cores to saturate the 10GbE link whereas mTCP requires all 8 cores. On Fig. 2.5b for 1024 round-trips per connection, IX delivers 8.5 million messages per second, which is 1.8× the throughput of mTCP and of and 6.3× that of Linux. With this packet rate, IX achieves line rate and is limited only by 10GbE bandwidth.

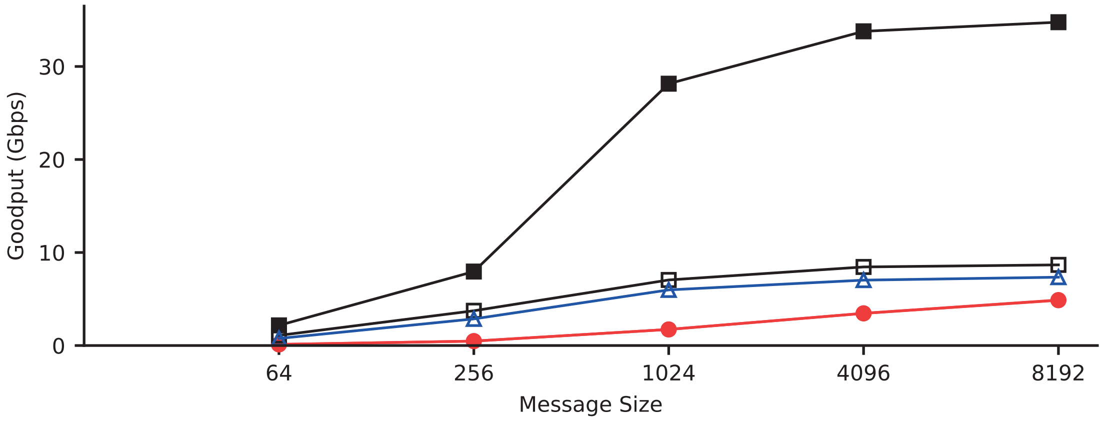
Fig. 2.5 also shows that IX scales well beyond 10GbE to a 4x10GbE configuration. Fig. 2.5a shows that IX linearly scales to deliver 4.2 million TCP connections per second on 4x10GbE. Fig. 2.5b shows a speedup of 2.0× with $n = 1$ and of 1.5× with $n = 1024$ over 10GbE IX. Finally, Fig. 2.5c shows IX can deliver 8KB messages with a goodput of 34.8 Gbps, for a wire throughput of 38.3 Gbps, out of a possible 39.7 Gbps. Overall, IX makes it practical to scale protected TCP/IP processing beyond 10GbE, even with a single socket multi-core server.



(a) Multi-core scalability (n=1, s=64B)

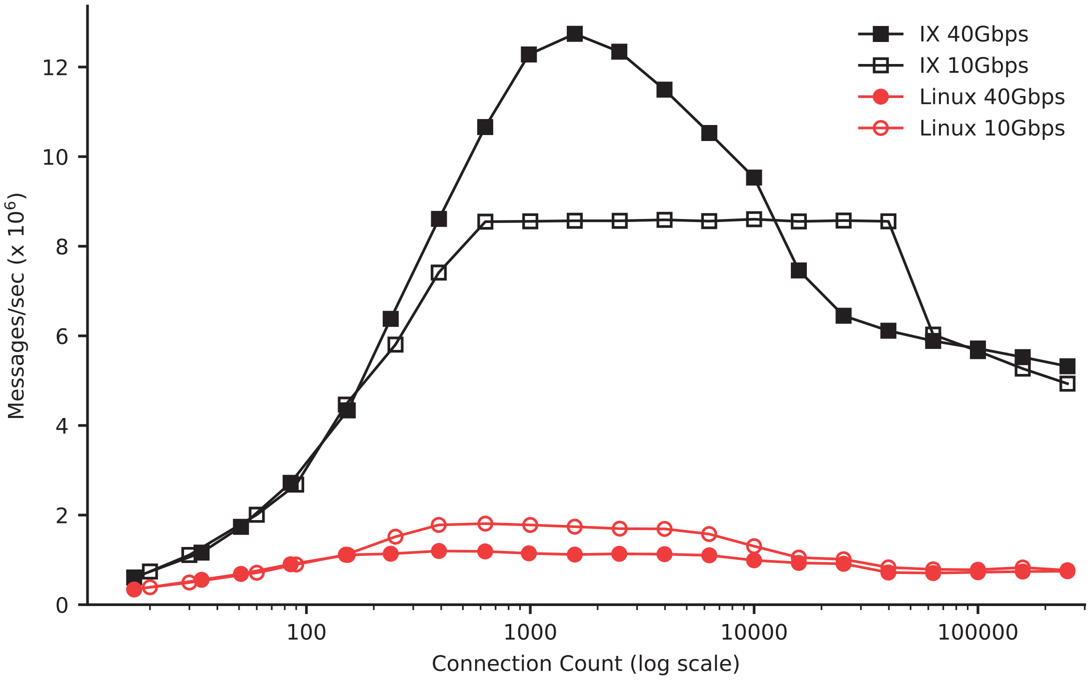


(b) n round-trips per connection. (s=64B)

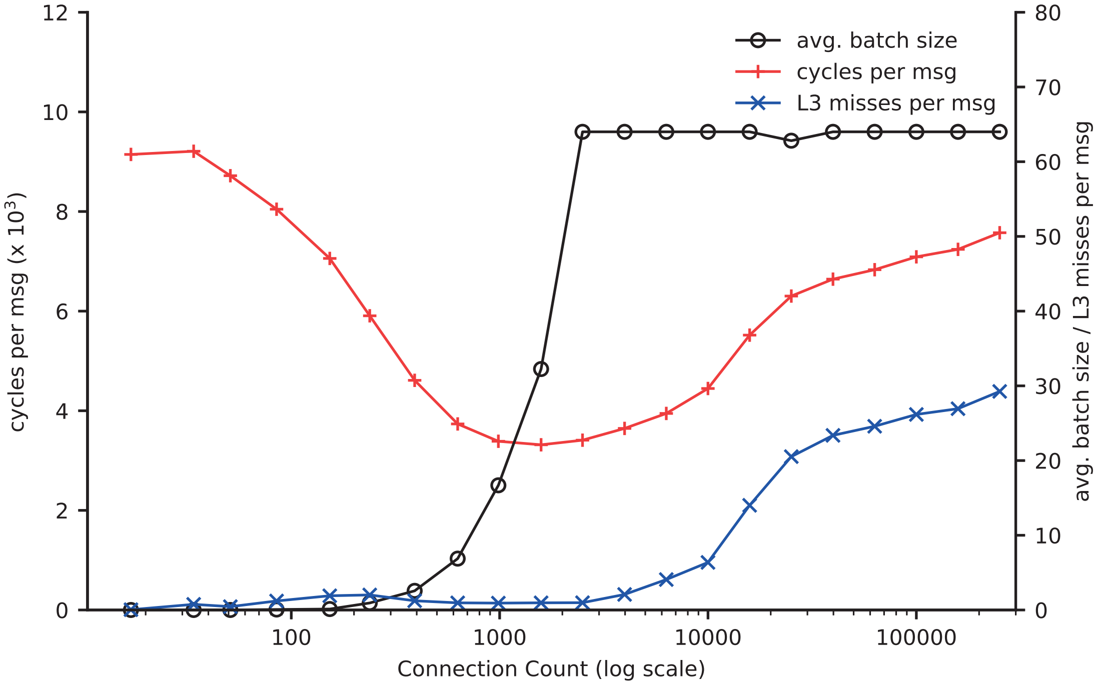


(c) Different message sizes s (n=1)

Figure 2.5 – Multi-core scalability and high connection churn for 10GbE and 4x10GbE setups. In (a), half steps indicate hyperthreads.



(a) Throughput for 10GbE and 4x10GbE configurations.



(b) Hardware metrics for IX on the 4x10GbE configuration.

Figure 2.6 – Connection scalability of IX.

Connection Scalability

We also evaluate IX's scalability when handling a large number of concurrent connections on the 4x10GbE setup. 18 client machines runs n threads, with each thread repeatedly performing a 64B remote procedure call to the server with a variable number of active connections. We experimentally set $n = 24$ to maximize throughput. We report the maximal throughput in messages per second for a range of total established connections.

Fig. 2.6 shows up to 250,000 connections, which is the upper bound we can reach with the available client machines. As expected, Fig. 2.6a shows that throughput increases with the degree of connection concurrency, but then decreases for very large connections counts due to the increasingly high cost of multiplexing among open connections. At the peak, IX performs 11× better than Linux, consistent with the results from Fig. 2.5b. With 250,000 connections and 4x10GbE, IX is able to deliver 42% of its own peak throughput.

Fig. 2.6b shows that the drop in throughput is not due to an increase in the instruction count, but instead can be attributed to the performance of the memory subsystem. Intel's Data Direct I/O technology, an evolution of DCA [51], eliminates nearly all cache misses associated with DMA transfers when given enough time between polling intervals, resulting in as little as 2.0 L3 cache misses per message for up to 2,500 concurrent connections, a scale where all of IX's data structures fit easily in the L3 cache. In contrast, the workload averages 29 L3 cache misses per message when handling 250,000 concurrent connections. At high connection counts, the working set of this workload is dominated by the TCP connection state and does not fit into the processor's L3 cache. Nevertheless, we believe that further optimizations in the size and access pattern of lwIP's TCP/IP protocol control block structures can substantially reduce this handicap.

Fig. 2.6b additionally gives insights about the positive impact of the adaptive batching. As the load increases, the average batch size increases from 0 to the maximum configured value, which is 64 in our benchmark setup. At the same time, the average number of cycles per message decreases from 9,000 to less than 4,000, before it starts increasing again due to the negative impact of L3 cache misses.

`Memcached` Performance

Finally, we evaluated the performance benefits of IX with `memcached`, a widely deployed, in-memory, key-value store built on top of the `libevent` framework [89]. It is frequently used as a high-throughput, low-latency caching tier in front of persistent database servers. `memcached` is a network-bound application, with threads spending over 80% of execution time in kernel mode for network processing [71]. It is a difficult application to scale because the common deployments involve high connection counts for `memcached` servers and small-sized requests and replies [3, 98]. Furthermore, `memcached` has well-known scalability limitations [77]. To alleviate some of the limitations, we configure `memcached` with a larger hash table size (`-o`

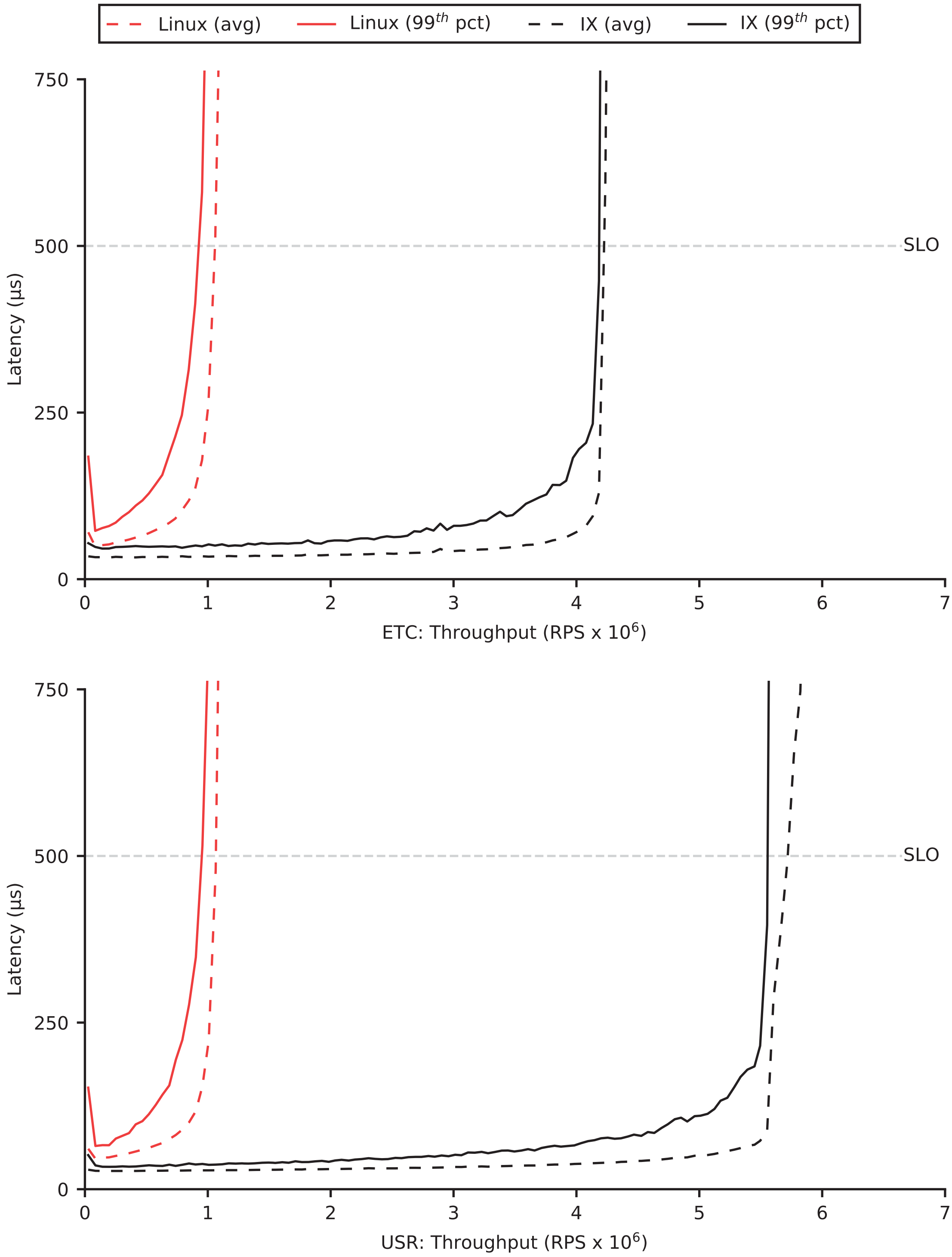


Figure 2.7 – Average and 99^{th} percentile latency as a function of throughput for the ETC and USR `memcached` workloads.

`hashpower=20`) and use a random replacement policy instead of the built-in LRU, which requires a global lock. We configure `memcached` similarly for Linux and IX.

We use the `mutilate` load-generator to place a selected load on the server in terms of requests per second (RPS) and measure response latency [94]. `mutilate` coordinates a large number of client threads across multiple machines to generate the desired RPS load, while a separate unloaded client measures latency by issuing one request at a time across 32 open connections, to eliminate statistical errors due to slight potential imbalances across network card queues and respective CPU cores handling those queues. We configure `mutilate` to generate load representative of two workloads from Facebook [3]: the ETC workload that represents that highest capacity deployment in Facebook, has 20B–70B keys, 1B–1KB values, and 75% GET requests; and the USR workload that represents deployment with most GET requests in Facebook, has short keys (<20B), 2B values, and 99% GET requests. In USR, almost all traffic involves minimum-sized TCP packets. Each request is issued separately (no `multiget` operations). However, clients are permitted to pipeline up to four requests per connection if needed to keep up with their target request rate. We use 11 client machines to generate load for a total of 2,752 connections to the `memcached` server.

To provide insights into the full range of system behaviors, we report average and 99^{th} percentile latency as a function of the achieved throughput. The 99^{th} percentile latency captures tail latency issues and is the most relevant metric for datacenter applications [25]. Most commercial `memcached` deployments provision each server so that the 99^{th} percentile latency does not exceed 200µs to 500µs.

We carefully tune the Linux baseline setup according to the guidelines in [71]: we pin `memcached` threads, configure interrupt-distribution based on thread-affinity, and tune interrupt moderation thresholds. Additionally, we increase the socket accept queue size and disable SYN cookies via `sysctl` and via the respective `memcached` command line argument to accommodate for the large connection accept rate at the beginning of the benchmark. Finally, to resolve observed unexpected 99^{th} pct. latency spikes when running `memcached` under Linux, we disable transparent huge pages via `sysfs`, instruct `memcached` to use the `mlockall` system call and utilize `numactl` to pin memory pages on the desired NUMA node of our server. We believe that our baseline Linux numbers are as tuned as possible for this hardware using the open-source version of `memcached-1.4.18`. We report the results for the server configuration that provides the best performance: 8 cores with hyperthreading enabled.

Porting `memcached` to IX primarily consisted of adapting it to use our event library. In most cases, the port was straightforward, replacing Linux and `libevent` function calls with their equivalent versions in our API. We did yet not attempt to tune the internal scalability of `memcached` [37] or to support zero-copy I/O operations.

Fig. 2.7a and Fig. 2.7b show the throughput-latency curves for the two `memcached` workloads for Linux and IX, while Table 2.3 reports the unloaded, round-trip latencies and maximum request rate that meets a service-level agreement, both measured at the 99^{th} percentile. IX

Configuration	Minimum latency @99^{th} pct	RPS for SLO: < 500µs @99^{th} pct
ETC-Linux	72µs	897K
ETC-IX	46µs	4183K
USR-Linux	65µs	903K
USR-IX	34µs	5552K

Table 2.3 – Unloaded latency and maximum RPS for a given service-level agreement for the `memcached` workloads ETC and USR.

cuts the unloaded latency of both workloads in half. Note that we use Linux clients for these experiments; running IX on clients should further reduce latency.

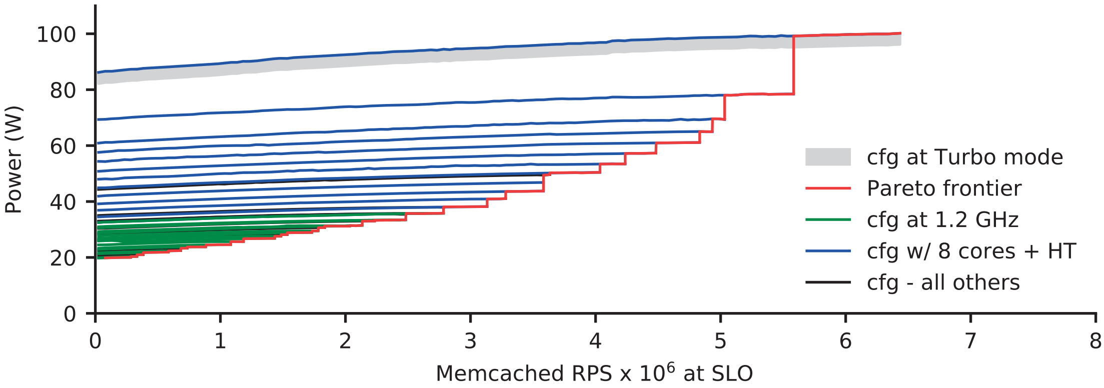
At high request rates, the distribution of CPU time shifts from being ∼ 80% in the Linux kernel to 60% in the IX dataplane kernel. This allows IX to increase throughput by 4.7× and 6.1× for ETC and USR respectively at a 500µs tail latency SLO.

2.5.3 Pareto-Optimal Static Configurations

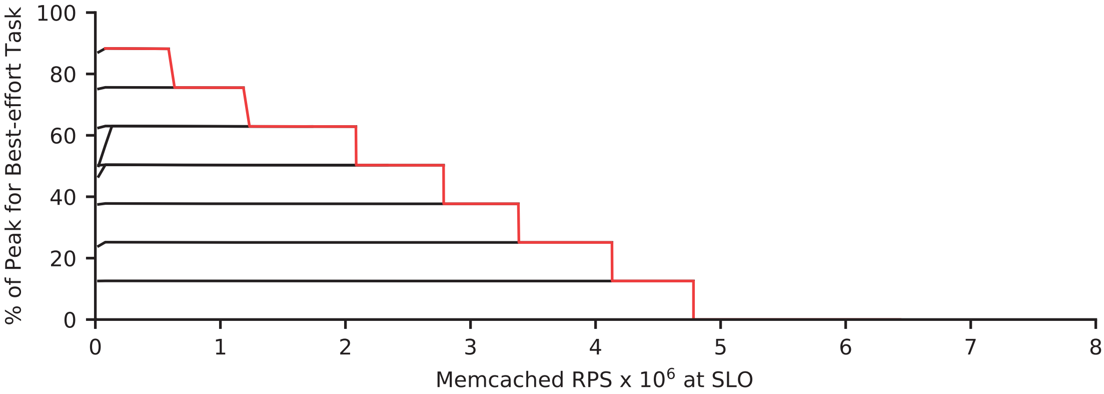
Static resource configurations allow for controlled experiments to quantify the tradeoff between an application's performance and the resources consumed. Our approach limits bias by considering many possible static configurations in the three-dimensional space of core, hyperthread, and frequency. For each static configuration, we characterize the maximum load that meets the SLO ($\leq 500\mu s$ @ 99^{th} percentile); we then measure the energy draw and throughput of the background job for all load levels up to the maximum load supported. From this large data set, we derive the set of meaningful static configurations and build the Pareto efficiency frontier. The frontier specifies, for any possible load level, the optimal static configuration and the resulting minimal energy draw or maximum background throughput, depending on the scenario.

Fig. 2.8 presents the frontier for the `memcached` USR workload for two different policies: energy proportionality, which aims to minimize the amount of energy consumed while maintaining SLO and workload consolidation, which aims to maximize the throughput of some background process while also maintaining the SLO of the latency-sensitive application.

The graphs each plot the objective—which is either to minimize energy or maximize background throughput—as a function of the foreground throughput, provided that the SLO is met. Except for the red lines, each line corresponds to a distinct static configuration of the system: the green curves correspond to configuration at the minimal clock rate of 1.2 Ghz; the blue curves use all available cores and hyperthreads; other configurations are in black. In Turbo Boost mode, the energy drawn is reported as a band since it depends on operating temperature.



(a) Energy Proportionality



(b) Server Consolidation (IX)

Figure 2.8 – Pareto efficiency for energy proportionality and workload consolidation for IX. The Pareto efficiency is in red while the various static configurations are color-coded according to their distinctive characteristics.

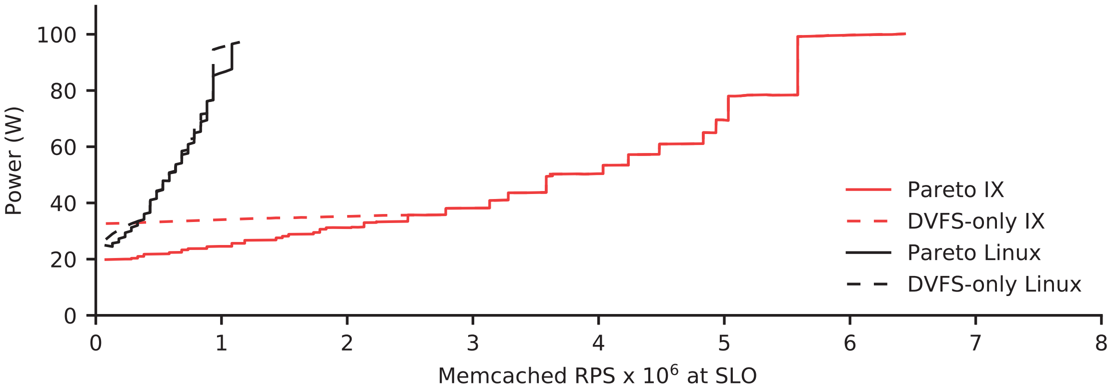


Figure 2.9 – Energy-proportionality comparison between the Pareto-optimal frontier considering only DVFS adjustments, and the full Pareto frontier considering core allocation, hyperthread allocations, and frequency.

Finally, the red line is the Pareto frontier, which corresponds, for any load level, to the optimal result using any of the static configurations available. Each graph only shows the static configurations that participate in the frontier.

A note on Turbo Boost: For any given throughput level, we observe that the reported power utilization is stable for all CPU frequencies *except* for Turbo Boost. When running in Turbo Boost, the temperature of the CPU gradually rises over a few minutes from 58°C to 78°C, and with it the dissipated energy rises by 4 W for the same level of performance. The experiments in §2.5.3 run for a long time in Turbo Boost mode with a hot processor; we therefore report those results as an energy band of 4 W.

Energy proportionality We evaluate 224 distinct combinations: from one to eight cores, using consistently either one or two threads per core, for 14 different DVFS levels from 1.2 Ghz to 2.4 Ghz as well as Turbo Boost. Fig. 2.8a shows the 45 static configurations (out of 224) that build the Pareto frontier for energy proportionality. The figures confirm the intuition that: (i) various static configurations have very different dynamic ranges, beyond which they are no longer able to meet the SLO; (ii) each static configuration draws substantially different levels of energy for the same amount of work; (iii) at the low-end of the curve, many distinct configurations operate at the minimal frequency of 1.2 Ghz, obviously with a different number of cores and threads, and contribute to the frontier; these are shown in green in the Figure; (iv) at the high-end of the range, many configurations operate with the maximum of 8 cores, with different frequencies including Turbo Boost.

Consolidation The methodology here is a little different. We first characterize the background job, and observe that it delivers energy-proportional throughput up to 2.4 Ghz, but that Turbo Boost came at an energy/throughput premium. Consequently, we restrict the Pareto configuration space at 2.4 Ghz; the objective function is the throughput of the background job, expressed as a fraction of the throughput of that same job without any foreground application. Background jobs run on all cores that are not used by the foreground application. Fig. 2.8b shows the background throughput, expressed as a fraction of the standalone throughput, as a function of the foreground throughput, provided that the foreground application meets the SLO: as the foreground application requires additional cores to meet the SLO, the background throughput decreases proportionally.

DVFS-only alternative Fig. 2.9 further analyzes the data and compares the Pareto frontiers of Linux 4.8 and IX for the energy-proportional scenario with an alternate frontier that only considers changes in DVFS frequency. We observe that the impact of DVFS-only controls differs noticeably between Linux and IX: with Linux, the DVFS-only alternate frontier is very close to the Pareto frontier, meaning that a DVFS-only approach such as Pegasus [79] or Adrenaline [50] would be adequate. This is due to Linux's idling behavior, which saves resources. In the case of IX however—and likely for any polling-based dataplane—a DVFS-only scheduler would provide worse energy proportionality at low-moderate loads than a corresponding Linux-based solution. As many datacenter servers operate in the 10%-30%

range [6], we conclude that a dynamic resource allocation scheme involving both DVFS and core allocation is necessary for dataplane architectures.

2.6 Evaluation

We use the results from §2.5.3 to derive a resource configuration policy framework, whose purpose is to determine the sequence of configurations to be applied, as a function of the load on the foreground application, to both the foreground (latency-sensitive) and background (batch) applications. Specifically, given an ever-increasing (or -decreasing) load on the foreground applications, the goal is to determine the sequence of resource configurations minimizing energy consumption or maximizing background throughput, respectively.

We observe that (i) the latency-sensitive application (`memcached`) can scale nearly linearly, up to the 8 cores of the processor; (ii) it benefits from running a second thread on each core, with a consistent speedup of 1.3×; (iii) it is most energy-efficient to first utilize the various cores, and only then to enable the second hyperthread on each core, rather than the other way around; and (iv) it is least energy-efficient to increase the frequency.

We observe that the background application (i) also scales linearly; but (ii) does not benefit from the 2nd hyperthread; (iii) is nearly energy-proportional across the frequency spectrum, with the exception of Turbo Boost. From a total cost of ownership perspective, the most efficient operating point for the workload consolidation of the background task is therefore to run the system at the processor's nominal 2.4 Ghz frequency whenever possible.

We combine these observations with the data from the Pareto analysis and derive the following policies:

Energy Proportional Policy As a base state, run with only one core and hyperthread with the socket set at the minimal clock rate (1.2Ghz). To add resources, first enable additional cores, then enable hyperthreads on all cores (as a single step), and only after that gradually increase the clock rate until reaching the nominal rate (2.4Ghz); finally enable Turbo Boost. To remove resources, do the opposite. This policy leads to a sequence of 22 different configurations.

Workload Consolidation Policy As a base state, run the background jobs on all available cores with the processor at the nominal clock rate. To add resources to the foreground application, first shift cores from the background thread to the foreground application one at a time. This is done by first suspending the background threads; use both hyperthreads of the newly freed core for the foreground application. Next, stop the background job entirely and allocate all cores to the foreground applications. As a final step, enable Turbo Boost. This policy leads to a sequence of 9 different configurations.

These policies closely track the corresponding Pareto frontier. For energy proportionality, (i) the 45 different static configurations of the frontier are a superset of the configurations

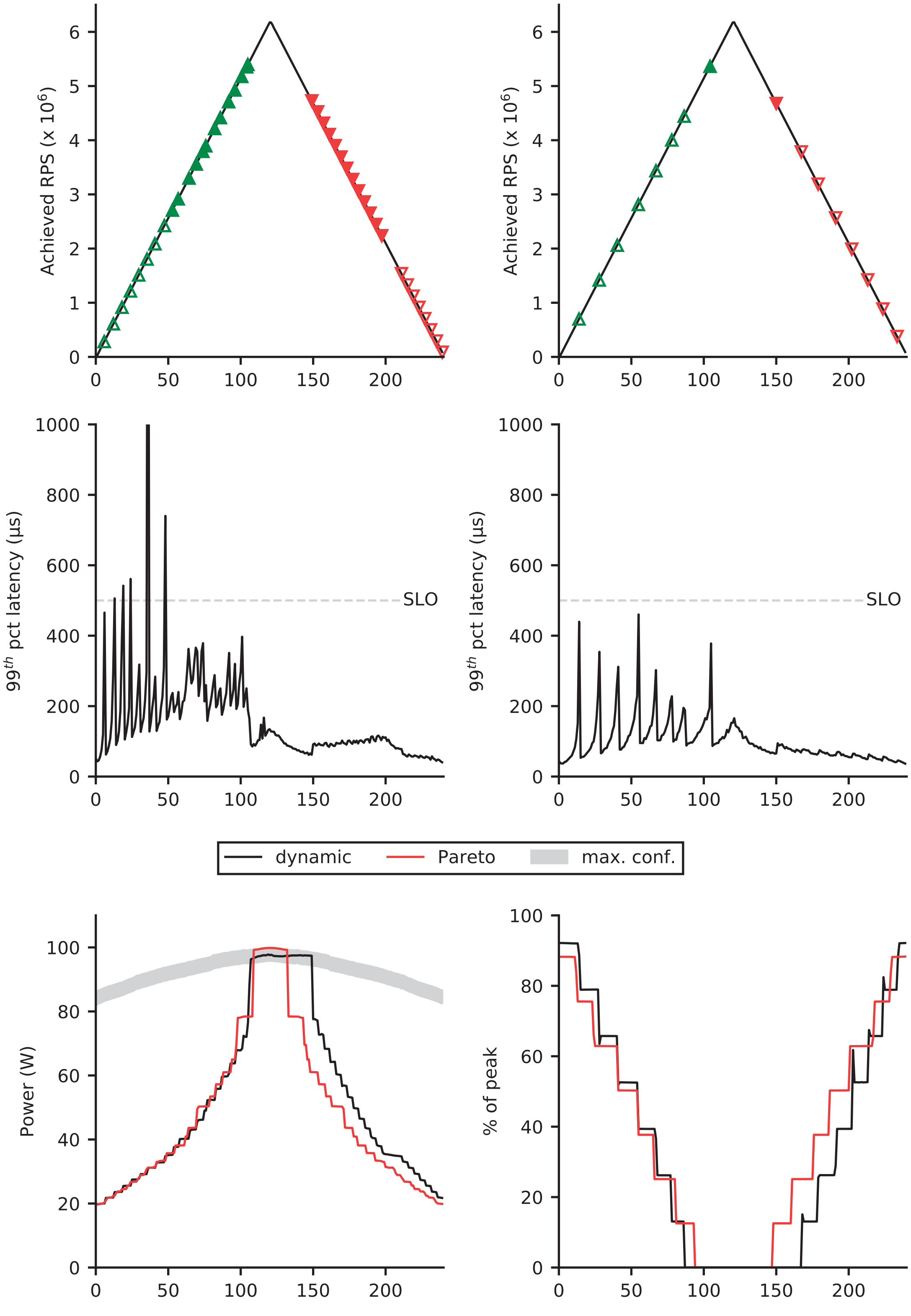


Figure 2.10 – Energy proportionality (left) and workload consolidation (right) for the *slope* pattern

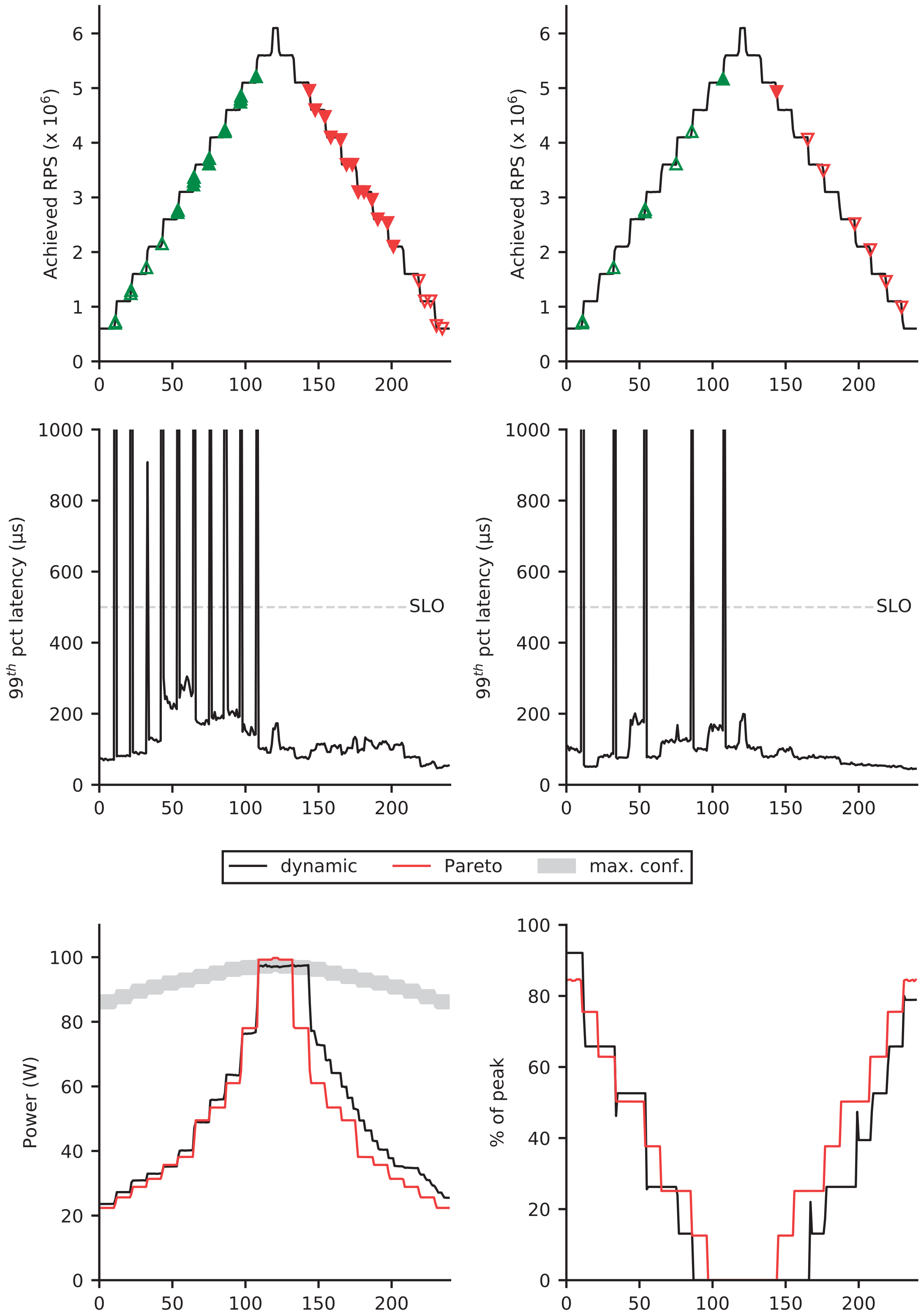


Figure 2.11 – Energy proportionality (left) and workload consolidation (right) for the *step* pattern

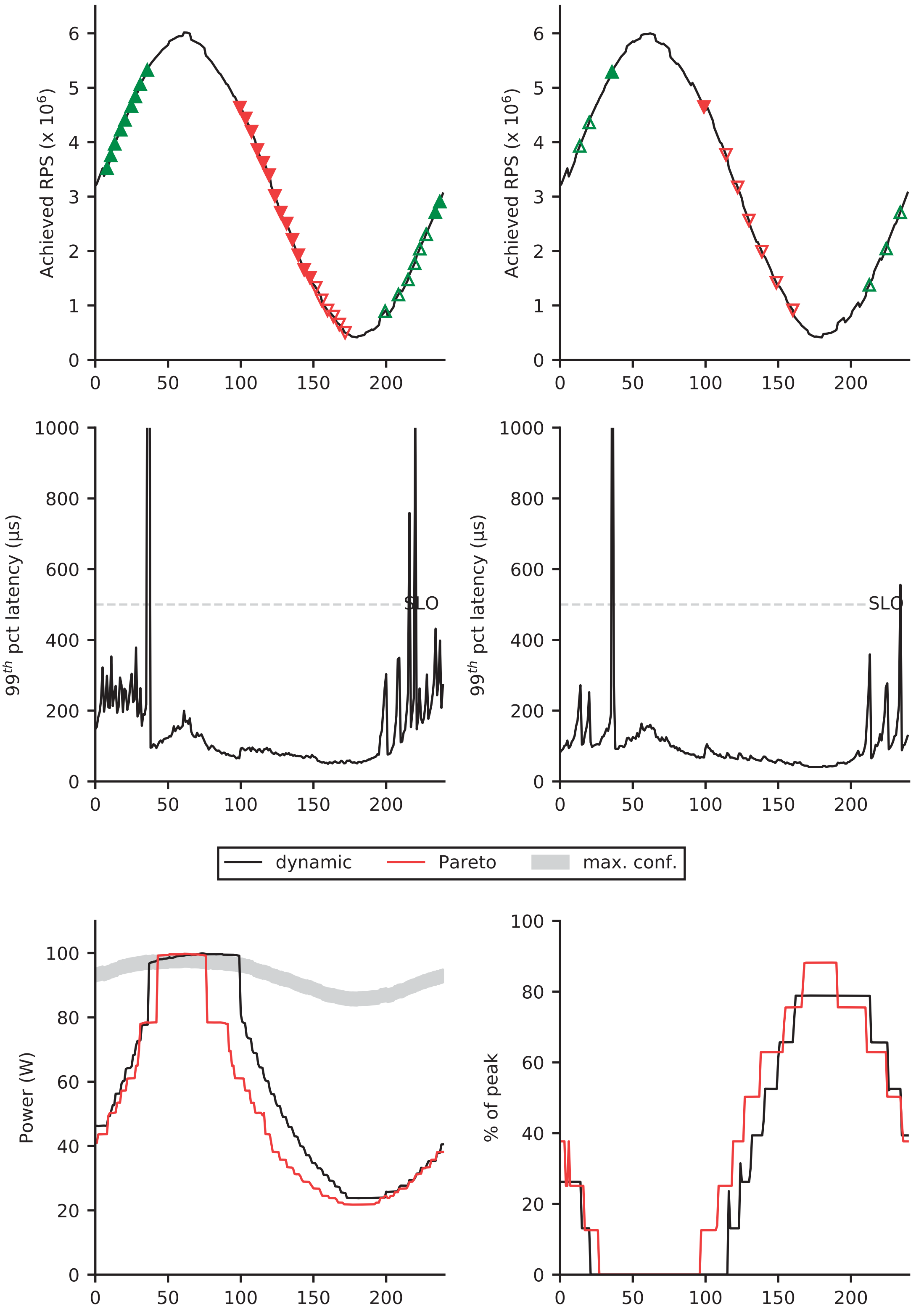


Figure 2.12 – Energy proportionality (left) and workload consolidation (right) for the *sin+noise* pattern

	Smooth	Step	Sine+noise
Energy Proportionality (W)			
Max. power	93	93	95
Measured	46 (-51%)	47 (-50%)	55 (-42%)
Pareto bound	42 (-54%)	43 (-54%)	49 (-49%)
Server consolidation opportunity (% of peak)			
Pareto bound	48%	47%	38%
Measured	44%	41%	31%

Table 2.4 – Energy Proportionality and Consolidation gains.

enabled by the policy, and (ii) the difference in overall impact in terms of energy spent is marginal. For consolidation, Pareto and policy nearly identically overlap.

We use three synthetic, time-accelerated load patterns to evaluate the effectiveness of the control loop under stressful conditions. All three vary between nearly idle and maximum throughput within a four minute period: the *slope* pattern gradually raises the target load from 0 and 6.2M RPS and then reduces its load; the *step* pattern increases load by 500 KRPS every 10 seconds; and finally the *sine+noise* pattern is a basic sinusoidal pattern modified by randomly adding sharp noise that is uniformly distributed over [-250,+250] KRPS and re-computed every 5 seconds. The slope pattern provides a baseline to study smooth changes, the step pattern models abrupt and massive changes, while the sine+noise pattern is representative of daily web patterns [132].

Fig. 2.10, Fig. 2.11 and Fig. 2.12, show the results of these three dynamic load patterns for the energy proportionality and workload consolidation scenarios. In each case, the top figure measures the observed throughput. They are annotated with the control loop events that add resources (green) or remove them (red). Empty triangles correspond to core allocations and full triangles to DVFS changes. The middle figure evaluates the soundness of the algorithm and reports the 99^{th} percentile latency, as observed by a client machine and reported every second. Finally, the bottom figures compare the overall efficiency of our solution based on dynamic resource controls with (i) the maximal static configuration, using all cores and Turbo Boost, and (ii) the ideal, synthetic efficiency computed using the Pareto frontier of Fig. 2.8.

Energy Proportionality

The left column of Figs. 2.10–2.11–2.12 shows the dynamic behavior for the energy proportionality scenario. The top-left graph shows that the workload tracks the desired throughput of the pattern and exercises the entire sequence of configurations, gradually adding cores, enabling hyperthreading, increasing the frequency and finally enabling Turbo Boost, before doing it in reverse. The step pattern of Fig 2.11 is particularly challenging, as the instant change in load level requires multiple, back-to-back, configurations changes. With a few exceptions, the middle-left graph shows that the latencies remain well below the 500µs SLO. We further discuss the violations below. For these three figures, the bottom-left graph compares the power dissipated by the workload with the corresponding power levels as determined by the Pareto frontier (lower bound) or the maximum static configuration (upper bound). This graph measures the effectiveness of the control loop to maximize energy proportionality. We observe that the dynamic (actually measured) power curve tracks the Pareto (synthetic) curve well, which defines a bound on energy savings. When the dynamic resource controls enter Turbo Boost mode, the measured power in all three cases starts at the lower end of the 4 W range and then gradually rises, as expected. Table 2.4 shows that the three patterns have Pareto savings bounds of 49%, 54% and 54%. IX's dynamic resource controls results in energy savings of 42%, 50% and 51%, which is 87%, 93% and 94% of the theoretical bound.

Consolidation

The right column of Figs. 2.10–2.11–2.12 shows the dynamic behavior for the workload consolidation scenario. Here also, the top-right graphs show that the throughput tracks well the desired load. Recall that the consolidation policy always operates at the processor's nominal rate (or Turbo), which limits the number of configuration changes. The middle-right graph similarly confirms that the system meets the SLO, with few exceptions. The bottom-right graphs plot the throughput of the background batch application, expressed as a percentage of its throughput on a dedicated processor at 2.4 Ghz. We compare it only to the Pareto optimal upper bound as a maximum configuration would monopolize cores and deliver zero background throughput. Table 2.4 shows that, for these three patterns, our consolidation policy delivers 31%–44% of the standalone throughput of the background job, which corresponds to 81%–92% of the Pareto bound.

SLO violations A careful study of the SLO violations of the 6 runs shows that they fall into two categories. First, there are 16 violations caused by delays in packet processing due to flow group migrations resulting from the addition of a core. Second, there are 9 violations caused by abrupt increase of throughput, mostly in the step pattern, which occur before any flow migrations. The control plane then reacts quickly (in ~100 ms) and accommodates to the new throughput by adjusting resources. To further confirm the abrupt nature of throughput increase specific to the step pattern, we note that the system performed up three consecutive increases in resources in order to resolve a single violation. 23 of the 25 total violations last a single second, with the remaining two violations lasting two seconds. We believe that the compliance with the SLO achieved by our system is more than adequate for any practical production deployment.

Flow group migration analysis Table 2.5 measures the latency of the 550 flow group migrations that occur during the 6 benchmarks, as described in §2.4.5. It also reports the total number of packets whose processing is deferred during the migration (rather than dropped or reordered). We first observe that migrations present distinct behaviors when scaling up and when scaling down the number of cores. The difference can be intuitively explained since the migrations during the scale up are performed in a heavily loaded system, while the system during the scale down is partially idle. In absolute terms, migrations that occur when adding a

		avg	95th pct.	max.	stddev
add core	prepare (µs)	98	249	3293	341
	wait (µs)	187	747	1120	237
	rpc (µs)	164	417	918	143
	deferred (µs)	134	433	2676	304
	total (µs)	585	1684	6126	697
	# packets	87	510	2711	263
remove core	prepare (µs)	22	48	287	24
	wait (µs)	36	105	309	39
	rpc (µs)	12	26	40	8
	deferred (µs)	16	35	78	11
	total (µs)	89	162	335	44
	# packets	3	8	13	2

Table 2.5 – Breakdown of flow group migration measured during the six benchmarks.

core take 585 on average and less than 1.5 ms 95% of the time. The outliers can be explained by rare occurrences of longer preparation times or when processing up to 2711 deferred packets.

2.7 Discussion

What makes IX fast The results in §2.5 show that a networking stack can be implemented in a protected OS kernel and still deliver wire-rate performance for most benchmarks. The tight coupling of the dataplane architecture, using only a minimal amount of batching to amortize transition costs, causes application logic to be scheduled at the right time, which is essential for latency-sensitive workloads. Therefore, the benefits of IX go beyond just minimizing kernel overheads. The lack of intermediate buffers allows for efficient, application-specific implementations of I/O abstractions such the `libix` event library. The zero-copy approach helps even when the user-level libraries add a level of copying, as it is the case for the `libevent` compatible interfaces in `libix`. The extra copy occurs much closer to the actual use, thereby increasing cache locality. Finally, we carefully tuned IX for multi-core scalability, eliminating constructs that introduce synchronization or coherence traffic.

The IX dataplane optimizations — run to completion, adaptive batching, and a zero-copy API — can also be implemented in a user-level networking stack in order to get similar benefits in terms of throughput and latency. While a user-level implementation would eliminate protection domain crossings, it would not lead to significant performance improvements over IX. Protection domain crossings inside VMX non-root mode add only a small amount of extra overhead, on the order of a single L3 cache miss [11]. Moreover, these overheads are quickly amortized at higher packet rates.

Subtleties of adaptive batching Batching is commonly understood to trade off higher latency at low loads for better throughput at high loads. IX uses adaptive, bounded batching to actually improve on both metrics. Fig. 2.13 compares the latency vs. throughput on the USR

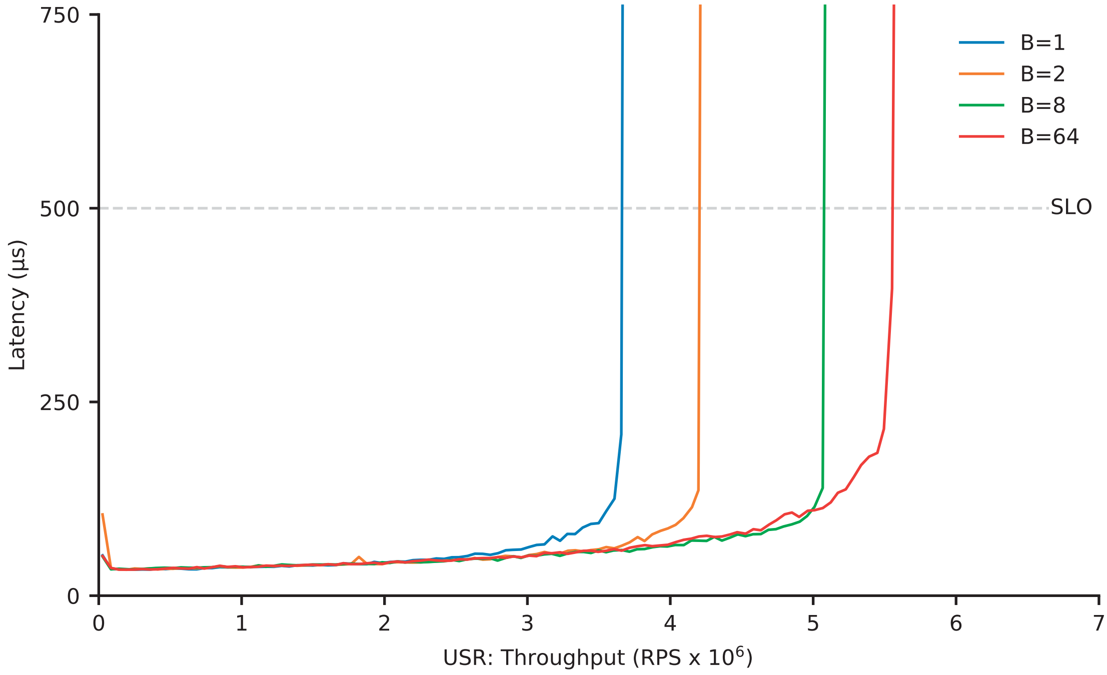


Figure 2.13 – 99^{th} percentile latency as a function of throughput for USR workload from Fig. 2.7b, for different values of the batch bound B.

`memcached` workload of Fig. 2.7b for different upper bounds B to the batch size. At low load, B does not impact tail latency, as adaptive batching does not delay processing of pending packets. At higher load, larger values of B improve throughput, by 29% between $B = 1$ to $B = 16$. For this workload, $B \geq 16$ maximizes throughput.

While tuning IX performance, we ran into an unexpected hardware limitation that was triggered at high packet rates with small average batch sizes (i.e. before the dataplane was saturated): the high rate of PCIe writes required to post fresh descriptors at every iteration led to performance degradation as we scaled the number of cores. To avoid this bottleneck, we simply coalesced PCIe writes on the receive path so that we replenished at least 32 descriptor entries at a time. Luckily, we did not have to coalesce PCIe writes on the transmit path, as that would have impacted latency.

Using Pareto as a guide Even though the Pareto results are not used by the dynamic resource controller, the Pareto frontier proved to be a valuable guide, first to motivate and quantify the problem, then to derive the configuration policy sequence, and finally to evaluate the effectiveness of the dynamic resource control by setting an upper bound on the gains resulting from dynamic resource allocation. Many factors such as software scalability, hardware resource contention, network and disk I/O bottlenecks, will influence the Pareto frontier of any given application, and therefore the derived control loop policies. Without violating the SLO, the methodology explicitly trades off worst average and tail latency for better overall efficiency. More complex SLOs, taking into account multiple aspects of latency distribution, would define a different Pareto frontier, and likely require adjustments to the control loop.

Adaptive, flow-centric scheduling The new flow-group migration algorithm of §2.4.5 leads to a *flow-centric* approach to resource scheduling, where the network stack and application logic always follow the steering decision. POSIX applications can balance flows by migrating file descriptors between threads or processes, but this tends to be inefficient because it is difficult for the kernel to match the flow's destination receive queue to changes in CPU affinity. Flow director can be used by Linux to adjust the affinity of individual network flows as a reactive measure to application-level and kernel thread migration rebalancing, but the limited size of the redirection table prevents this mechanism from scaling to large connection counts. By contrast, our approach allows flows to be migrated in entire groups, improving efficiency, and is compatible with more scalable hardware flow steering mechanisms based on RSS.

Limitations of current prototype The current IX implementation does not yet exploit IOMMUs or VT-d. Instead, it maps descriptor rings directly into IX memory, using the Linux pagemap interface to determine physical addresses. Although this choice puts some level of trust into the IX dataplane, application code remains securely isolated. In the future, we plan on using IOMMU support to further isolate IX dataplanes. We anticipate overhead will be low because of our use of large pages. We also plan to add support for interrupts to the IX dataplanes. The IX execution model assumes some cooperation from application code running in elastic threads. Specifically, applications should handle events in a quick, non-blocking manner; operations with extended execution times are expected to be delegated to background threads rather than execute within the context of elastic threads. The IX dataplane is designed around polling, with the provision that interrupts can be configured as a fallback optimization to refresh receive descriptor rings when they are nearly full and to refill transmit descriptor rings when they are empty (steps (1) and (6) in Fig 2.2). Occasional timer interrupts are also required to ensure full TCP compliance in the event an elastic thread blocks for an extended period.

Hardware trends Our experimental setup using one *Sandy Bridge* processor and the Intel 82599 NIC [53]. Hash filters for flow group steering could benefit from recent trends in NIC hardware. For example, Intel's new XL710 chipset [54], has a 512 entry hash LUT (as well as independent 64 entry LUTs for each VF) in contrast to the 128 entries available in the 82599 chipset. This has the potential to reduce connection imbalances between cores, especially with high core counts. The newly released *Haswell* processors provide per-core DVFS controls, which further increases the Pareto space.

Future work We also plan to explore the synergies between IX and networking protocols designed to support microsecond-level latencies and the reduced buffering characteristics of IX deployments, such as DCTCP [1] and ECN [109]. Note that the IX dataplane is not specific to TCP/IP. The same design principles can benefit alternative, potentially application specific, network protocols, as well as high-performance protocols for non-volatile memory access.

Finally, we will investigate library support for alternative APIs on top of our low-level interface, such as MegaPipe [45], cooperative threading [138], and rule-based models [127]. Such APIs and programming models will make it easier for applications to benefit from the performance and scalability advantages of IX.

2.8 Related Work

We organize the discussion topically, while avoiding redundancy with the commentary in §2.2.3.

Hardware virtualization Hardware support for virtualization naturally separates control and execution functions, e.g., to build type-2 hypervisors [17, 63], run virtual appliances [116], or provide processes with access to privileged instructions [11]. Similar to IX, Arrakis uses hardware virtualization to separate the I/O dataplane from the control plane [103]. IX differs in that it uses a full Linux kernel as the control plane; provides three-way isolation between the control plane, networking stack, and application; and proposes a dataplane architecture that optimizes for both high throughput and low latency. On the other hand, Arrakis uses Barrelfish as the control plane [10] and includes support for IOMMUs and SR-IOV.

Library operating systems Exokernels extend the end-to-end principle to resource management by implementing system abstractions via library operating systems linked in with applications [35]. Library operating systems often run as virtual machines [16] used, for instance, to deploy cloud services [83]. IX limits itself to the implementation of the networking stack, allowing applications to implement their own resource management policies, e.g. via the `libevent` compatibility layer.

Asynchronous and zero-copy communication Systems with exception-less, asynchronous, or batched system calls substantially reduce the overheads associated with frequent kernel transitions and context switches [45, 57, 113, 124]. IX's use of adaptive batching shares similar benefits but is also suitable for low-latency communication. Zero-copy reduces data movement overheads and simplifies resource management [101]. POSIX OSes have been modified to support zero-copy through page remapping and copy-on-write [21]. By contrast, IX's cooperative memory management enables zero-copy without page remapping. Similar to IX, TinyOS passes pointers to packet buffers between the network stack and the application in a cooperative, zero-copy fashion [72]. However, IX is optimized for datacenter workloads, while TinyOS focuses on memory constrained, sensor environments.

Scheduling Scheduler activations [2] give applications greater control over hardware threads and provide a mechanism for custom application-level scheduling. Callisto [48] uses a similar strategy to improve the performance of co-located parallel runtime systems. Our approach differs in that an independent control plane manages the scheduling of hardware threads based on receive queuing latency indicators while the dataplane exposes a simple kernel threading abstraction. SEDA [140] also monitors queuing behavior to make scheduling decisions such as thread pool sizing. Chronos [59] makes use of software-based flow steering, but with a focus on balancing load to reduce latency. Affinity Accept [102] embraces a mixture of software and hardware-based flow steering in order to improve TCP connection affinity and increase throughput. We focus instead on energy proportionality and workload consolidation.

Energy Proportionality The energy proportionality problem [7] has been well explored in previous work. Some systems have focused on solutions tailored to throughput-oriented workloads [87] or read-only workloads [67]. Meisner et. al. [88] highlight unique challenges for low latency workloads and advocate full system active low-power modes. Similar to our system, Pegasus [79] achieves CPU energy proportionality for low latency workloads. Our work expands on Pegasus by exploring the elastic allocation of hardware threads in combination with processor power management states and by basing scheduling decisions on internal latency metrics within a host endpoint instead of an external controller. Niccolini et. al. show that a software router, running on a dedicated machine, can be made energy-proportional [97]. Similar to our approach, queue length is used as a control signal to manage core allocation and DVFS settings. However, we focus on latency-sensitive applications, rather than middlebox traffic, and consider the additional case of workload consolidation.

Co-location Because host endpoints contain some components that are not energy proportional and thus are most efficient when operating at 100% utilization, co-location of workloads is also an important tool for improving energy efficiency. At the cluster scheduler level, BubbleUp [85] and Paragon [26] make scheduling decisions that are interference-aware through efficient classification of the behavior of workload co-location. Leverich et. al. [71] demonstrate that co-location of batch and low latency jobs is possible on commodity operating systems. Our approach explores this issue at higher throughputs and with tighter latency SLOs. Bubble-Flux [144] additionally controls background threads; we control background and latency-sensitive threads. CPI^2 [149] detects performance interference by observing changes in CPI and throttles offending jobs. This work is orthogonal to ours and could be a useful additional signal for our control plane. Heracles manages multiple hardware and software isolation mechanisms, including packet scheduling and cache partitioning, to co-locate latency-sensitive applications with batch tasks while maintaining millisecond SLOs [80]. We limit our focus to DVFS and core assignment but target more aggressive SLOs.

2.9 Conclusion

We described IX, a dataplane operating system that leverages hardware virtualization to separate and isolate the Linux control plane, the IX dataplane instances that implement in-kernel network processing, and the network-bound applications running on top of them. The IX dataplane provides a native, zero-copy API that explicitly exposes flow control to applications. The dataplane architecture optimizes for both bandwidth and latency by processing bounded batches of packets to completion and by eliminating synchronization on multi-core servers.

The dynamic resource controller allocates cores and sets processor frequency to adapt to changes in the load of latency-sensitive applications. The novel rebalancing mechanisms do not impact the steady-state performance of the dataplane and can migrate a set of flow groups in milliseconds without dropping or reordering packets. We develop two resource control policies focused on optimizing energy proportionality and workload consolidation.

On microbenchmarks, IX noticeably outperforms both Linux and mTCP in terms of both latency and throughput, scales to hundreds of thousands of active concurrent connections, and can saturate 4x10GbE configurations using a single processor socket. Finally, we show that porting `memcached` to IX removes kernel bottlenecks and improves throughput by up to 6.1×, while reducing tail latency by more than 1.9×.

We use three varying load patterns to evaluate the effectiveness of our approach to resource control. Our results show that resource controls can save 42%–51% of the processor's energy, or enable a background job to deliver 31%–44% of its standalone throughput. We synthesize the Pareto frontier by combining the behavior of all possible static configurations. Our policies deliver 87%–94% of the Pareto optimal bound in terms of energy proportionality, and 81%–92% in terms of consolidation.

3 Scheduling for dataplane OS

This chapter 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 duration.

We present ZYGOS[1], 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 99^{th} 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 99^{th} 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).

3.1 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 [6]. In such deployments, a single application can comprise hundreds of software components, deployed on thousands of servers organized in multiple tiers and connected by commodity Ethernet switches. Such applications must support high

---

[1] The Greek word for balancing scales.

concurrent connection counts and operate with user-facing SLOs, often defined in terms of tail latency [3, 25, 98]. 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 [130, 126, 137, 38, 139], NoSQL databases [110, 93], key-value stores [31, 77, 89, 148], or specialized graph stores [15].

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 to complete, the SLO of the data services must consider the long tail of the latency distributions of requests [25]. Individual tasks often require only a handful of µs to execute. 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 [8].

This hunt for the killer microseconds [8] requires researchers to revisit assumptions across the network and compute stacks, whose policies and implementations play a significant role in exacerbating the problem [71].

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 [143].

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 [90]) 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 [19] but suffers from high overheads for µs-scale tasks. The latter improves throughput substantially (by up to 6× for key-value stores [13]) 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 [13, 103, 77, 57, 84, 59].

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 [89, 3] with an SLO at $> 100\times$ the mean service time [13]), these 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, 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 [22]. The shuffle layer eliminates head-of-line-blocking while also offering strong ordering semantics of events associated with the same connection.

(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 [11] and handles IPIs in an exit-less manner similar to ELI [42].

(3) A methodology using microbenchmarks with synthetic service times to identify system overheads as a function of task duration 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 (§3.6.1).

(4) We compare ZYGOS to IX, a state-of-the-art dataplane with strict run-to-completion that partitions flows onto cores [13]. 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 (§3.6.1). IX does outperform ZYGOS for workloads with very small task durations such as `memcached`. The difference is primarily due to IX's adaptive bounded batching, which is not currently supported in ZYGOS. (§3.6.2)

(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 [130], 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 (§3.6.3).

The source code of ZYGOS, along with benchmarks, scripts and simulation models, is available in open source [151].

The rest of the chapter is organized as follows: §3.2 provides background on the problem and the theory. §3.3 describes the experimental methodology and characterizes existing systems. We describe the design (§3.4), implementation (§3.5) and evaluation of ZYGOS (§3.6). We discuss a key tradeoff (§3.7), related work (§3.8), and conclude.

3.2 Background

3.2.1 Scaling remote procedure calls

In-memory data services typically expose a remote procedure call (RPC) interface. The problem of efficiently handling incoming RPCs dates back to the original C10k problem [18] 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 [141, 102, 146].

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) [19] and Borrowed Virtual Time (BVT) [32], 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 [75, 76]. 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 [68]. 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` [36].

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` [44] and applications such as recent versions of `nginx` [96] and Apache Lucene [82]. While this pattern may increase the latency of an

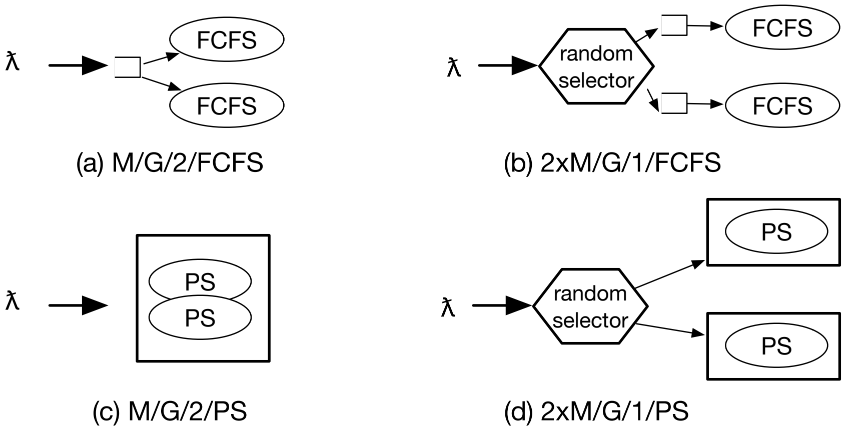


Figure 3.1 – Queuing models for $n = 2$ processors.

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.

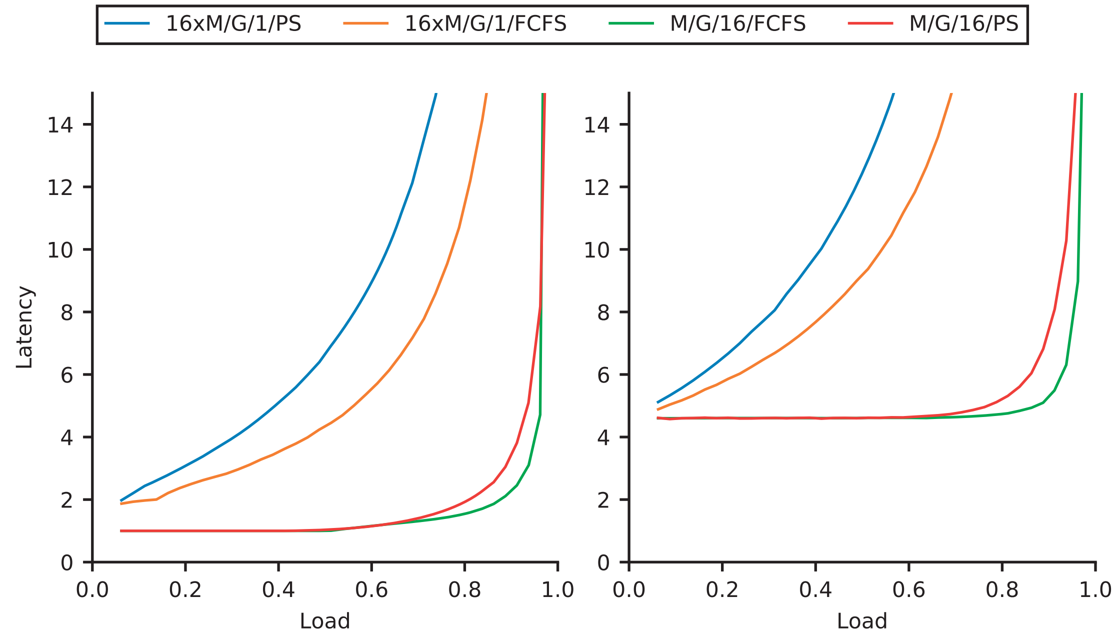
3.2.2 Kernel bypass and sweeping simplifications

Data plane approaches such as IX [13], Arrakis [103] and user-level stacks [120, 57, 30, 112, 104, 128] bypass the kernel and rely on I/O polling to both increase throughput and reduce latency jitter [71, 74]. For example, IX increases the throughput of `memcached` by up to 6.4× over Linux [13].

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 [13]. 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 [77]. 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.

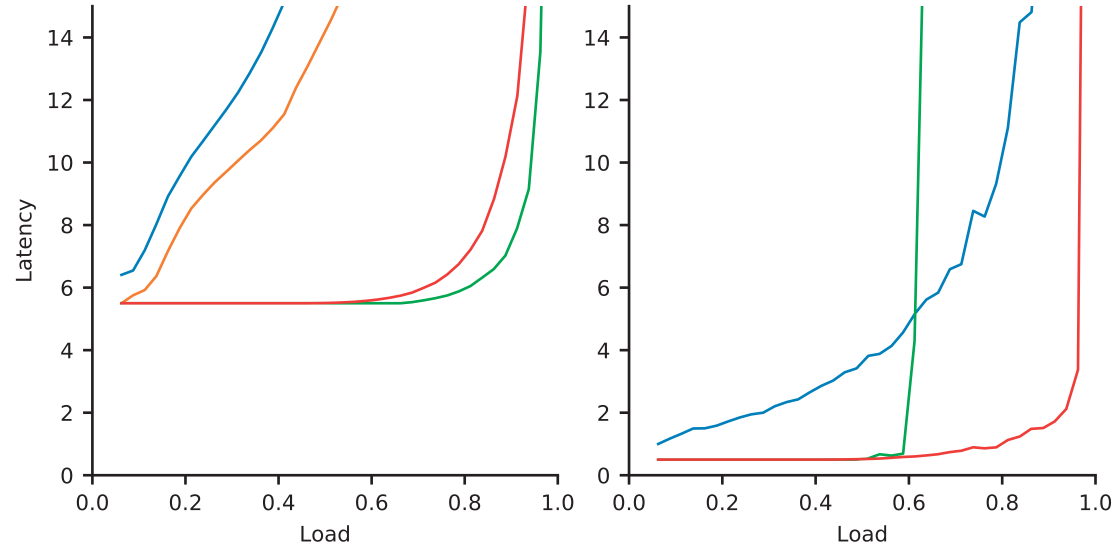
3.2.3 Just enough queuing theory

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



(a) Deterministic

(b) Exponential



(c) Bimodal-1

(d) Bimodal-2

Figure 3.2 – Simulation results for the 99^{th} percentile tail latency for four service time distributions with $\bar{S} = 1$.

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 [77]).

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.

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 3.1 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 §3.2.

- 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 duration 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 3.2 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 [81]:

- 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 3.2 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 99^{th}-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.

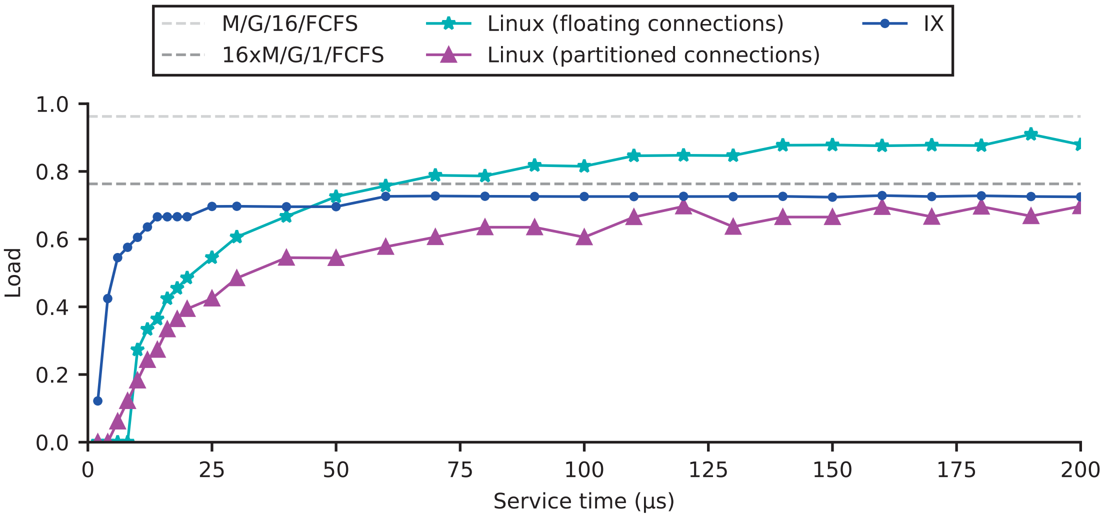
We make two observations that inform our system design:

Observation 1: Single-queue systems (i.e., *M/G/n/) perform better compared to systems with multiple queues (i.e., $n\times$*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.

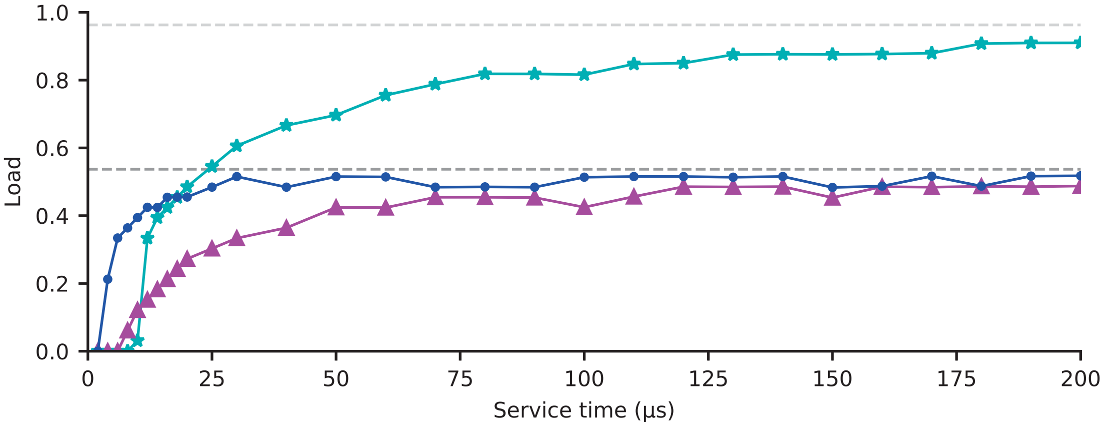
Observation 2: FCFS performs better in regards to tail latency for distributions with low dispersion This result has also been theoretically analyzed by Wierman et al. [143]. In Figure 3.2, FCFS outperforms PS for the deterministic, exponential and bimodal-1 distribution. PS only outperforms FCFS when 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.

3.3 Experimental Methodology

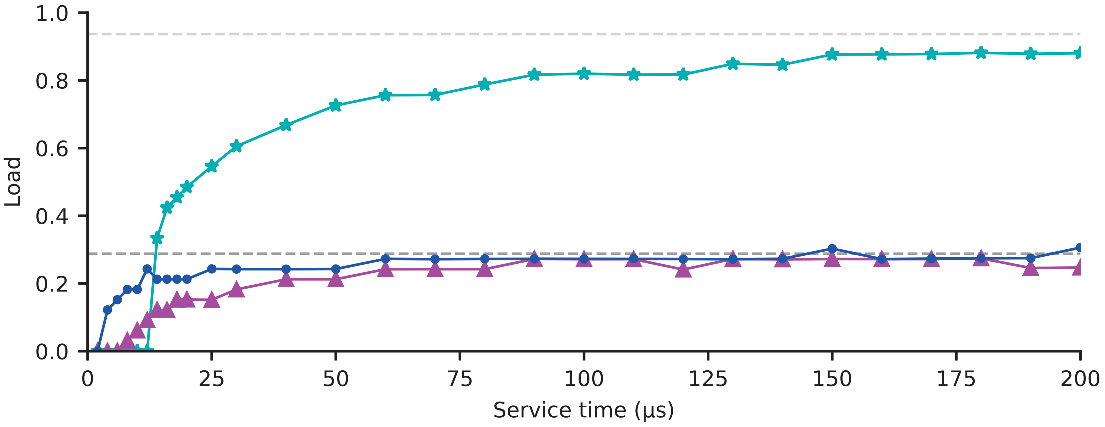
We now describe the experimental methodology used to evaluate existing low-latency systems. The challenge is to define metrics that help determine (a) the inherent design tradeoffs by



(a) Deterministic



(b) Exponential



(c) Bimodal-1

Figure 3.3 – 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.

comparing real-life systems with the idealized models of §3.2.3; 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.

3.3.1 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 [117]. 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 96.2% for centralized-FCFS.

3.3.2 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` [71] 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.

3.3.3 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 [90]) 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*.

Linux configuration The Linux systems were tuned to maximize throughput and minimize latency, 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 [36]). 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 durations. 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.

3.3.4 Baseline results

Figure 3.3 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 §3.3.3. 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

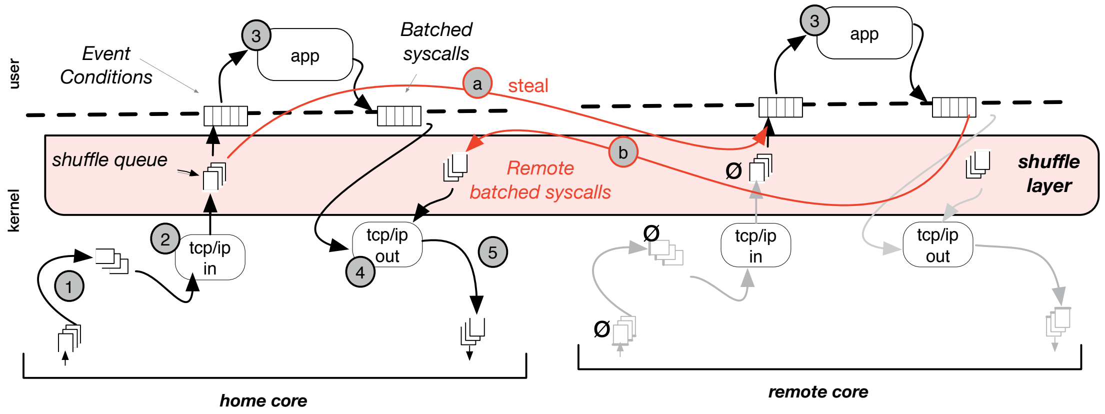


Figure 3.4 – Dataflow in the ZYGOS operating system. Steps (1) – (6) correspond to the normal execution on the home core. Steps (a)-(b) occur during stealing and involve the home and remote cores.

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 3.3 shows the result for three of the four distributions studied analytically in Figure 3.2. We omit the bimodal-2 results as the analysis of §3.2.3 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 $16\times M/G/1$ 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.

3.4 Design

3.4.1 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, 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 §3.4.3. As discussed in §3.2.2, 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.

3.4.2 ZYGOS High-level Design

ZYGOS shares a number of architectural and implementation building blocks with IX [13]: 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 [11]; 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 3.4:

(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 [124]. 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 3.4 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 3.4 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 3.4 is only a high-level illustration of the system. In ZYGOS, each core acts as the home core for a set of TCP connections and can act as the remote core for any other TCP connection whenever it is idle. We now describe the ordering semantics that enable stealing (§3.4.3) and the data structures of the shuffle queue that eliminates head-of-line blocking (§3.4.4).

3.4.3 Ordering semantics in multi-threaded applications

When TCP sockets are statically assigned to threads, applications can rely on intuitive ordering and concurrency semantics [70]. 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 [36], 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 §3.3.3. 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 [60, 61].

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.

3.4.4 Eliminating Head-of-Line Blocking

The ordering semantics of §3.4.3 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 3.5 shows the state machine diagram that controls the decisions for each socket. Changes to the state machine and to the shuffle queue are atomic.

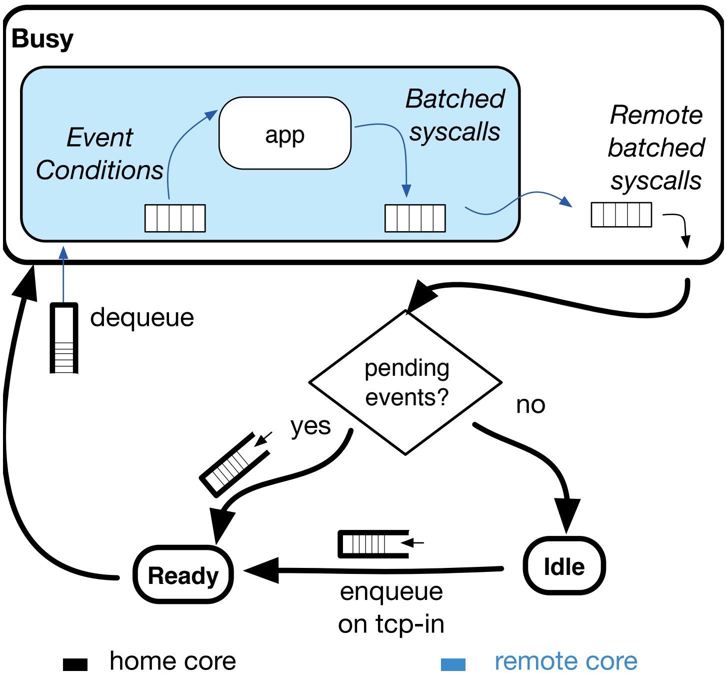


Figure 3.5 – Connection state machine transitions for the general case where an event is executed on a remote core (in blue). The connection is present in the shuffle queue exactly once when it is in the "ready" state, and never otherwise.

- *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 executing on a 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.

3.4.5 Inter-processor Interrupts

The design in §3.4.4 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 3.4. 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 Inter-processor Interrupt (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 3.4). 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 batched 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.

3.5 Implementation

The system architecture of ZYGOS is derived from the IX open-source release v1.0 [105]: it relies on hardware virtualization and the Dune framework [11] to host a protected operating system with direct access to VMX non-root mode ring 0 in the x86-64 architecture [131]. The kernel links in with DPDK [30] for NIC drivers and lwIP for TCP/IP [33]. 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 §3.4.4. 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 [42], we added support in Dune for exit-less interrupts in non-root mode. ZYGOS's interrupt handler processes only interrupt 242 and redirects all other interrupts to the Linux host operating system by performing a specific `vmcall`. 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 [64]. 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 [107]. 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.

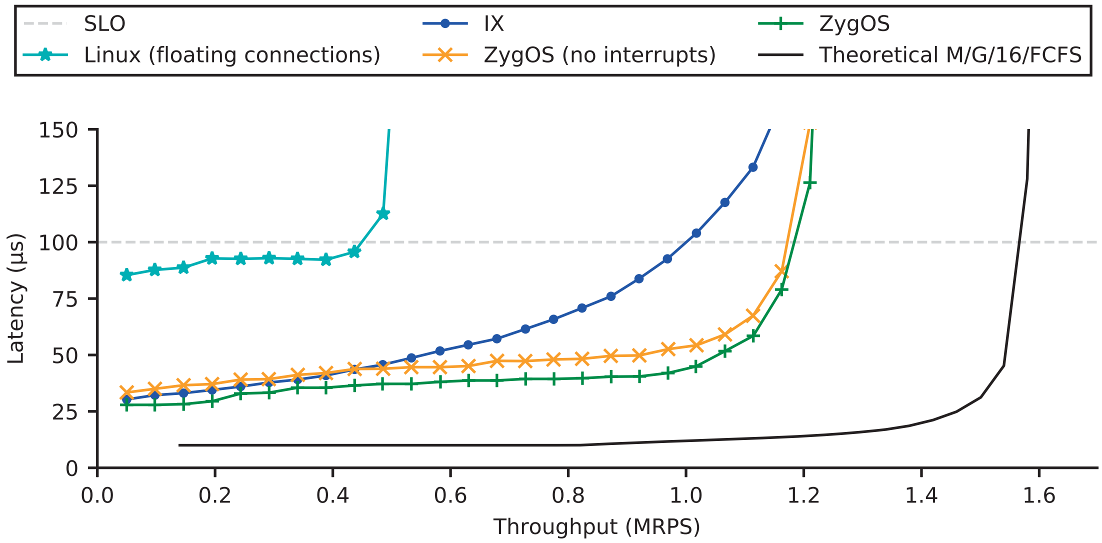
3.6 Evaluation

We use the same experimental setup explained in Section 3.3 to evaluate ZYGOS in a series of microbenchmarks, use `memcached` [89] to evaluate overheads on tiny tasks, and with a real application running TPC-C [129].

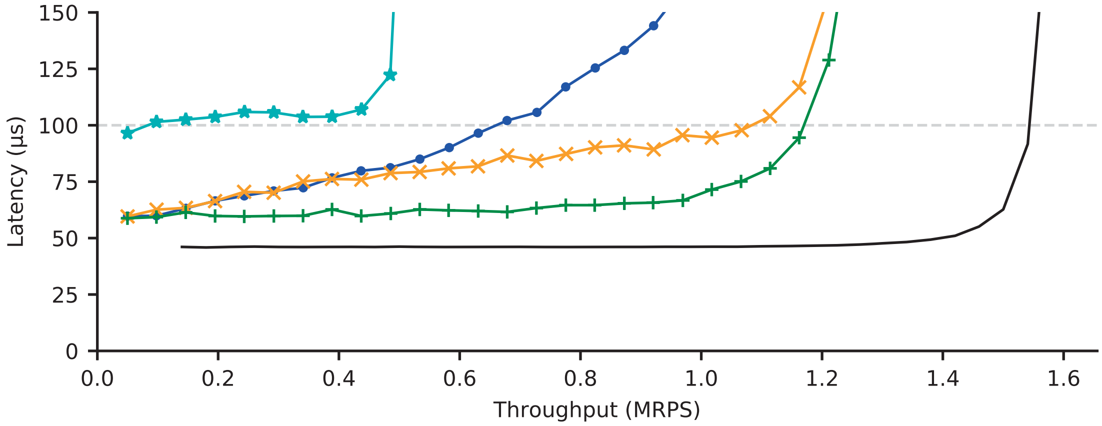
3.6.1 Synthetic micro-benchmarks

Figure 3.6 and Figure 3.7 shows the latency vs. throughput of the three synthetic micro-benchmarks of §3.3. 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:

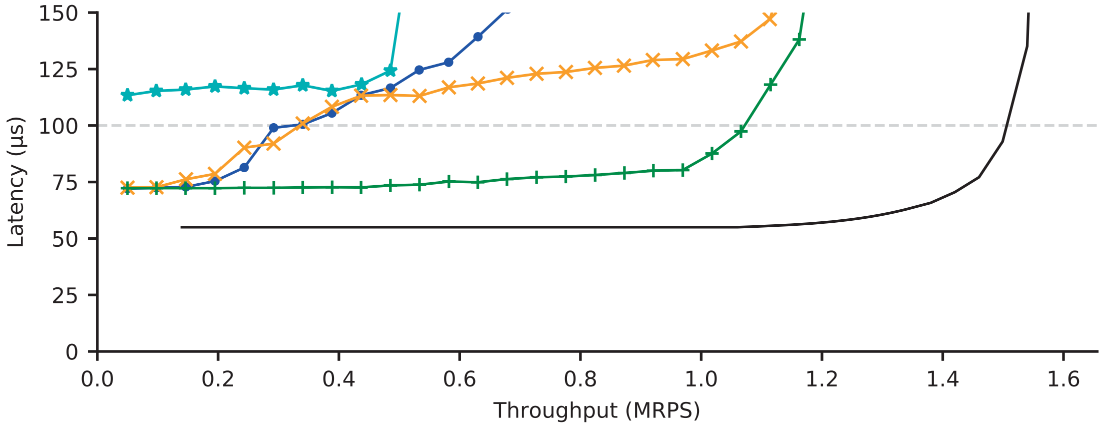
- ZYGOS and Linux-floating both approximate the theoretical model, with ZYGOS substantially reducing tail latency over IX;



(a) Fixed ($\bar{S} = 10\mu s$)

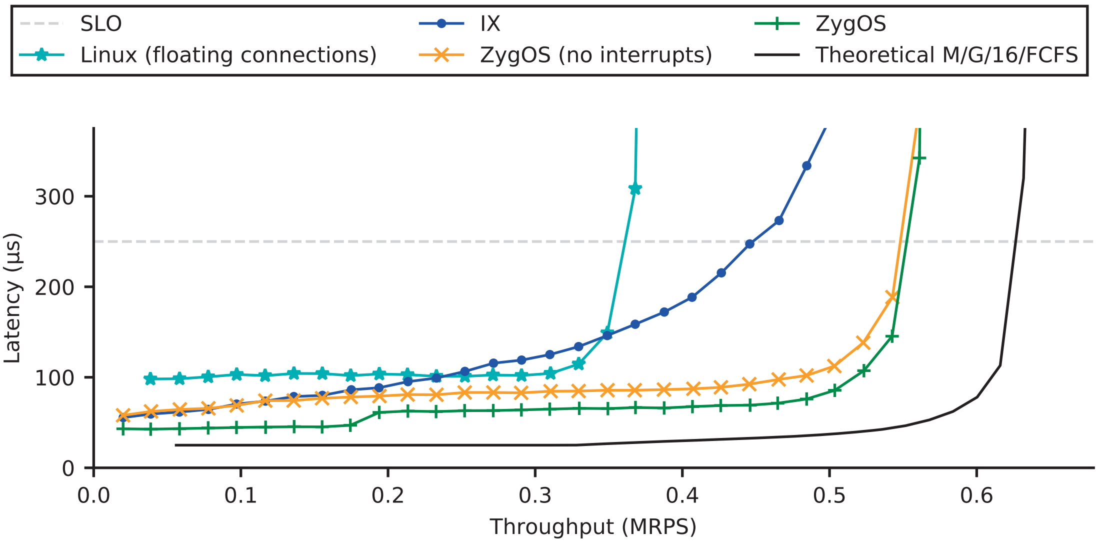


(b) Exponential ($\bar{S} = 10\mu s$)

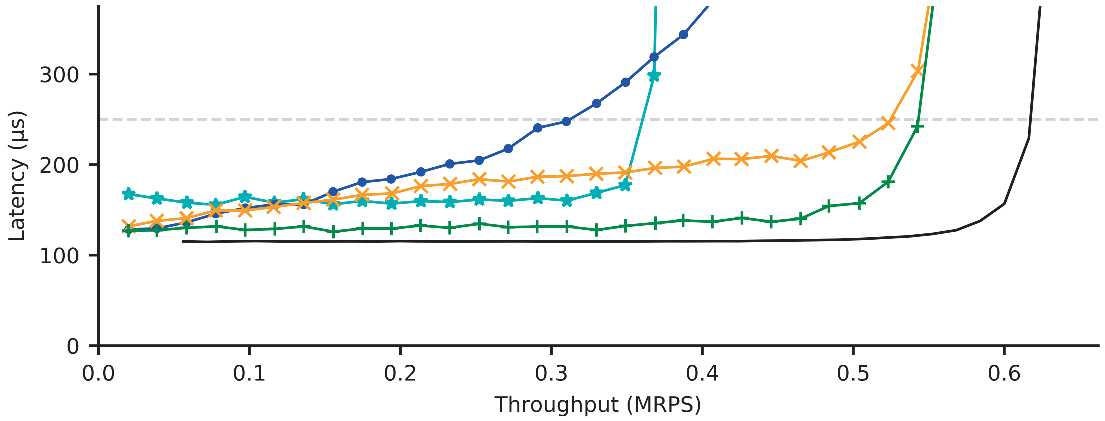


(c) Bimodal-1 ($\bar{S} = 10\mu s$)

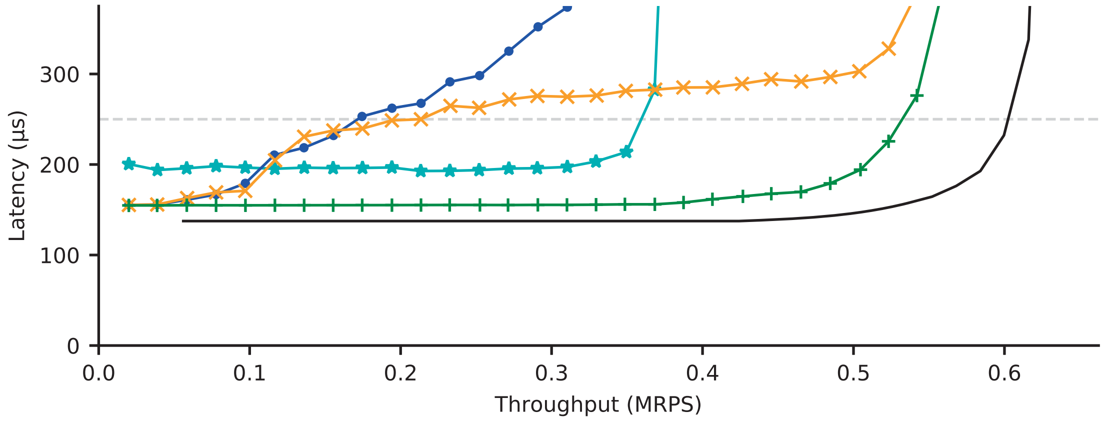
Figure 3.6 – 99^{th} percentile tail latency according to throughput for three distributions with 10µs mean task granularity. The horizontal line corresponds to the SLO of $\leq 10 \times \bar{S}$.



(a) Fixed ($\bar{S} = 25\mu s$)

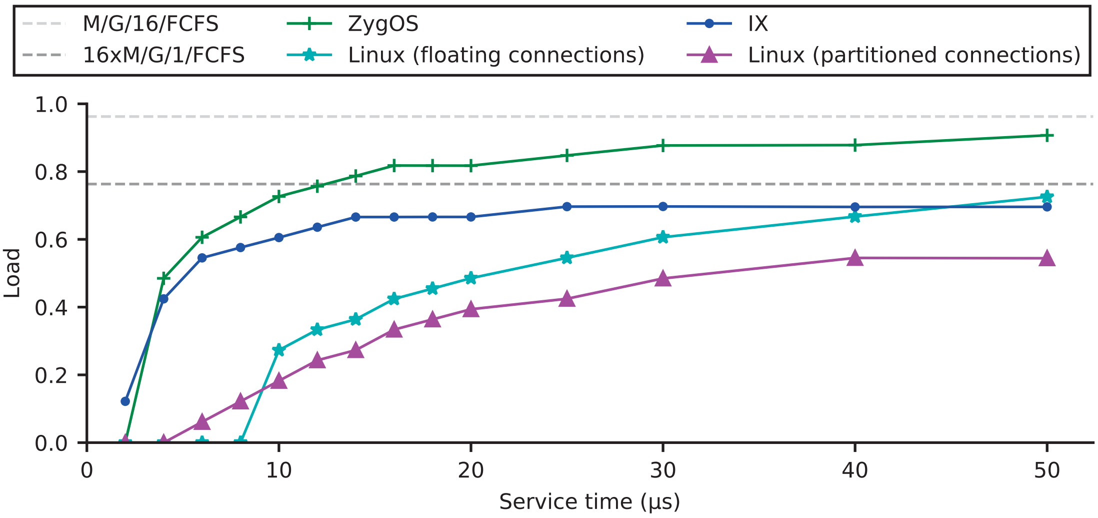


(b) Exponential ($\bar{S} = 25\mu s$)

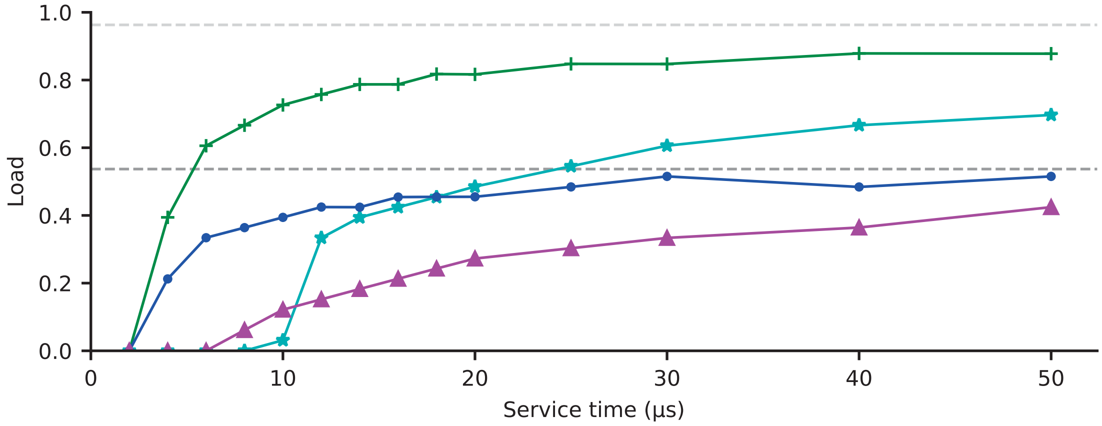


(c) Bimodal-1 ($\bar{S} = 25\mu s$)

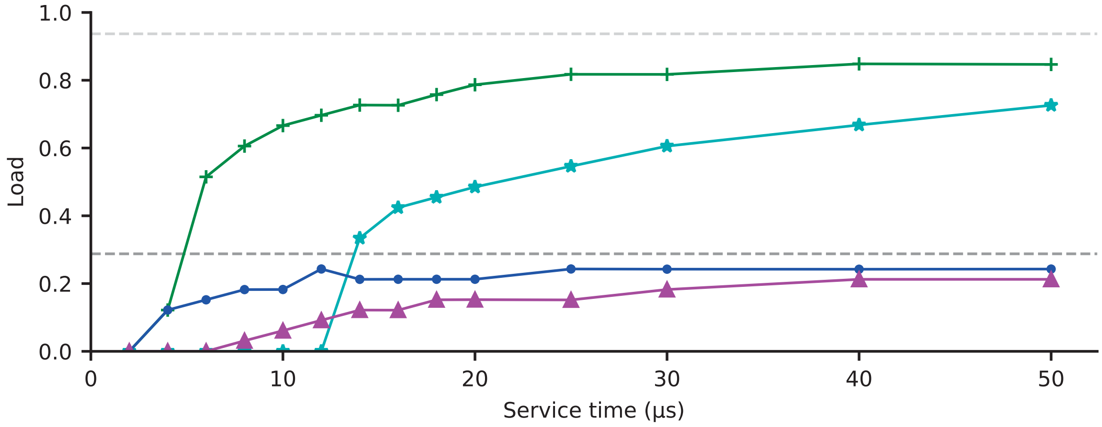
Figure 3.7 – 99^{th} percentile tail latency according to throughput for three distributions with $25\mu s$ mean task granularity. The horizontal line corresponds to the SLO of $\leq 10 \times \bar{S}$.



(a) Deterministic



(b) Exponential



(c) Bimodal-1

Figure 3.8 – 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).

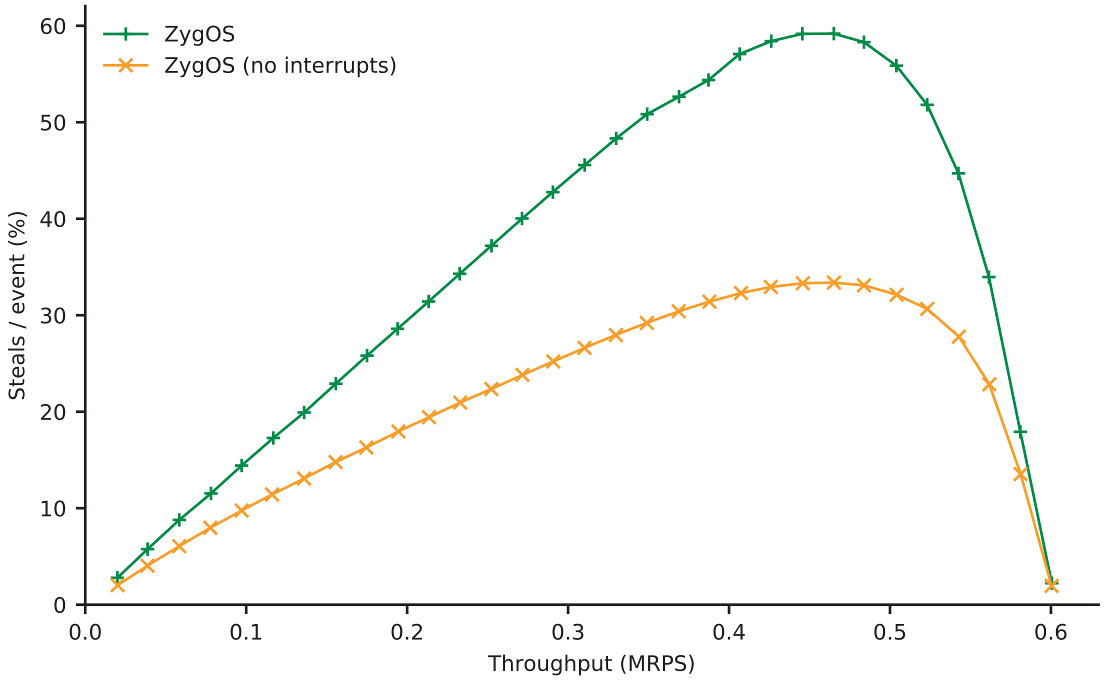


Figure 3.9 – Normalized rate of stealing vs. throughput for exponential service time with mean 25 µs

- 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 × $\bar{S}$ for $\bar{S} = 10\mu s$ (Figure 3.6b) and 88% for $\bar{S} = 25\mu s$ (Figure 3.7b);
- interrupts are necessary to eliminate head-of-line blocking with medium and high dispersion workloads, and the cooperative model of ZYGOS without interrupts, which is typical of pure user-level application, visibly impacts tail latency.

Efficiency for the 10 × $\bar{S}$ tail latency SLO Figure 3.8 reports the efficiency (in terms of max load at SLO) as a function of task duration. We compare ZYGOS with the baseline shown in Figure 3.3. 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 3.9 provides an insight into the rate of stealing events as a function of load. The results are for the exponential distribution of Figure 3.7b 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.

3.6.2 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 duration), and use the methodology and reproduce the results from IX [13]. We consider `memcached` a near worst case for ZYGOS as the application has very small task duration with a small dispersion best approximated by a deterministic distribution.

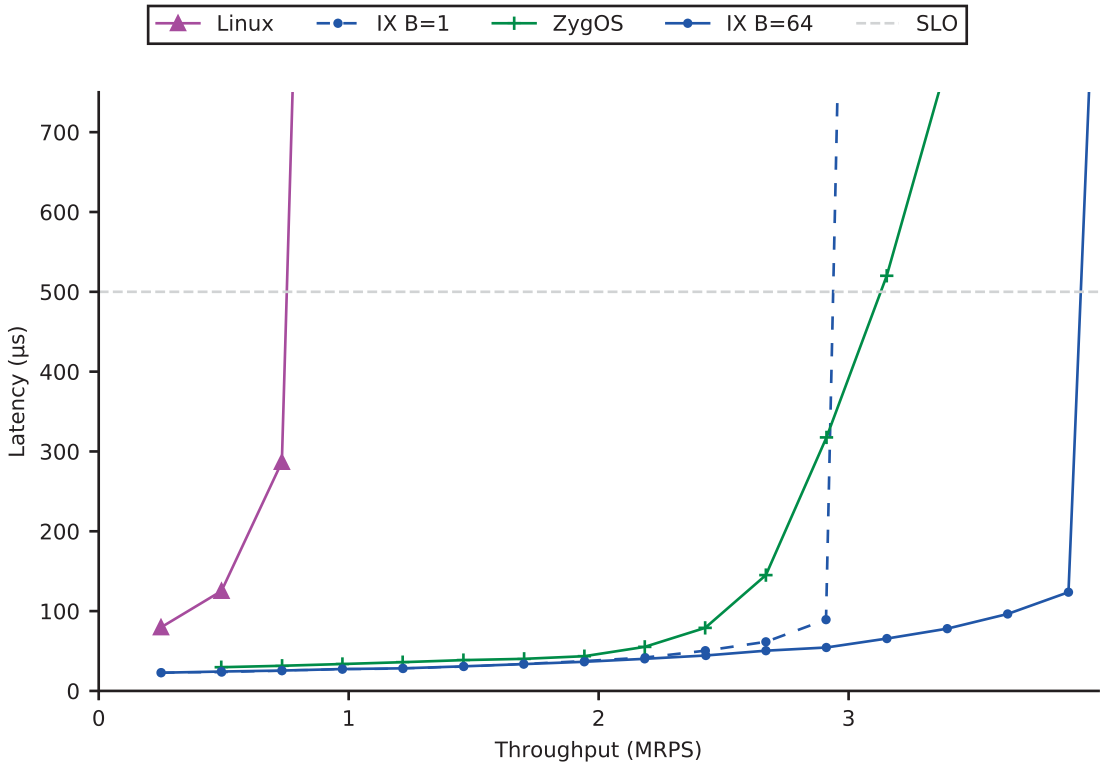
Figure 3.10 shows the latency vs. throughput for the USR and ETC workloads, [3], as modelled by `mutilate` [71]. We compare Linux, ZYGOS, and IX. For IX, we choose two configurations: with adaptive batching disabled (B=1) and with adaptive batching enabled with the default setting (B=64).

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.

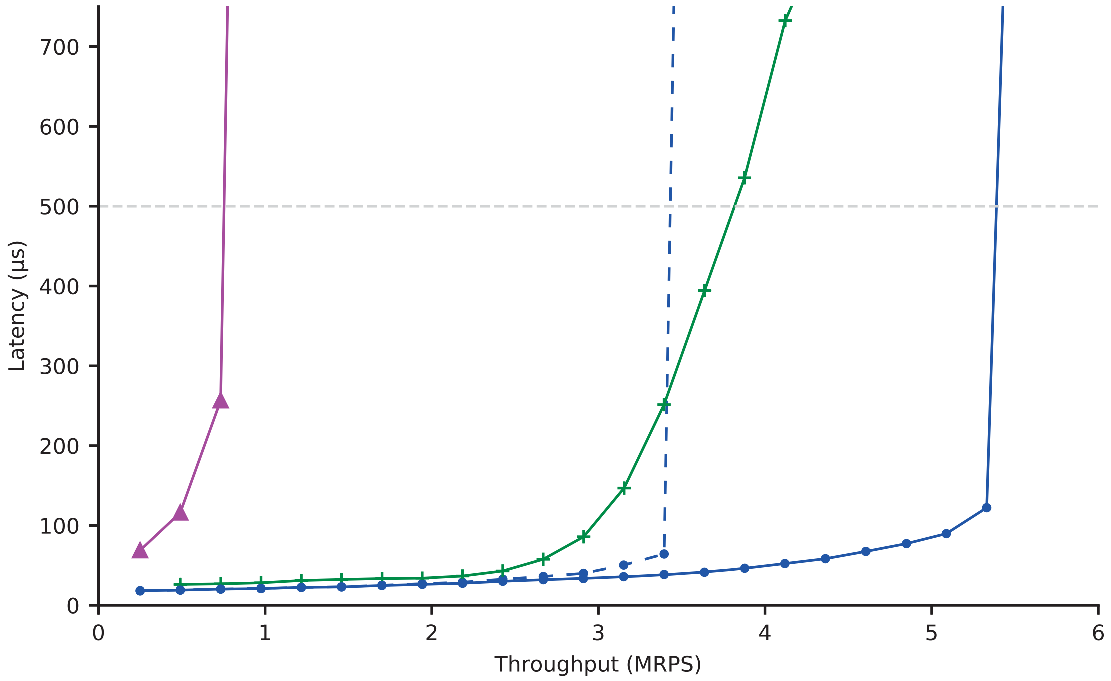
Of note, ZYGOS has a differently shaped latency vs. throughput curve for this workload. As described in §3.4.3 and §3.4.4, ZYGOS does not respect strict FIFO ordering on servicing packets across different connections. For this workload configuration, up to four distinct `memcached` requests can be pipelined 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 behavior 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 [60] can potentially handle this situation.

3.6.3 A real application: Silo running TPC-C

We validate the tail latency benefits of ZYGOS using Silo [130], a state-of-the-art in-memory database optimized for multicore scalability.



(a) ETC



(b) USR

Figure 3.10 – 99^{th} percentile tail latency vs. throughput for two `memcached` workloads for Linux, IX and, ZYGOS.

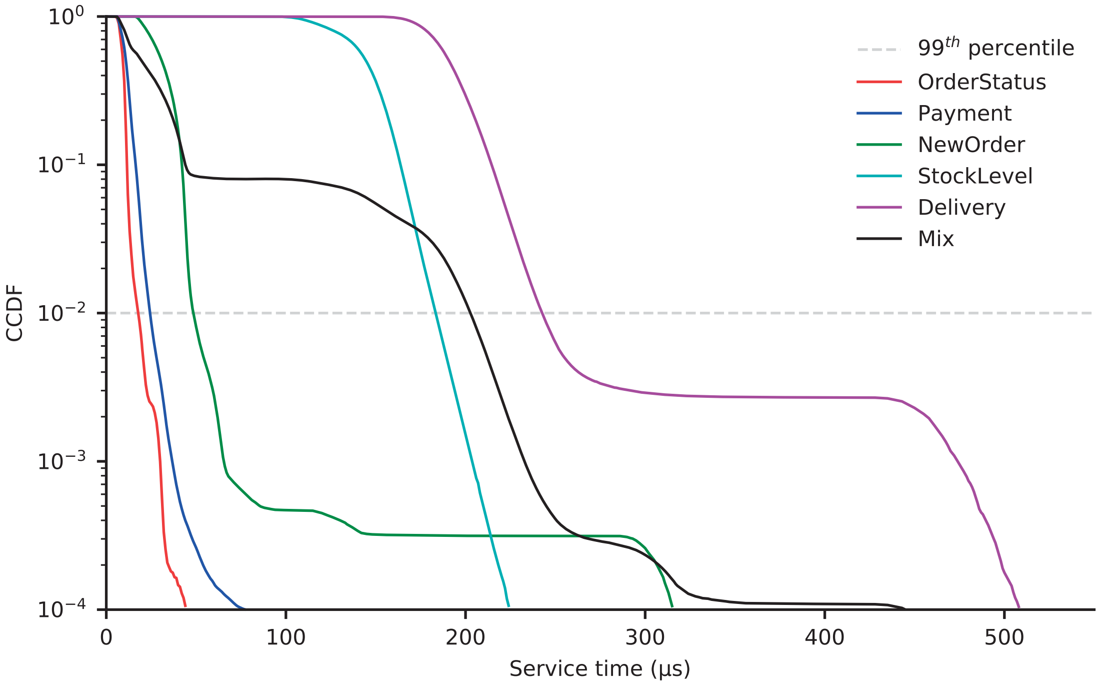


Figure 3.11 – Complementary CDF of task execution time for Silo running the TPC-C benchmark under Linux.

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` [71] with the same setup described in §3.3.2 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)

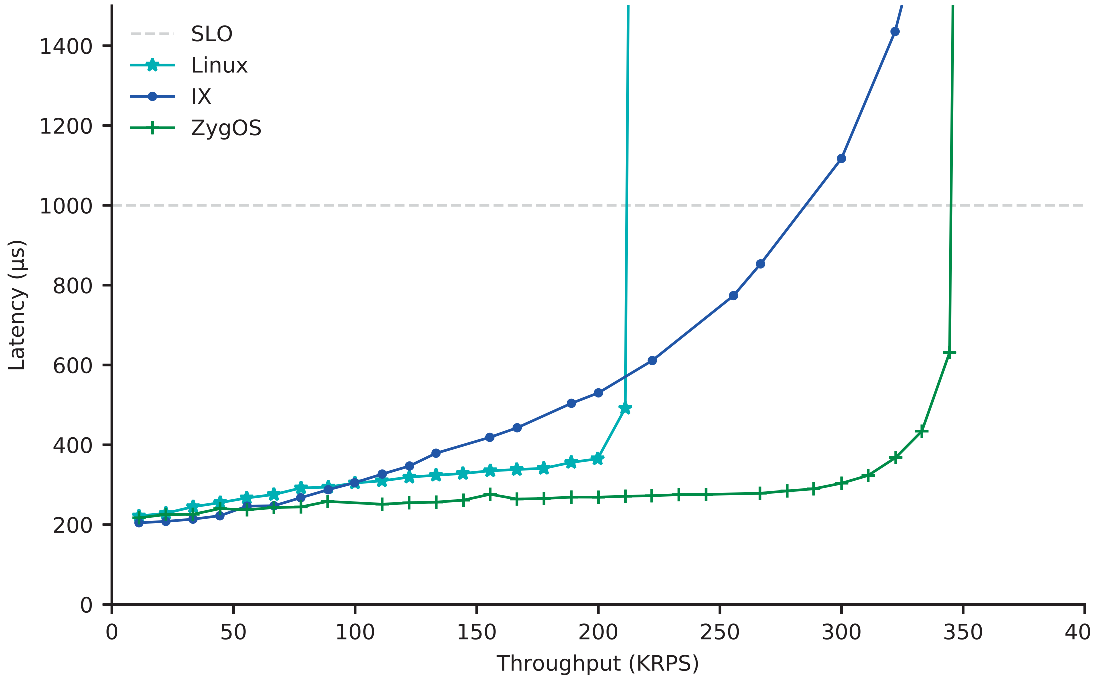


Figure 3.12 – 99^{th} percentile end-to-end latency vs. throughput for Silo running the TPC-C benchmark

Results

Figure 3.11 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 [122], 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 the end-to-end latency (i.e., it excludes any queueing delays).

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

Table 3.1 – 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 99^{th} percentile end-to-end latency to Silo's 99^{th} percentile service time (203µs).

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

Figure 3.12 shows the tail latency at the 99^{th} 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 99^{th} percentile tail latency. We observe:

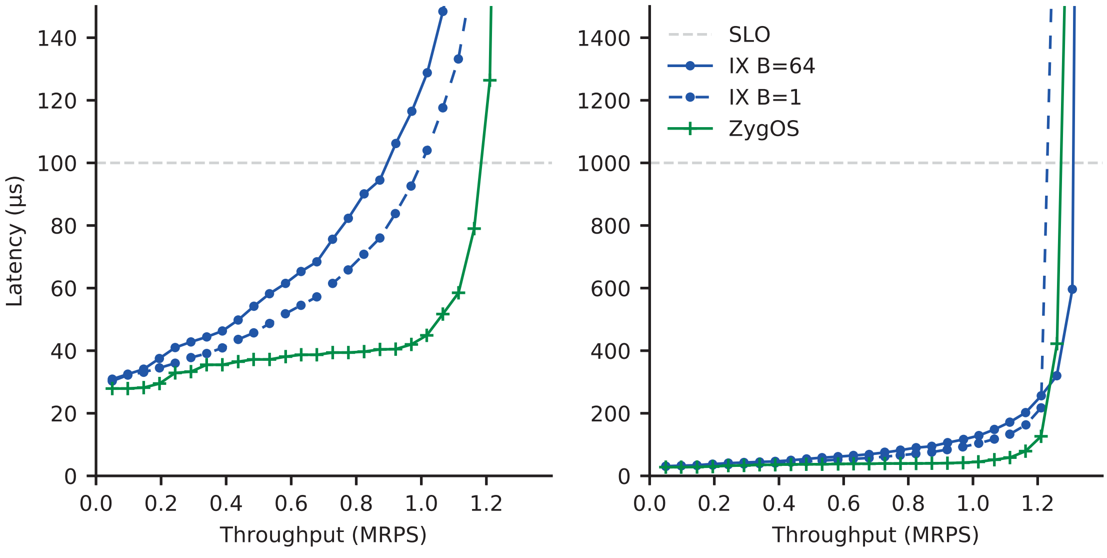
- ZYGOS can support 344 KTPS without violating the SLO; this corresponds to a speedup of 1.63× over Linux. This demonstrates the benefits of our approach for real-life in-memory applications. The achieved transaction rate corresponds to 75% of the ideal, zero-overhead load with no SLO restrictions.
- 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 3.1 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.

3.7 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 3.13 illustrates the tradeoff through the latency vs. throughput curves for the synthetic benchmark of §3.6.1 with an exponential service time of $\bar{S}$ =10 µs. Figure 3.13a and 3.13b 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 3.13a, $10 \times \bar{S}$) 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.



(a) 100 µs SLO

(b) 1000 µs SLO

Figure 3.13 – Comparison of IX (batch size 1 and 64) and ZYGOS for a deterministic service time of 10 µs and 2 different SLOs.

However, for a more lenient SLO (Figure 3.13b, 100 × $\bar{S}$), IX's adaptive batching delivers marginally higher throughput than ZYGOS before violating the SLO.

3.8 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 `EPOLLEXCLUSIVE` eliminates thundering herds [36], 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 [102] 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. [46] 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 Standard operating system pre-emptive schedulers, such as CFS [19] and BVT [32], favor latency-sensitive tasks. Applications can benefit from multi-threading 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 [103], IX [13], mTCP [57], MICA [77], Seastar [120] and Sandstorm [84] bypass the kernel (via frameworks such as DPDK [30] or netmap [112]) 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 [95] and Reflex [65] 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 [99]. 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., [20, 23, 28]. Statically mapping connections to cores can result in load imbalance in event-based programs and requires a solution at the library level [41, 147]. Recent focus on work stealing for latency-critical applications is at coarser timescales. [73, 145, 47]. The prior work largely targets applications with millisecond-scale task granularities that are easily accommodated by conventional operating systems. ZYGOS implements work-stealing 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. [81] 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 [100] also relies on power-of-two choices for batch job scheduling. Google's Maglev [34] is a generic distributed network load balancer that leverages consistent hashing to load balance packets across the corresponding services.

3.9 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.

4 Conclusion

4.1 What we did

This thesis examines the use of dataplane operating systems in the deployment of modern datacenter scale web applications. The state of the art dataplane tackles many inefficiencies of the networking stacks of standard operating systems. Nevertheless, the use of a dataplane introduces new challenges, such as increased energy usage and µs-scale scheduling decisions. The adoption of dataplanes in modern datacenters relies on the resolution of such challenges.

In this thesis, we propose two systems that address those challenges. The first system deals with the energy efficiency of a dataplane and proves that it is possible to run a dataplane without wasting excessive energy. The second system introduces a scheduler that operates on the µs-scale and is able to reduce the tail latency and alleviate head-of-line blocking.

The first contribution is a system that provides resource management to the IX dataplane operating system. The goal of that solution is to decrease energy usage and/or enable work consolidation. In order to achieve that goal, we start by doing an exhaustive analysis of the power vs. performance tradeoff for our workload of interest on our hardware. This analysis allows us to derive the optimal resource allocation strategy. The strategy maps the dataplane's load (expressed in requests per second) to a specific resource configuration (expressed in number of cores utilized and operating frequency). Then, we present the design and implementation of a control agent which monitors queueing delays and adjusts resource usage based on the selected strategy. Finally, we evaluate the control plane by providing time-varying load patterns and measuring the latency and power.

The second contribution is a system that provides µs-scale scheduling for the IX dataplane operating system. Initially, we conduct an extensive study regarding the overheads as a function of the task duration for Linux and IX. This study proves that existing dataplanes, such as IX, outperform commodity operating system, such as Linux, for short task duration that exhibit low dispersion. We recognize that Linux possesses a sophisticated scheduler which gives it performance advantage as the task duration increases and the system overheads are not anymore the bottleneck. We designed and implement ZYGOS which combines IX with a work-conserving scheduler. The basic operating principle is the use of work stealing in order to mitigate the transient imbalances in queue depths among different cores. Additionally, ZYGOS uses inter-processor interrupts (IPI) in order to further mitigate the head-of-line blocking which occurs when a core is busy processing a long request. Finally, we evaluate ZYGOS and compare it against Linux and IX on two different applications.

4.2 Future work

Despite the ongoing effort, there are a few missing core features before dataplanes become ubiquitous. In this section, we will describe a few of them which are also indications of future work for this thesis.

- This thesis discusses work on two axes: (a) coarse grain allocation of resources for energy efficiency, (b) µs-scale scheduling for work conservation. It is possible to combine those two aspects in a single system that achieves both goals. In this effort, we must rethink the control plane so that it takes into account stolen TCP connections when it decides to scale up or down the number of CPU cores. In addition, we must evaluate the performance gain of work stealing when the system operates on a few cores, as we expect that the benefits will be smaller in these cases. We anticipate that combining this thesis into a single system is a worthwhile exercise.

- Throughout the duration of this thesis, the Linux kernel networking stack has received numerous updates that improved its performance, reduced scalability problems, and added more features for latency critical applications. We believe that this stream of improvements is partially based on the parallel research effort on dataplanes and expect that it will continue in the future. We are confident that it makes sense to incorporate specific ideas from this thesis into a conventional operating system, such as Linux. A potential first attempt can be the introduction of a thread pool mechanism inside the Linux kernel which is controlled by the load of incoming network requests. As the load increases, the kernel spawns additional threads on more cores and reconfigures the NIC to forward packets to those cores. Another possibility is the introduction of a power governor that is controlled by the latency of request-response pairs and adjusts the frequency of the cores where a latency critical application is executing. We realize that adding features in a mainstream operating system is a process that requires a lot of coordination, effort, and time. We expect, though, that as hardware becomes faster and provides more functionality, all operating systems must improve in order to expose more performance and features to their applications.

- Currently, the IX dataplane operating system does not support the `fork` system call. This limitation stems from the use of Dune, which does not support multiple address spaces. If a Dune process spawns a child process, then the new process is created outside of

Dune mode and consequently outside of the dataplane. While certain applications (such as `memcached`) use threads to exploit parallelism, other applications (such as `apache`) use processes to achieve the same goal. We believe that adding support for multiple address spaces in Dune would allow more applications to profit from the benefits of dataplanes.

- Currently, dataplanes are restricted to execute a single application. Additionally, if the machine is not using a PCI virtualization technology, such as SR-IOV, the restriction is even more severe: single dataplane per physical NIC. The reason is that a dataplane requires explicit hardware access to the NIC. Often running a single latency sensitive application on a single server is adequate. There are cases, though, where it is feasible to colocate two or more latency critical applications on the same server. Multiplexing applications in such a way is very common with traditional operating systems. A commodity operating system has the necessary mechanisms to provide scheduling and fairness across multiple applications which access a single hardware device, such as a NIC. It is our belief that such mechanisms must be also implemented in dataplanes. In doing so, we expect side benefits such as improved energy efficiency because of the consolidation of applications.

- The IX dataplane operating system supports the standard network protocols, namely IP, TCP, and UDP. In the last few years, there is an effort to design and adopt new protocols that are more suited to each application. A prominent example is QUIC [69], which is a protocol designed and deployed by Google and according to their estimates accounts for 7% of Internet traffic. Regarding the potential use of dataplanes, we can observe that they host RPC-style services most of the time. We expect that it is worthwhile to explore the possibility of designing a new protocol to support such services, in order to avoid the overheads of TCP and handle the missing features of UDP. We anticipate that using such a newly designed protocol within a dataplane operating system will lead to further improvements in performance.

- At present, the IX dataplane operating system supports only a limited number of network interface cards, including the Intel X520 adapter and the Intel XL710 family of adapters. The engineering effort to support a new device is split in two parts: (a) the first part is the device initialization which is handled exclusively by DPDK and (b) the second part is the device receive and transmit path which are modified to fit in the IX pipeline. It is worth noting that support for the Intel XL710 family has been added during the project and based on our experience from that effort, we argue that supporting more devices is an engineering effort of easy to medium difficulty, as long as DPDK already supports these devices.

- An extension of the above point is the fact that IX does not support RDMA. This is not a fundamental design limitation and IX can support RDMA by simply supporting a respective network adapter. In fact, the basic design principles for RDMA and IX are similar, such as kernel bypass and user-space networking. RDMA operation is usually classified as one-sided or two-sided. We clarify the relationship of these modes to IX:

  - Support for the two-sided RDMA primitives is straight forward, since in essence it is similar to the receive and transmit operations of a conventional network adapter. We expect that supporting RDMA-capable hardware will further reduce the latency that IX can offer by a couple of µs. This reduction will mostly benefit applications and workloads with short service times.
  - On the other hand, one-sided RDMA operations assume that there is no executable code running on the server. Therefore, support for one-sided RDMA primitives is relevant only if IX is deployed as a client. It is important to mention that developers must rewrite their applications in order to utilize one-sided RDMA. IX is an operating system and as such one of its main design goals is to accelerate existing applications without requiring extensive modifications to them. It is up to the developers of an application to decide if they want to operate with one-side RDMA primitives. If they do, then IX can certainly run on the client side of the application.

- We performed the energy proportionality part of this thesis on an Intel Sandy Bridge processor that has a single power domain for the whole CPU package. This CPU allowed us to perform an exhaustive analysis of all the possible power configurations, but at the same time limited the amount of available power levels. Newer processors (such as Intel Haswell) provide an independent power domain per core. This technology introduces new challenges regarding our work in energy efficiency. First, it expands the configuration space beyond the point where exhaustive analysis is possible. Thus, we have to come up with some heuristics to prune the configuration space. Second, the power efficiency (or the workload consolidation) policy will involve more steps, which potentially means that changes in frequency will happen more frequently. We must evaluate whether the existing control loop algorithm will be able to take decisions that are more frequent. We believe that it makes sense to design and implement an extension of the dynamic resource control system for such processors, because it will make the load vs. power curve even more proportional.

Bibliography

[1] M. Alizadeh, A. G. 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.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, 1992.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. 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*, pages 53–64, 2012.

[4] L. A. Barroso. Warehouse-Scale Computing: Entering the Teenage Decade. *SIGARCH Computer Architecture News*, 39(3), 2011.

[5] L. A. Barroso. Three things that must be done to save the data center of the future (ISSCC 2014 Keynote). http://www.theregister.co.uk/Print/2014/02/11/google_research_three_things_that_must_be_done_to_save_the_data_center_of_the_future/, 2014.

[6] L. A. Barroso, J. Clidaras, and U. Hölzle. *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.

[7] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

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

[9] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 237–248, 2013.

[10] A. Baumann, P. Barham, P.-É. Dagand, T. L. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore

systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[11] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–348, 2012.

[12] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI)*, pages 49–65, 2014.

[13] 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 Trans. Comput. Syst.*, 34(4):11:1–11:39, 2017.

[14] S. M. Bellovin. A Look Back at "Security Problems in the TCP/IP Protocol Suite". In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, pages 229–249, 2004.

[15] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[16] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, 1997.

[17] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, 2012.

[18] 10 thousand connections problem. http://www.kegel.com/c10k.html, 1999.

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

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

[21] H.-K. J. Chu. Zero-Copy TCP in Solaris. In *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, pages 253–264, 1996.

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

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

[24] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–48, 2013.

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

[26] C. Delimitrou and C. Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Data centers with Paragon. *IEEE Micro*, 34(3):17–30, 2014.

[27] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 127–144, 2014.

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

[29] M. Dobrescu, N. Egi, K. J. Argyraki, B.-G. Chun, K. R. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 15–28, 2009.

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

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

[32] K. J. Duda and D. R. Cheriton. 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)*, pages 261–276, 1999.

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

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

[35] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, 1995.

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

[37] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 371–384, 2013.

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

[39] M. Fisk and W. Feng. Dynamic Adjustment of TCP Window Sizes. Technical report, Tech. Rep. Los Alamos Unclassified Report (LAUR) 00-3221, Los Alamos National Laboratory, 2000.

[40] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. Netw.*, 1(4):397–413, 1993.

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

[42] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. 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)*, pages 411–422, 2012.

[43] R. Graham. The C10M Problem. http://c10m.robertgraham.com, 2013.

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

[45] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 135–148, 2012.

[46] N. Hanford, V. Ahuja, M. Balman, M. K. Farrens, D. Ghosal, E. Pouyoul, and B. Tierney. 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*, pages 1:1–1:10, 2013.

[47] M. E. Haque, Y. H. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. 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)*, pages 161–175, 2015.

[48] T. Harris, M. Maas, and V. J. Marathe. Callisto: co-scheduling parallel runtime systems. In *Proceedings of the 2014 EuroSys Conference*, pages 24:1–24:14, 2014.

[49] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[50] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. F. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 21st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 271–282, 2015.

[51] R. Huggahalli, R. R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA)*, pages 50–59, 2005.

[52] Intel Corp. Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/open-source-kernel-enhancements-paper.pdf, 2013.

[53] Intel Corp. Intel 82599 10 GbE Controller Datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf, 2014.

[54] Intel Corp. Intel Ethernet Controller XL710 Datasheet. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf, 2014.

[55] The ix project, v1.0. https://github.com/ix-project/, 2016.

[56] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. o. Park. mTCP source code release, v. of 2014-02-26. https://github.com/eunyoung14/mtcp, 2014.

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

[58] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing (ICPP)*, pages 743–752, 2011.

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

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

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

[62] W. Kim, M. S. Gupta, G.-Y. Wei, and D. M. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the 14th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 123–134, 2008.

[63] A. Kivity. KVM: The Linux Virtual Machine Monitor. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS)*, pages 225–230, July 2007.

[64] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

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

[66] E. Kohler, R. T. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[67] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. E. Culler, and R. H. Katz. NapSAC: design and implementation of a power-proportional web cluster. *Computer Communication Review*, 41(1):102–108, 2011.

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

[69] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. R. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 183–196, 2017.

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

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

[72] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. A. Brewer, and D. E. Culler. The Emergence of Networking Abstractions and Techniques in TinyOS. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2004.

[73] 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 (PPoPP), pages 14:1–14:13, 2016.

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

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

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

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

[78] G. Linden. Make data useful. https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxnbGluZGVufGd4OjVmZDIzMWMzMGI4MDc3OWM, 2006.

[79] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312, 2014.

[80] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 450–462, 2015.

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

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

[83] A. Madhavapeddy, R. Mortier, C. Rotsos, D. J. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*, pages 461–472, 2013.

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

[85] J. Mars, L. Tang, K. Skadron, M. L. Soffa, and R. Hundt. Increasing Utilization in Modern Warehouse-Scale Computers Using Bubble-Up. *IEEE Micro*, 32(3):88–99, 2012.

[86] P. E. McKenney and J. D. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[87] D. Meisner, B. T. Gold, and T. F. Wenisch. The PowerNap Server Architecture. *ACM Trans. Comput. Syst.*, 29(1):3:1–3:24, 2011.

[88] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 319–330, 2011.

[89] memcached – a distributed memory object caching system. http://memcached.org, 2014.

[90] Microsoft Corp. Receive Side Scaling. http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx, 2014.

[91] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 103–114, 2013.

[92] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

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

[94] Mutilate. https://github.com/ix-project/mutilate, 2014.

[95] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 17–33, 2017.

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

[97] L. Niccolini, G. Iannaccone, S. Ratnasamy, J. Chandrashekar, and L. Rizzo. Building a Power-Proportional Software Router. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 89–100, 2012.

[98] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[99] J. K. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. M. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, 2015.

[100] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[101] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: a unified I/O buffering and caching system. *ACM Trans. Comput. Syst.*, 18(1):37–66, 2000.

[102] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 2012 EuroSys Conference*, pages 337–350, 2012.

[103] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.*, 33(4):11:1–11:30, 2016.

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

[105] G. Prekas, A. Belay, M. Primorac, A. Klimovic, S. Grossman, M. Kogias, B. Gütermann, C. Kozyrakis, and E. Bugnion. IX Open-source version 1.0 – Deployment and Evaluation Guide. Technical report, EPFL Technical Report 218568, 2016.

[106] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[107] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the 2015 ACM Symposium on Cloud Computing (SOCC)*, pages 342–355, 2015.

[108] N. Provos and N. Mathewson. libevent: an event notification library. http://libevent.org, 2003.

[109] K. Ramakrishnan, S. Floyd, D. Black, et al. The Addition of Explicit Congestion Notification (ECN) to IP. IETF RFC 3168, 2001.

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

[111] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC)*, page 7, 2012.

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

[113] L. Rizzo. Revisiting network I/O APIs: the netmap framework. *Commun. ACM*, 55(3):45–51, 2012.

[114] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2):20–27, 2012.

[115] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's Time for Low Latency. In *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)*, 2011.

[116] C. P. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow, M. S. Lam, and M. Rosenblum. Virtual Appliances for Deploying and Maintaining Software. In *Proceedings of the 17th Large Installation System Administration Conference (LISA)*, pages 181–194, 2003.

[117] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[118] E. Schurman and J. Brutlag. Performance related changes and their user impact. http://assets.en.oreilly.com/1/event/29/The%20User%20and%20Business%20Impact%20of%20Server%20Delays,%20Additional%20Bytes,%20and%20HTTP%20Chunking%20in%20Web%20Search%20Presentation.pptx, 2009.

[119] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 2013 EuroSys Conference*, pages 351–364, 2013.

[120] ScillaDB Project. Seastar – high-performance service-application framework. https://github.com/scylladb/seastar/.

[121] P. Shinde, A. Kaufmann, T. Roscoe, and S. Kaestle. We Need to Talk About NICs. In *Proceedings of The 14th Workshop on Hot Topics in Operating Systems (HotOS-XIV)*, 2013.

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

[123] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*, volume 6, 1996.

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

[125] Solarflare Communications. Introduction to OpenOnload: Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. http://www.solarflare.com/content/userfiles/documents/solarflare_openonload_intropaper.pdf, 2011.

[126] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. 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)*, pages 1150–1160, 2007.

[127] R. Stutsman and J. K. Ousterhout. Toward Common Patterns for Distributed, Concurrent, Fault-Tolerant Code. In *Proceedings of The 14th Workshop on Hot Topics in Operating Systems (HotOS-XIV)*, 2013.

[128] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska. Implementing Network Protocols at User Level. In *Proceedings of the ACM SIGCOMM 1993 Conference*, pages 64–73, 1993.

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

[130] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, 2013.

[131] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, 2005.

[132] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[133] G. Varghese and A. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, pages 25–38, 1987.

[134] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proceedings of the ACM SIGCOMM 2009 Conference*, pages 303–314, 2009.

[135] 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 2015 EuroSys Conference*, pages 18:1–18:17, 2015.

[136] W. Vogels. Beyond Server Consolidation. *ACM Queue*, 6(1):20–26, 2008.

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

[138] J. R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. A. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 268–281, 2003.

[139] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015.

[140] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

[141] WhatsApp, Inc. 1 million is so 2011. https://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011, 2012.

[142] D. A. Wheeler. SLOCCount, v2.26. http://www.dwheeler.com/sloccount/, 2001.

[143] A. Wierman and B. Zwart. Is Tail-Optimal Scheduling Possible? *Operations Research*, 60(5):1249–1257, 2012.

[144] H. Yang, A. D. Breslow, J. Mars, and L. Tang. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 607–618, 2013.

[145] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 309–322, 2016.

[146] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 43–56, 2016.

[147] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazières, and M. F. Kaashoek. Multiprocessor Support for Event-Driven Programs. In *USENIX Annual Technical Conference*, pages 239–252, 2003.

[148] H. Zhang, M. Dong, and H. Chen. 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)*, pages 167–180, 2016.

[149] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI. In *Proceedings of the 2013 EuroSys Conference*, pages 379–391, 2013.

[150] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. Goiri, and R. Bianchini. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 755–770, 2016.

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

George Prekas

Data Center Systems Laboratory
INN 233, Station 14, EPFL, CH-1015 Lausanne, Switzerland
+41 78 776 2131 | george.prekas@epfl.ch | https://people.epfl.ch/george.prekas

INTERESTS

Improving datacenter performance, application latency, and energy efficiency. My recent contributions have focused on optimizing the behavior of microsecond-scale tasks through enhancements of the underlying operating system and virtualization infrastructure. In general, I enjoy building and understanding complex computer systems.

EDUCATION

École Polytechnique Fédérale de Lausanne (EPFL) Lausanne, Switzerland
Ph.D., Computer Science 2013-2018
Thesis: "Bridging the gap between dataplanes and commodity operating systems"
Advisor: Prof. Edouard Bugnion

National Technical University of Athens Athens, Greece
M.Sc. in Electrical and Computer Engineering 2004-2010
GPA: 9.20/10
Thesis: "Advanced web application for the manipulation of biosciences data"
Advisor: Prof. Timos Sellis

EXPERIENCE

École Polytechnique Fédérale de Lausanne (EPFL) Lausanne, Switzerland
Graduate research assistant *2013-Present*
Work on virtualization, operating systems, and networking. Contributions to two open source projects: a) the kernel module Dune, which is a special purpose virtualization hypervisor, and b) the dataplane operating system IX [2, 5]. Design and implementation of the energy management subsystem of IX [2, 4] and of a new dataplane operating system named ZygOS [1], which features a work-conserving scheduler for microsecond-scale networked tasks.

VMware Palo Alto, CA
Research intern *Summer of 2016*
Design and implementation of a series of tools to assist C/C++ developers in programming applications that use non-volatile memory for storing data structures.

Microsoft Research Redmond, WA
Research intern *Summer of 2015*
Contributions to Apache Hadoop Distributed File System that enable using Apache HDFS as a secondary tenant of nodes, without negatively impacting the service level objectives of the primary tenants [3]. At the same time, the placement of replicas is optimized in order to provide ideal data availability and performance.

European Dynamics SA Athens, Greece
Software engineer *2012-2013*
Design, deployment, maintenance, and administration of several production-grade high-availability clusters of database systems (Oracle, PostgreSQL, and MySQL).

Private education institutions Athens, Greece
Private tutor *2008-2009, 2012-2013*
Lectures and exercise sessions for students of the course PLI24 Software Design (UML, Java, and compilers) of the Hellenic Open University.

National Technical University of Athens Athens, Greece
E-business consultant *2005-2007*
Consultation of small and medium enterprises in order to acquaint them with the use of new technologies and e-business practices for the GoOnline EU program.

CURRENT PROJECTS

ZygOS [1] (https://github.com/ix-project/zygos) is a system optimized for microsecond-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.

IX [2] (https://github.com/ix-project/ix) is a dataplane operating system that provides high I/O performance and high resource efficiency while maintaining the protection and isolation benefits of existing kernels. IX uses hardware virtualization to separate management and scheduling functions of the kernel (control plane) from network processing (dataplane).

AWARDS

Google European Doctoral Fellowship in Operating Systems	2015
OSDI ’14 Jay Lepreau Best Paper Award	2014
4th place in the National Contest for Innovative Software by Undergraduate Students	2009
Advanced to Online Round 1 (top 250) at Google CodeJam 2004 (as prekageo)	2004

PUBLICATIONS

[1] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks. In Proceedings of the 26th ACM Symposium on Operating Systems Principles. SOSP, 2017.

[2] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The IX operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. ACM Transactions on Computer Systems (TOCS), 34(4):11, 2016.

[3] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation. OSDI, 2016.

[4] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In Proceedings of the Sixth ACM Symposium on Cloud Computing, pages 342–355. SOCC, 2015.

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation (OSDI). OSDI, 2014.

[6] J Chrin, L Hubert, R Krempaska, G Prekas, and T Pelaia. Xml constructs for developing dynamic applications or towards a universal representation of particle accelerators in xml, 2011.

[7] J Chrin and G Prekas. A taste of cafe. ICALEPCS, 2009.

[8] Panagiotis Alexiou, Thanasis Vergoulis, Martin Gleditzsch, George Prekas, Theodore Dalamagas, Molly Megraw, Ivo Grosse, Timos Sellis, and Artemis G Hatzigeorgiou. mirgen 2.0: a database of microrna genomic information and regulation. Nucleic acids research, 38(suppl 1):D137–D141, 2009.

[9] Kleanthis Prekas, Maria Rangoussi, Savvas Vassiliadis, and George Prekas. Performance of laboratory experiments over the internet: Towards an intelligent tutoring system on automatic control. Int. J. Inf. Commun. Eng, 1(4):208–211, 2005.

PATENTS

[1] Ricardo Bianchini, Inigo Goiri Presa, Marcus Felipe Fontoura, Georgios Prekas. Harvesting spare storage in a data center. US Patent 20,170,374,144, 2016.