TAMING LATENCY IN DATA CENTER APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

by

Mohan Kumar Kumar

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2019

TAMING LATENCY IN DATA CENTER APPLICATIONS

Approved by:

Professor Taesoo Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Professor Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Professor Tushar Krishna
School of Electrical Engineering
Georgia Institute of Technology

Dr. Keon Jang
Faculty
Max Planck Institute for Software Systems

Date Approved: April 1, 2019

To my wife, son, amma, appa, and sister.

ACKNOWLEDGEMENTS

First and foremost I would like to express my sincere gratitude and appreciation to my advisor Prof. Taesoo Kim without whom this dissertation would not have been possible. His invaluable ideas and comments helped me shape my projects during the course of my degree. As a Ph.D. student, he helped me understand the importance of articulation and presenting ideas in a paper. I would like to also thank Prof. Tushar Krishna and Prof. Abhishek Bhattacharjee for their support and guidance during the projects on TLB. The discussions with them were thought provoking and helped us shape the projects. Additionally, I would like to thank the other members of my dissertation committee, Prof. Ada Gavrilovska, Prof. Kishore Ramachandran, and Dr. Keon Jang for serving on my committee and for their suggestions in improving this dissertation work.

My time at Georgia Tech was made enjoyable in large parts due to my labmates, friends and mentors I gained here. A special thanks to Steffen Maass who is a dear friend and a collaborator in most of my projects. Furthermore, I would like to thank my fellow Ph.D. students in systems and SSlab. A special mention to Ming-Wei Shih, Sanidhya Kashyap, Insu Yun, Meng Xu, Pradeep Fernando, Sudarsun Kannan, Ketan Bhardwaj, and Minsung Jang, and former post-docs Woonhak and Changwoo. A special mention goes to Prof. H. Venkateswaran for all his help during my time in Georgia tech. I am also grateful to the support staff, Elizabeth Ndongi, Trinh Doan, Rojauna McPherson, and Susie McClain for their help in all the administrative work.

I would like to especially thank Prof. Karsten Schwan and Prof. Ada Gavrilovska for allowing me to be a part of their lab when I started. I am not sure if would have started my Ph.D. without their support. And, thanks to Prof. Ada Gavrilovska for all the discussions during my time in Georgia tech.

Finally, I would like to thank my family: my wife and son for their support and my parents for their encouragement over the years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

A new breed of low-latency I/O devices, such as the emerging remote memory access and the high-speed Ethernet NICs, are becoming ubiquitous in current data centers. For example, big data center operators such as Amazon, Facebook, Google, and Microsoft are already migrating their networks to 100G. However, the overhead incurred by the system software, such as protocol stack and synchronous operations, is dominant with these faster I/O devices. This dissertation approaches the above problem by redesigning a protocol stack to provide an interface for the latency-sensitive operation, and redesigning synchronous operation such as TLB shootdown and consensus in the operating systems and distributed systems respectively.

First, the dissertation presents an extensible protocol stack, XPS to address the software overhead incurred in protocol stacks such as TCP and UDP. XPS provides the abstractions to allow an application-defined, latency-sensitive operation to run immediately after the protocol processing (called the *fast path*) in various protocol stacks: in a commodity OS protocol stack (*e.g.*, Linux), a user space protocol stack (*e.g.*, mTCP), as well as recent smart NICs. For all other operations, XPS retains the popular, well-understood socket interface. XPS' approach is practical: rather than proposing a new OS or removing the socket interface completely, our goal is to provide stack extensions for latency-sensitive operations and use the existing socket layer for all other operations.

Second, the dissertation provides a lazy, asynchronous mechanism to address the system software overhead incurred due to a synchronous operationTLB shootdown. The key idea of the lazy shootdown mechanism, called LATR, is to use lazy memory reclamation and lazy page table unmap to perform an asynchronous TLB shootdown. By handling TLB shootdowns in a lazy fashion, LATR can eliminate the performance overheads associated with IPI mechanisms as well as the waiting time for acknowledgments from remote cores. By proposing an asynchronous mechanism, LATR provides an eventually consistent solution

to TLB shootdowns.

Finally, the dissertation untangles the logically coupled consensus mechanism from the application which alleviates the overhead incurred by consensus algorithms such as Multi-Paxos/Viewstamp Replication(VR). By physical isolation, DYAD eliminates the consensus component from competing for system resources with the application which improves the application performance. To provide physical isolation, DYAD defines the abstraction needed from the SmartNIC and the operations performed on the application running on the host processor. With the resulting consensus mechanism, the host processor handles only the client requests on the host processor in the normal case and the disappropriate messages needed for consensus is handled on the SmartNIC.

CHAPTER 1

INTRODUCTION

Data center applications, such as search, social networking, and e-commerce platforms, are commonly developed as hundreds of micro-services deployed over thousands of servers [1]. For instance, a single-user search query turns into thousands of remote procedure calls (RPCs) [2, 3, 4]. To accommodate the large traffic generated by these applications, 100/200 Gbps network interface cards (NICs) are becoming part of the next-generation data centers. In addition to traditional TCP/IP processing, NIC cards with remote direct memory access (RDMA) capability are finding their way into data centers. With such NIC cards, however, latency incurred by the existing system services, due to virtual memory and protocol stacks, is dominant in data centers. A recent study by Gao *et al*. [5] shows the latency induced by systems software in data centers to be 66% of the inter-rack latency and 81% of the intra-rack latency. To overcome the system service overheads, SmartNICs that contain processing elements to reduce the host processing overhead are becoming popular in data centers. However, an end-host system services does not have any abstractions to utilize these SmartNICs.

To address these challenges, this dissertation presents three low-level mechanisms to reduce latency induced by systems services on data center applications, and in addition provides abstractions to leverage the emerging SmartNICs.

We first focus on the software overheads induced by both the kernel and user-space protocol stacks, which is reduced by an extensible protocol stack, XPS, that allow a service-defined, latency-sensitive operation to run immediately after the protocol processing. XPS demonstrates the applicability of such an approach in various protocol stacks: in a commodity OS protocol stack (*e.g.*, Linux), a user space protocol stack (*e.g.*, mTCP), as well as recent smart NICs [6, 7]. In addition, XPS retains the existing socket layer for the rest of the

operations that are not latency sensitive. XPS' approach is practical: rather than proposing a new OS [8, 9, 10] or removing the socket interface completely [11], our goal is to provide stack extensions for latency-sensitive operations and use the existing socket layer for all other operations.

The dissertation then focuses on synchronous operations: TLB shootdown in operating systems and consensus in distributed systems. System services, such as virtual memory operations, page swap, and NUMA page migration, suffer from a synchronous TLB shootdown operation which impacts the throughput and latency of system services. LATR presents an asynchronous mechanism to maintain TLB coherence which eliminate the performance overheads associated with IPI mechanisms as well as the waiting time for acknowledgments from remote cores. The proposed lazy migration approach can play a critical part in emerging systems using heterogeneous memory where pages are migrated to faster on-chip memory [12, 13, 14] and in emerging disaggregated memory systems in data centers where pages are swapped to remote memory using RDMA [15, 5]. LATR shows the impact of reducing the TLB shootdown overhead on the tail latency of system services such as key-value stores.

Consensus algorithms in distributed systems require expensive coordination similar to a TLB shootdown. However, a lazy approach which is similar to eventual consistency is not applicable to consensus algorithms. Finally, we introduce DYAD, a system that untangles tightly coupled consensus mechanism from the system services by physically isolating them, allowing the high-overhead consensus component to run on the SmartNIC and the system service to run on the host processor. By physical isolation, DYAD eliminates the consensus component from competing for system resources with the services. To provide physical isolation, DYAD defines the abstraction needed from the SmartNIC and the operations performed on the system service running on the host processor. With the resulting consensus mechanism, the host processor handles only the client requests on the host processor in the normal case and the disappropriate messages needed for consensus is handled on the

SmartNIC.

1.1 Statement of Problem

Protocol stack:. Most services running in data centers rely on TCP/IP through the socket interface provided by operating systems (OS) such as Linux and FreeBSD. Using the socket interface is known to incur two performance overheads: First, overheads associated with the socket interface itself , such as `epoll()` and socket `read()/write()`, and second, overheads induced by cache misses when accessing cold network data followed by `epoll` events. To make the matter worse, recent kernel features, such as kernel page table isolation (KPTI), exacerbate the socket interface overheads. To address this problem, XPS presents the abstractions to execute the latency-sensitive operations of a service inside the protocol stack.

L7 abstractions and stack specific infrastructure:. Packet processing frameworks such as eBPF and XDP, while supporting an inline packet processing model, lack L7 abstractions. Such frameworks provide L2 packet processing abstractions that does not support multiple services. In addition to leveraging the OS and user-space protocol stack, the L7 abstractions are needed to leverage the emerging smart NICs, that provide low latency for packet processing, which is not available in existing systems. XPS' demonstrates the flexibility of its L7 abstractions by running them on three types of protocol stacks: in-kernel and user space protocol stacks, and on smart NICs.

Synchronous TLB shootdown:. The synchronous TLB shootdown mechanism available in existing operating systems show three types of performance overheads that increases the system service latency: sending IPIs to remote cores, which has an increased overhead on large NUMA machines; handling interrupts on remote cores, which might be delayed due to temporarily disabled interrupts; and the wait time for acks on the initiating core. The TLB shootdown mechanism is performed during a `munmap()` system call, a NUMA page migration, and a page swap, which impose high overhead on these operations. To alleviate the impact of TLB shootdown, LATR provides an asynchronous scheme for TLB

shootdowns.

Logically coupled consensus. : A typical consensus algorithm is developed as a library which provides an upcall to the service after reaching an agreement with a majority of the replicas [16, 17, 18]. Such a library provides a nice logical abstraction which isolates the consensus algorithm from the system service. However, with a logical isolation, the performance overheads of an consensus algorithm are entangled with the service, i.e., the consensus component competes with the service for system resources such as CPU, processor cache, etc. To overcome this challenge, DYAD untangles the tightly coupled consensus mechanism from the system services by physically isolating them, allowing the high-overhead consensus component to run on the SmartNIC and the services to run on the host processor. By physical isolation, DYAD eliminates the consensus component from competing for system resources with the services.

1.2 Thesis statement

The existing system services used by current data center applications suffer from inefficient implementation of lower-level mechanisms. By optimizing the lower-level mechanisms, the latency of these system services can be significantly improved.

To support the hypothesis, this dissertation makes the contributions outlined below.

1.3 Contributions

- We first show the protocol stack overhead in existing kernel and user-space protocol stacks. In addition, we designed XPS that allows an latency-sensitive operation to run immediately after the protocol processing (called the *fast path*) in various protocol stacks: in a commodity OS protocol stack (*e.g.*, Linux), a user-space protocol stack (*e.g.*, mTCP). In addition, XPS provides L7 abstractions to execute the latency sensitive operations in the SmartNIC.
- XPS provides a practical solution to reduce the socket overhead: rather than proposing

a new OS or removing the socket interface completely, XPS provides protocol stack extensions for latency-sensitive operations and use the existing socket layer for all other operations. XPS improves the throughput and tail latency of the Redis key-value store by up to 98.1% and 73.3% respectively.

- LATR shows the overhead of a synchronous TLB shootdown overhead. In addition, we designed a asynchronous mechanism that reduces the kernel overhead of a synchronous TLB shootdown operation from the critical path of system services. LATR eliminates the performance overheads associated with IPI mechanisms as well as the waiting time for acknowledgments from remote cores. LATR improves the latency of Apache web server by up to 26.1%.
- DYAD shows the overhead of existing consensus mechanism on distributed services. In addition, we designed a mechanism that filters the client requests, and executes the consensus data operations on the SmartNIC reducing the overhead of consensus on the host processor. The service, running on the host, executes only the client request which reduces the end-to-end latency of a system service. DYAD improves the latency of time-stamp server by up to 90%.

1.4 Organization

The reminder of this dissertation is organized as follows. In Chapter 2 (§2), we provide a background on SmartNICs followed by the system overhead in protocol stacks and the existing research approaches that address the protocol stack overhead. We next discuss the synchronous TLB shootdown overhead followed by the existing research approaches that address the TLB shootdown overhead. Finally, we discuss the consensus algorithm overheads due to the consensus messages and the system overhead.

In Chapter 3 (§3), we detail the design of XPS, a practical approach to eliminate the socket overhead in existing protocol stacks. In addition, this chapter details the evaluation of XPS with services such as Redis, Memcached, and Nginx.

In Chapter 4 (§4), we detail the design of LATR that provides a lazy shootdown mechanism In addition, this chapter details the evaluation of LATR with services such as Apache, Memcached, and `Mosaic`.

In Chapter 5 (§5), we detail the design of DYAD that leverages the future SmartNICs to reduce the consensus overhead. In addition, this chapter details the evaluation of DYAD with services such as time-stamp server, key-value store, and Memcached.

In Chapter 6 (§6), we discuss the advantages of combining the approaches provided by XPS, LATR, and DYAD. In addition, we demonstrate the applicability of a combined approach with Memcached.

And finally in Chapter 7 (§7), we conclude this dissertation and present ideas for future research.

CHAPTER 2
RELATED WORK

In this chapter, we provide the background and related work for the latency incurred due to protocol stacks, TLB shootdown in operating systems, and consensus algorithms.

2.1 Protocol stack

In this section, we present the overheads incurred in the protocol stack and the research approaches that address these overheads. To address the protocol stack overheads, in chapter 3 (§3) we present XPS that provides an extensible protocol stack that eliminates the socket interface overheads, in protocol stacks, by providing a *fast path*.

2.1.1 Socket interface overheads

To demonstrate the importance of reducing the overheads associated with the socket interface, we quantified their overheads when processing packets in two common scenarios: handling `GET` requests with 32-byte keys in Redis and restricting HTTP `POST` methods with 256 bytes in Nginx. Both services simply wait for a packet (or multiple packets) via `epoll()`, read it via `read/v()`, process it, and respond back via `write/v()`. Our evaluation shows that the socket interface cost is dominant in both services: 83.3% in Redis and 43.5% in Nginx (refer to Table 2.1 for a detailed breakdown). In addition, the socket interface cost increases with new kernel features such as KPTI [19]. Similar to Linux, with Redis running on Arrakis, up to 44% of the time is spent on Arrakis' POSIX APIs that does not provide a zero copy interface [11]. The costs involved in these common operations are eliminated by delegating the service logic to the fast path in XPS.

Table 2.1: Breakdown of the time spent when handling `GET` requests in Redis, and when blocking HTTP packets in Nginx. Up to 83.3% (Redis) and 43.5% (Nginx) of the total time is spent in the socket interface (machine configuration: Table 4.2).

Functions	**Redis** (a key-value store)		**Nginx** (a web server)	
	Time (%, μs)		**Time (%, μs)**	
Socket interface	83.3%	(11.64 μs)	43.5%	(13.32 μs)
`epoll_wait()`	23.3%	(3.25 μs)	8.4%	(2.58 μs)
`read/v()`	16.8%	(2.34 μs)	5.9%	(1.82 μs)
`write/v()`	43.3%	(6.05 μs)	29.2%	(8.92 μs)
Application logic	16.7%	(2.33 μs)	56.5%	(17.27 μs)
Total	100.0%	(13.97 μs)	100.0%	(30.59 μs)

2.1.2 Existing Approaches

Existing approaches for alleviating the socket interface overheads can be classified into three categories: 1) kernel-*batching* mechanisms, which attempt to amortize kernel crossings by batching; 2) kernel-*bypass* mechanisms, which eliminate the kernel crossing overhead by bypassing it; and 3) *extensions*, which attempt to place service logic inside the kernel.

Kernel batching mechanisms. Existing kernel protocol approaches [20, 21, 22, 23] amortize the system call overhead by using extensive batching. For example, IX [22] and ZygOS [23], the latest state-of-the-art kernel protocol stacks, provide an event-based system call interface with adaptive batching. However, although batching-based approaches amortize the system call overhead, their effect of increasing latency is a fundamental limitation.

Kernel bypass mechanisms. One might think that using kernel bypass mechanisms, such as DPDK [24], resolves the latency problem incurred as a result of system calls. But unless tightly integrated into the service logic, protocol stacks built in user space suffer from similar, often more serious, problems. For example, mTCP [25], a state-of-the-art protocol stack developed with DPDK, uses extensive socket-level batching. It efficiently amortizes the cost of control transfer between the TCP and service threads[1]but imposes

[1] mTCP has two threads per core: one performs protocol processing, and another processes service logic using the mTCP socket interface.

an up to 10× higher latency with `mtcp_epoll_wait()` as compared to `epoll_wait()` in the Linux kernel (see Table 3.1), which becomes predominant with increasing TCP connections (see Figure 3.4). Even worse, mTCP incurs the control transfer overhead twice: once for transferring data from the TCP thread to the service thread, and vice versa [25, 22]. Another problem of batching mechanisms is that the temporal cache locality of the network data is not retained during service processing, as services read the data after protocol processing [22, 26]. For example, with Redis, mTCP incurs 17% more L3 cache misses compared to Linux (see Table 3.6). Instead of using batching, we advocate a run-to-completion fast path that utilizes the service context, allowing the protocol stack to execute a latency-sensitive operation.

Extension approaches. Extending the protocol stack is not a new idea; Plexus [27] on SPIN [28] and ASHs [8, 9] on Exokernel [29] introduced kernel protocol extensions. However, ASHs and Plexus do not provide a mechanism to classify the latency-sensitive operations of a service. ASHs and Plexus apply their packet filters at the L2 layer, and provide these filtered packets to different protocol handlers. However, XPS introduces a concept of the *predicates* at the L4 layer and invokes corresponding service handlers. ASHs and Plexus are kernel-based approaches, whereas XPS demonstrates the applicability of immediate handler execution in the kernel, a user space protocol stack, and Table 2.2 shows a comparison of XPS with the existing approaches.

2.1.3 eBPF

eBPF [30] is an extension of the Berkeley Packet Filters (BPF) [31, 32], which introduce the concept of an in-kernel interpreter, allowing a safe translation of restricted bytecodes. To implement network tracing and L2/L3 functionality close to the NIC driver, eBPF provides extension points in the Traffic Control (TC) [33] and eXpress Data Path (XDP) [34] components, respectively.

Safety properties. eBPF provides kernel-safe code execution by first verifying inserted

Table 2.2: XPS compared with existing approaches.

	Plexus (Micro.)	**ASHs** (Exo.)	**Linux** (Mono.)	**IX** (Dataplane)	**mTCP** (Userspace)	**XPS**
Extension	✓	✓	-	-	-	✓
Predicate	L2	L2	N/A	N/A	N/A	L4 & L7
Fast path	✓	✓	-	-	-	✓
Slow path	-	-	✓	✓	✓	✓
Kernel	✓	✓	✓	✓	-	✓
Userspace	-	-	-	-	✓	✓
Smart NIC	-	-	-	-	-	✓

code, and then providing native performance guarantees via JIT compilation for x86 and ARM. The eBPF verifier performs the following checks before loading the eBPF program: the first test ensures that the eBPF program terminates and does not contain any infinite loops; next it ensures that out-of-range data accesses are not performed, and finally it restricts the kernel function access based on the eBPF program type.

Data structures. eBPF supports sharing a memory region between eBPF programs running in the kernel and user space. One such data structure are `maps`, pre-allocated binary blobs, which allow program-specific interpretation and are accessible from the user space via the `bpf()` system call. They are also accessible in parallel, protected by a kernel lock, from any eBPF module. The size of each map entry can be up to 32 MB, and the entire map size should not exceed the available memory. XPS' fast path handlers are eBPF functions that keep an isolated copy of the service data in eBPF maps.

Limitations. In the Linux kernel, eBPF extension points are located at the L2 layer, which cannot provide a fast path, as the transport layer (L4) is not available at the L2 layer. In addition, eBPF does not provide a mechanism to account the execution of a service handler to the user space service, resulting in unfairness among user space processes. XPS addresses the above limitations of eBPF, and abstracts the fast path (L7) operations such that they are executed in kernel, user-space stacks, and smart NICs.

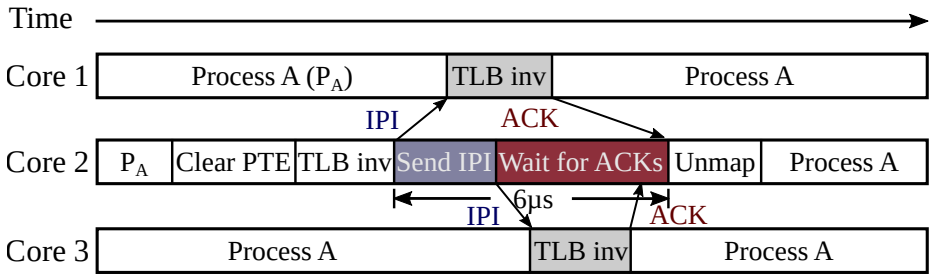


Figure 2.1: Page munmap design in Linux.

2.2 TLB shootdown

In this section, we present the overheads incurred by a synchronous TLB shootdown and the research approaches (both hardware and software) to address this problem. To address the delay incurred by synchronous shootdowns, in chapter 4 (§4) we will present LATR that provides lazy mechanism for TLB shootdowns during free and migration operations. With its lazy mechanism, LATR eliminates three types of performance overheads associated with the current TLB shootdown, namely, sending IPIs and waiting for the ACKs in the initiating core, and handling interrupts in the remote cores.

2.2.1 Existing OS Designs

Most architectures, including x86, do not support TLB cache coherence. The current x86 architecture allows two operations on TLBs: invalidating a TLB entry using the `INVLPG` instruction and flushing all local TLB entries by writing to the `CR3` register. However, both instructions provide control only over the local, per-core TLB. To invalidate entries in remote TLBs on other cores, a process known as *TLB shootdown*, commodity OSes use an expensive, IPI-based mechanism. IPIs are individually delivered to each remote cores via the Advanced Programmable Interrupt Controller (APIC) [35] as it does not support flexible multicast delivery [13].

Free operations. We analyze the existing handling of a free operation (`munmap()`) in Linux, on a system with three cores (as shown in Figure 2.1). The OS receives an `munmap()` system call from the service to remove a set of virtual addresses on core `C2` with the current process

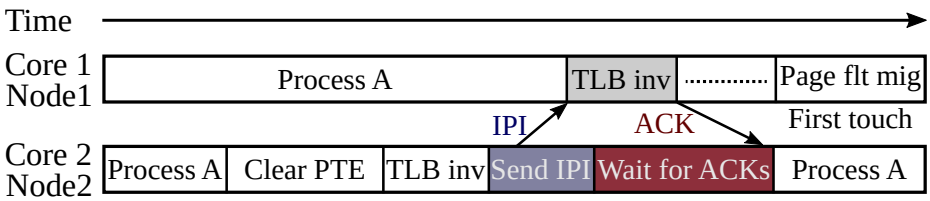


Figure 2.2: AutoNUMA page migration in Linux.

running on all existing cores (`C1`, `C2`, and `C3`). The `munmap()` handler removes the page table mappings for the set of virtual addresses and frees the virtual addresses and physical pages associated with the virtual addresses. In addition, `C1` performs a local TLB invalidation for the set of virtual addresses before initiating an IPI (to `C1` and `C3`) to perform the TLB shootdown. On receipt of the interrupt, `C1` and `C3` perform a local TLB invalidation in their IPI handlers and send an `ACK` to `C2` by the means of cache coherence. After receiving both `ACK`s from `C1` and `C3`, the `munmap()` handler on `C2` finishes processing the `munmap()` system call and returns control back to the service. The same TLB shootdown mechanism is used for all virtual address operations, though the page table changes are different.

The TLB shootdown mechanism outlined above shows three types of performance overheads: sending IPIs to remote cores, which has an increased overhead on large NUMA machines; handling interrupts on remote cores, which might be delayed due to temporarily disabled interrupts; and the wait time for `ACK`s on the initiating core.

AutoNUMA page migration. AutoNUMA page migration is a feature provided by Linux to migrate pages to the NUMA node where they are being frequently accessed from, to avoid costly cross-NUMA-domain accesses.

To identify pages that are predominantly used on a remote NUMA node, the AutoNUMA background task periodically scans a process' address space and changes page table entries which triggers frequent TLB shootdowns. After the TLB shootdown for a specific virtual address, any subsequent memory access to this virtual address triggers a page fault. If, during this page fault, a page is accessed twice from a NUMA node different from the current node the page resides on, the page will be migrated to the node accessing the page, pending other factors such as enough free memory on the target node. Figure 2.2 gives a

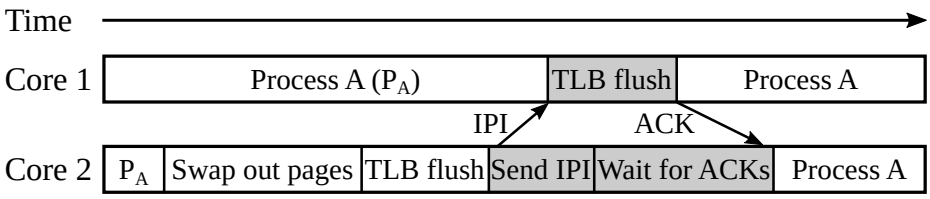


Figure 2.3: Page swapping in Linux.

high-level overview of this process in Linux (shown with a single page fault for simplicity). As explained above, if the page is accessed from the same NUMA node, the page migration is not performed even though the expensive TLB shootdown (5.8%, with one 4 KB page, to 21.1%, with 512 4 KB pages, of the overall migration cost) was performed.

Page swapping. Page swapping is a feature in commodity OSes, such as Linux and FreeBSD, to swap least recently-used (LRU) pages to disk during high memory pressure. In Linux, a kernel background task (`kswapd`) maintains an active and inactive list of pages. To begin with, the kernel adds the allocated pages to the inactive list. By tracking the PTE access bits, `kswapd` takes an informed decision to move pages from the inactive to the active list, and vice versa. During high memory pressure, pages are swapped out to disk from the inactive list, which triggers a synchronous TLB flush as shown in Figure 2.3.

With the advent of disaggregated data centers, the paradigm for page swapping shifts from disks to remote memory using fast network interconnects. Instead of swapping pages to disk, recent research systems, such as INFINISWAP [15], advocate for the usage of remote memory using Infiniband RDMA, which reduces the tail latency of page swapping by up to 61×. Due to the reduced remote paging latency, the TLB shootdowns needed for swapping become an important contributor to the cost of page swapping (contributing up to 18% for a Memcached workload using INFINISWAP).

2.2.2 Hardware-based Approaches

Hardware-based research approaches provide cache coherence to the TLB. UNITD [36], a scalable hardware-based TLB coherence protocol, uses a bit on each TLB entry to store sharing information, thereby eliminating the use of IPIs. However, UNITD still

Table 2.3: Comparison between LATR and other approaches to TLB shootdowns.

Properties	DiDi [38]	Oskin *et al.* [13]	ARM TLBI [46, 47, 48]	UNITD [36]	HATRIC [37]	ABIS [49]	Barrelfish [50]	Linux [51]	LATR
Asynchronous approach	-	-	-	-	-	-	-	-	✓
Non-IPI-based approach	✓	-	✓	✓	✓	-	✓	-	✓
No remote core involvement	✓	✓	✓	✓	✓	-	-	-	✓
No hardware changes required	-	-	-	-	-	✓	✓	✓	✓

resorts to broadcasts for invalidating shared mappings. Furthermore, UNITD adds a costly content-addressable memory (CAM) to each TLB to perform reverse address translations when checking whether a page translation is present in a specific TLB, thereby greatly increasing the TLB's power consumption. HATRIC [37] is a hardware mechanism similar to UNITD and piggybacks translation coherence information using the existing cache coherence protocols.

DiDi [38] employs a shared second-level TLB directory to track which core caches which PTE. This allows efficient TLB shootdowns, while DiDi also includes a dedicated per-core mechanism that provides support for invalidating TLB entries on remote cores without interrupting the instruction stream they execute, thereby eliminating costly IPIs. Similarly, other approaches provide microcode optimizations to handle IPIs without remote core interventions [13]. Though these approaches remove interrupts on remote cores, the wait time on the core initiating the TLB shootdown is not removed. Finally, these approaches require intrusive changes to the micro-architecture, which adds additional verification cost to ensure correctness [39, 40, 41, 42, 43, 44, 45].

2.2.3 Software-based Approaches

Commodity OSes, such as Linux and FreeBSD, implemented a set of non-trivial software-base optimizations. For example, Linux made two important optimizations for TLB shootdowns: 1) batched remote TLB invalidation [52], where multiple invalidation entries are batched together in one IPI, and 2) lazy TLB invalidation that balances between the overheads of TLB flushes and TLB misses when a core becomes idle. It is worth noting that Linux's lazy invalidation mechanism refers to lazily invalidating entries on the local TLB

in *idle cores*, which is different from LATR's lazy mechanism that lazily invalidates entries on *remote cores*. More specifically, it works as follows: when a core becomes idle, the OS speculates that the same process may get scheduled on the same core, so it defers the invalidation of the local TLB to avoid potential future TLB misses. However, if the idle core subsequently receives a TLB shootdown, the OS performs a full TLB invalidation and indicates to the other cores not to send further shootdown IPIs to this core while it remains idle. Unfortunately, all these optimizations do not eliminate the IPIs needed for TLB invalidation.

Barrelfish [50], a research multi-kernel OS, uses message passing instead of IPIs to shoot down remote TLB entries. Thus, it eliminates the interrupt handling on remote cores. However, it still has to wait for the `ACK` from all remote cores participating in the shootdown. We note that Barrelfish thereby still takes a synchronous approach for TLB shootdowns. ABIS [49], a recent state-of-the-art research prototype based on Linux, uses page table access bits to reduce the number of IPIs sent to remote cores by tracking the set of CPUs sharing a page, which can be complementary to LATR. However, the operations in ABIS to track page sharing introduce additional overheads.

Similarly, there are a number of approaches in the OS [53, 54, 55, 50, 49, 56, 57, 58, 59] to optimize TLB shootdowns. However, none of them eliminate the synchronous TLB shootdown overhead. An alternative approach for reducing TLB shootdown is that services can inform the OS on how memory is used or handle TLB flushes explicitly. Corey OS [60] avoids TLB shootdowns of private PTEs by requiring the user services to explicitly define shared and private pages. Apart from this, FreeBSD uses versions with process context identifiers (PCIDs) [61] to eliminate the IPI operation. However, this approach invalidates all the TLB entries by using a version-based mechanism, which induces TLB misses.

Other TLB-related optimizations. SLL TLBs introduced a shared last-level TLB [62, 63] and evaluated the benefit of using a shared last-level TLB compared to a private second-level TLB. However, their design still relies on IPI-based coherency transactions. In addition,

research approaches showed that TLB misses are predictable and that inter-core TLB cooperation and prefetching mechanisms can be applied to improve TLB performance [64, 65]. However, this implies that a TLB shootdown must also invalidate mappings in the TLB prefetch buffers. In addition, other approaches improve TLB misses, which is an orthogonal problem [66, 67, 68, 69].

We conclude that these hardware- and software-based approaches for TLB shootdowns do not eliminate all TLB shootdown overheads and are not easy to apply in current systems due to their required hardware changes. Table 2.3 provides a comprehensive comparison of existing approaches with LATR.

2.3 SmartNIC consensus

In this section, we present the overheads (direct and indirect) incurred by a consensus algorithm such as VR and the approaches that address these overheads. To address the direct and indirect cost of consensus, in chapter 5 (§5) we will present DYAD that delegates the consensus algorithm to the SmartNICs that handles the coordination with replicas closer to the network I/O, eliminating the PCIe and CPU overheads.

2.3.1 Consensus algorithms

Data center services provide high availability and consistency by replicating their data using consensus algorithms. For example, lock services such as Chubby [16] and Zookeeper [17], persistent storage systems such as H-Store [70], Granola [71], Megastore [72], and Spanner [73], use consensus algorithms for replicating their data. These systems provide high availability by using multiple replicas, and they provide better performance by maintaining in-memory state.

Consensus protocols, such as Paxos [74, 75], Viewstamped Replication (VR) [76, 77], atomic broadcast [78], or Raft [18], ensure that operations execute in a consistent order across replicas. In DYAD, we consider systems that provide leader based state machine

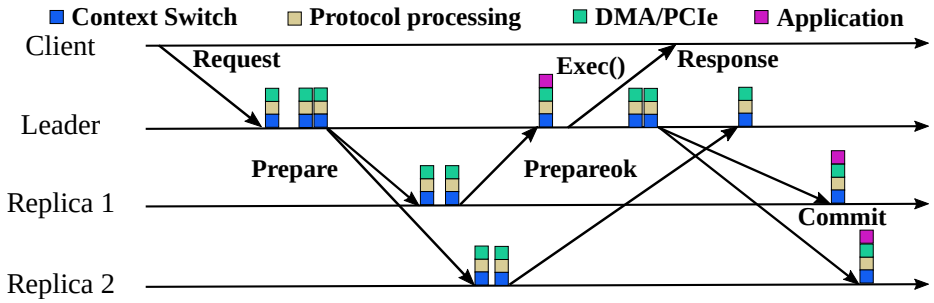


Figure 2.4: Normal-case execution of VR. The existing VR algorithm incurs high direct and indirect cost which increases with increasing replicas.

replication where a set of nodes are either clients or replicas. The replicas run the service code and communicate with the other replicas using the consensus algorithms. Note that, we explicitly use the term leader to refer to the leader replica and the replicas to refer to the replicas other than the leader. Clients submit a request, containing an operation, to the leader that begins a multi-round protocol with the replicas to agree on a consistent order of operations before executing the request.

We will look at the normal case operation of leader-based multi-paxos which is equivalent to VR algorithm. Figure 2.4 shows the normal case operation of multi-paxos when there are no failures. The leader node is responsible for ordering requests which is replaced by a new leader upon failure. Clients send their requests to the leader which assigns a sequence number to each request, and sends a PREPARE message to the other replicas containing the request and the corresponding sequence number. The other replicas record the request in the log and acknowledge with a PREPAREOK message to the leader. When the leader receives the PREPAREOK message from the majority of the replicas, it executes the operation and sends a reply to the client. In addition, the leader sends a COMMIT message to all the replicas to commit the sequence number.

2.3.2 Consensus overheads

Data center services demand high throughput and low latency from replication services. Low latency is an important factor for modern online services that access data from thousands of servers. For consensus algorithms, throughput and latency is limited by the high CPU

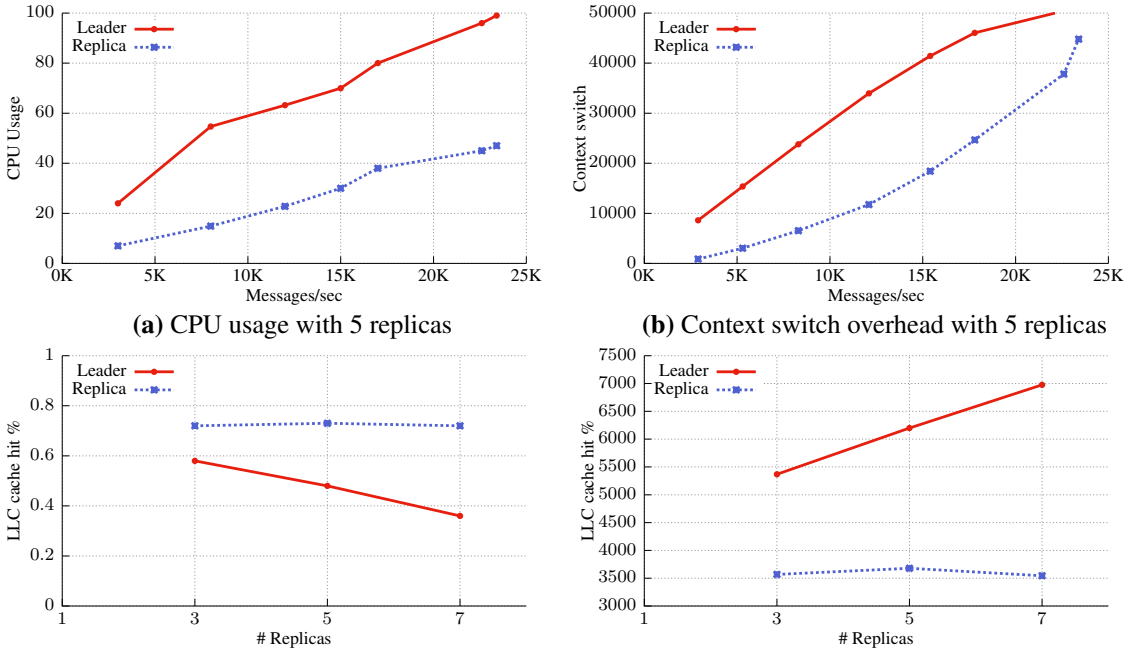
overhead on the leader node to process disproportional number of messages. For example, to execute a single client request with three replicas, the leader handles eight messages: receive one client request, send two PREPARE messages, receive two PREPAREOK messages, send one client response, and send two commit messages. And, the number of messages handled on the leader increases with the number of replicas. Handling disproportional number of messages incur typical system overheads, such as context switch, protocol processing, and PCIe [79, 80], which increases the CPU utilization on the leader. We classify the cost of consensus latency into direct and indirect cost.

$$Latency_{direct} = \frac{n-1}{2} * RTT + \overbrace{(n-1) * TX + \frac{n-1}{2} * RX}^{\text{Leader prepare}} + \overbrace{\frac{n-1}{2} * (RX + TX + t_{process})}^{\text{Replicas prepare}} + Latency_{other} \quad (2.1)$$

$$Latency_{other} = \overbrace{\frac{n-1}{2} * RX}^{\text{Leader prepare}} + \overbrace{(n-1) * TX}^{\text{Leader commit}} + \overbrace{(n-1) * RX + t_{process}}^{\text{Replicas commit}} \quad (2.2)$$

Direct cost. We define the direct cost as the sum of the round-trip times (RTT) needed to reach consensus, the system overhead on the leader in sending PREPARE message and processing PREPAREOK messages, and the system latency on replicas for processing a PREPARE message (as shown in Equation 2.1). In addition, the direct cost comprises of the other costs that includes the cost of processing the COMMIT messages and the PREPAREOK from the remaining replicas after reaching consensus (as shown in Equation 2.1). In the direct cost, the data center network is optimized for microsecond RTTs whereas the system overhead in processing the consensus messages play a larger role, which increases with increasing replicas.

Indirect cost. The indirect cost is defined as the cost incurred due to sharing the hardware



(a) CPU usage with 5 replicas

(b) Context switch overhead with 5 replicas

(c) LLC cache hits with increasing replicas

(d) Memory accesses with increasing replicas

Figure 2.5: The system overhead of consensus algorithms with a timestamp server.

resources, such as cache pollution, context switches, and CPU overhead, which is caused by handling consensus messages on the host processors along with the service. The impact of the indirect cost on the end-to-end latency is difficult to measure, though the indirect cost also increases with increasing replicas. Approaches that reduce the direct cost by using kernel-bypass mechanisms, such as DPDK and RDMA, does not eliminate the indirect cost. For example, these approaches rely on busy polling which wastes CPU cycles and increases the CPU overhead. In conclusion, the system overheads play a vital role in consensus latency due to both the direct and indirect cost.

2.3.3 Network-assisted approaches

Research approaches propose to move the ordering guarantees of consensus protocol to the data center network [81, 82, 83, 84, 85]. However, such approaches impose certain (ordering) requirements on the network to eliminate the consensus overheads, and their resulting throughput drops as the out-of-order messages increase. NetPaxos, a prototype implementation of Paxos at the network level, consists of a set of OpenFlow extensions implementing Paxos on SDN switches. Similarly, another research extends P4 to implement

Paxos on switches which is constrained by hardware resources [86, 87]. However, the Paxos implementations on the switches suffer from performance bottlenecks that limit their throughput.

2.3.4 RDMA approaches

Recent research systems focus on optimizing consensus using RDMA [88, 89, 90, 91]. Systems such as DARE [88] built state machine replication on top of a protocol similar to Raft and optimized for one-sided RDMA. Similarly, APUS [89] built an RDMA based paxos protocol that scales to multiple client connections. However, consensus algorithms built using RDMA is tightly coupled with the service, and does not reduce the CPU overhead for consensus.

2.3.5 Hardware approaches

To avoid the consensus overhead, research approaches implement the Zookeeper Atomic Broadcast (ZAB) consensus protocol and the services on FPGA devices using a low-level language [92]. This hardware-based solution, however, may not be scalable as it requires the storage of potentially large amounts of service data which is limited on the FPGA hardware. In addition, other than the normal case, the corner failure cases which are common is distributed systems is hard to implement in the FPGAs. Finally, such systems tightly couple both the consensus component and the service in the hardware restraining the service to run on special FPGA hardware with limited memory.

2.4 Hardware trends

In this section, we provide a primer on SmartNICs that are becoming part of the data center architectures.

2.4.1 SmartNIC

SmartNICs are gaining popularity because of the increasing network bandwidth (100 and 200 Gbps) available in Ethernet NICs and the software and PCIe overhead [79] for packet processing on the host processors [93, 94]. In addition, SmartNICs eliminate the PCIe overhead predominant in current servers [79]. Such SmartNICs are already deployed in Microsoft Azure data centers [93]. Three different processing elements are available in SmartNICs: ASICs, SoCs, and FPGAs [95, 93]. SmartNICs with SoCs or FPGAs are available commercially, with the existing SoC-based SmartNICs providing better programmability compared to FPGAs [93]. Commercial SmartNIC SoCs are available with ARM processors (8 GB memory)or with custom network processing units (NPUs) (up to 24 GB).

The Netronome Agilio, an NPU-based SmartNIC, provides P4 programming [96] capabilities that allow for flexible header parsing, and allows custom C program invocation from P4 [97]. Similar to Netronome, other FPGA based SmartNICs support P4 programmability. The Agilio NIC has a many core processor, which contains 72 processing elements called micro engine (ME) with 8 threads each.

CHAPTER 3

XPS: ADDRESSING LATENCY INCURRED BY PROTOCOL STACK

3.1 Introduction

Data center applications, such as search, social networking, and e-commerce platforms, are commonly developed as hundreds of micro-services deployed over thousands of servers [1]. Micro-services communicate using the protocol stacks, which should provide low latency and high throughput over high bandwidth links (10–40 Gbps) [3, 4, 2]. Lower latency, while serving a large number of requests, is a requirement for such data center applications. In reality, however, latency incurred by the existing protocol stack is dominant in data centers. A recent study by Gao *et al*. [5] shows the latency induced by network software in data centers to be 66% of the inter-rack latency and 81% of the intra-rack latency.

Most system services running in data centers rely on TCP/IP through the socket interface provided by operating systems (OS) such as Linux and FreeBSD. Using the socket interface is known to incur two performance overheads: first, overheads associated with the socket interface itself [25, 98, 20, 11] , such as `epoll()` and socket `read()/write()` (shown in Table 3.1), and second, overheads induced by cache misses when accessing cold network data followed by `epoll` events [99, 100, 101]. To make the matter worse, recent kernel features, such as kernel page table isolation (KPTI) [102], exacerbate the socket overheads by up to 48% (shown in Table 3.1). Apart from Linux, even an optimized socket interface provided by Arrakis incurs 44% overhead for a `GET` operation with Redis [11]. These overheads, as a result, negatively impact an service's performance by decreasing throughput and increasing tail latency.

To amortize the socket overheads, research approaches that use the kernel protocol stack propose a batched system call based on sockets [20, 21] or events [10, 22, 23]. For

Table 3.1: Breakdown of the socket interface costs in an echo server (64 B messages) with Linux and mTCP. Socket overhead is dominant in Linux, and increases by 48% with new features such as KPTI. Similarly, mTCP incurs a high control transfer overhead due to the mTCP socket APIs. (Note: AWS has a faster CPU, the machine configuration is in Table 4.2)

Functions	**Linux kernel**			**User space**
	AWS	Bare metal	w/ KPTI	mTCP
Socket APIs†	5.2 μs (99.6%)	8.9 μs (99.8%)	13.9 μs (99.9%)	23.5 μs (99.9%)
`epoll()`	1.8 μs (34.4%)	2.4 μs (26.4%)	4.9 μs (35.0%)	22.3 μs (94.6%)
`read()`	1.0 μs (18.2%)	1.8 μs (19.6%)	2.8 μs (19.8%)	0.4 μs (1.5%)
`write()`	2.5 μs (47.0%)	4.8 μs (53.8%)	6.3 μs (45.1%)	0.9 μs (3.8%)
Application	0.02 μs (0.4%)	0.02 μs (0.2%)	0.02 μs (0.1%)	0.02 μs (0.1%)

† : `mtcp_*` for mTCP's socket interface.

example, MegaPipe [20], FlexSC [21], and IX [22] process a group of system calls as a batch, amortizing the cost of context switching. Similar to the kernel batching approaches, mTCP [25], a user space stack developed with the data plane development kit (DPDK) [24], avoids system calls by providing mTCP socket APIs that use extensive batching to amortize the cost of control transfers (detailed in §2.1.2). However, such batching approaches sacrifice latency in pursuit of higher throughput which negatively impacts system service performance [26].

Alternatively, textbook extension systems such as Plexus [27] and *Application-specific Safe Handlers* (ASHs) [8, 9] allow specific handlers to run in a research OS, thus avoiding the socket overheads. The main goal of these approaches is to assemble various protocol layers to build a custom protocol stack that suits the system service, while also running the entire service within the kernel. However, these approaches require significant refactoring of services and are not supported in current commodity OSes. Apart from software protocol stacks, a more drastic approach, popularly used by modern systems, is to replace the socket interface with a native RDMA API. However, this requires the presence of specialized adapters at both ends of the connection, and often needs a complete redesign of services [103, 91, 104, 90, 105, 106].

Instead of running the entire service within the kernel, we observe that network services such as key-value stores, web servers, and distributed systems often have a latency-sensitive

operation that should be performed with less software stack overhead. For example, GET latency is critical in key-value stores, which is optimized by research approaches that process GETs in the network using programmable switches [107, 108]. Similarly, web servers restrict requests based on an HTTP method or any HTTP headers, which should be done with less software stack overhead [109, 110]. The above observation implies that low software stack overhead is a key requirement for the latency-sensitive operation of a service, whereas the rest of the operations, *e.g.*, SET operations in Memcached, can incur reasonable overhead.

We propose XPS, an extensible protocol stack, which provides the abstractions to allow an latency-sensitive operation to run immediately after the protocol processing (called the *fast path*) in various protocol stacks: in a commodity OS protocol stack (*e.g.*, Linux), a user space protocol stack (*e.g.*, mTCP), as well as recent smart NICs [6, 7]. In addition, XPS retains the existing socket layer for the rest of the operations (called the *slow path*). XPS' approach is practical: rather than proposing a new OS [8, 9, 10] or removing the socket interface completely [11], our goal is to provide stack extensions for latency-sensitive operations and use the existing socket layer for all other operations. Though extending the protocol stack is not a new idea, XPS provides a general and portable framework for embedding latency-sensitive operations within three different protocol stacks by separating the stack-specific infrastructure and API that is stack agnostic. To provide the fast path, XPS leverages eBPF that provides extension points only at the L2 layer in the Linux kernel (see §2.1.3). However, XPS is a generalization of eBPF, which abstracts the fast path (L7) operations such that they are executed in kernel, user-space stacks, and smart NICs.

To show the benefits of XPS, we ported three types of real-world services with XPS—caching in a key-value store, filtering and restricting HTTP requests in a web server [111], and handling heartbeats and consensus in a distributed system—each of which requires service changes of less than 200 LoC. We show that XPS improves the throughput and tail latency of the Redis [112] key-value store by up to 98.1% and 73.3% and the Nginx web server [113] by up to 2.2× and 82.0%, respectively. In addition, compared to IX [22] and

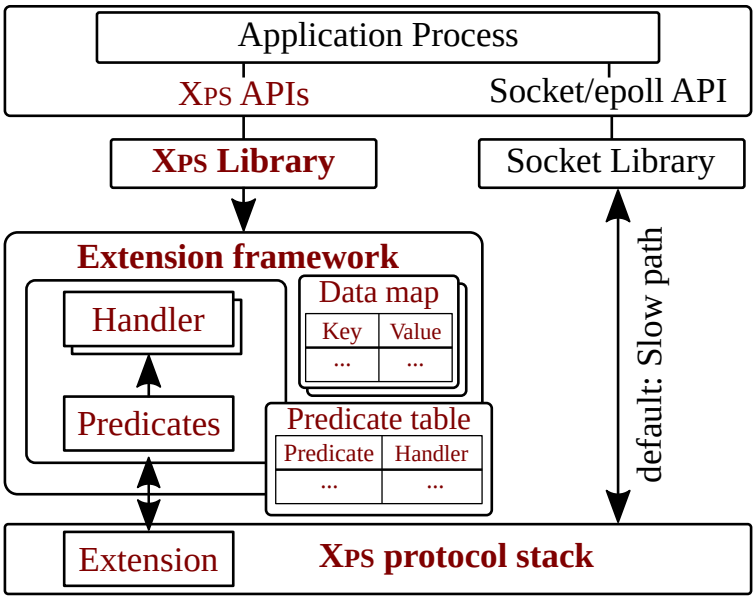


Figure 3.1: The three XPS components (in bold) co-existing with the existing interface.

ZygOS [23], XPS-mTCP improves Redis' throughput and tail latency by up to 50.1% and 63.5%, respectively.

With XPS, We make the following contributions:

- **Abstraction:** We devised the key properties and abstractions of XPS that is stack agnostic and allow the adoption of our approach with minimal code changes.
- **Demonstration:** We demonstrate the applicability of an extensible protocol stack in kernel, user-space stacks, and smart NICs.
- **System service:** We demonstrate XPS' benefits with three types of real-world services in various machine and network environments.

3.2 Design

Overview. XPS consists of three components: the XPS *library* that provides the abstractions, the *extension framework*, and the XPS *protocol stack*. The extension framework contains both the predicate table and the packet processing framework and allows inserting application handlers into the framework. The XPS protocol stack invokes the extension framework, and the corresponding application handler containing the service logic and a data map to store the associated data, before the payload is delivered to the service using the

socket interface. The predicate table, used by the extension framework, binds an application handler to a predicate. During packet processing, the appropriate application handler is invoked if the predicate is true. Predicates are based on the standard 5-tuple: (source IP, source port, destination IP, destination port, protocol). Figure 3.1 shows the three components that co-exist with the existing socket interface.

We first present XPS' abstractions before introducing its extension framework and protocol stack. Finally, we describe XPS' characteristics in three different execution environments: in-kernel, user space, and smart NICs.

3.2.1 XPS Abstractions

XPS provides four abstractions—control, data, composability, and statistics—that are stack agnostic and allow the fast path operations; Table 3.2 shows each API in detail.

Control abstractions. The control abstractions provide APIs to create and delete a handler context that contains the predicate and the handler. The `create_context` operation returns the context that is used for the data operations, while the `update_context` operation allows changing the handler for an already existing context. The `delete_context` operation deletes the predicate and the associated handler details from the predicate table. As the control operations are isolated from the data path, the control operations are eventually consistent (i.e., a change of a predicate and its handler details will take effect on the incoming packets eventually).

Data abstractions. The data abstractions are associated with the application handler context, which provides APIs to insert/update/delete data from the data map. The `data_put` and `data_update` operations take the handler context along with a key and a value to update the data map. The `data_delete` operation takes the context along with a key to delete an existing entry. Similar to the control operations, data operations are eventually consistent. In addition, XPS supports strong consistency that is discussed in §3.2.5.

L7 composability abstractions. The composability abstraction binds a handler to a

Table 3.2: APIs provided by XPS in each category of abstractions, namely, *Control*, *Data*, *Data statistics*, and *Composability*.

Abstraction	API	Description
Control	`context* create_context(char *file, int server_sock)`	Create context with the handler in an object file, for a socket
	`context* update_context(context *c, char *file)`	Update existing context with the new handler in an object file
	`void delete_context(context *c)`	Delete the already existing handler context
Data	`bool put_data(context *c, key *k, val *v)`	Create new data entry for an existing context with given data
	`bool update_data(context *c, key *k, val *v)`	Update existing data entry with the new value
	`bool delete_data(context *c, key *k)`	Delete the existing data entry
Statistics	`void* get_stats(context *c, key *k)`	Get the number of data access events for a given key
	`void* get_allstats(context *c)`	Get the number of data access events for all keys
Composability	`context* create_compcontext(int server_sock, int offset, int len)`	Create composability context with the given offset and length
	`context* create_subcontext(context *c, char *file, char *value)`	Create subcontext with the handler in an object file for a value

L7 predicate that matches a value retrieved from an offset and length in the L7 payload. For a single TCP/UDP flow, the L7 predicate invokes an application handler depending on the value retrieved from the offset and length. For example, IncBricks' [108] packet format contains the *command*, which selects the operation, with a length of four bytes at offset eight. With the composability abstraction, the value retrieved at offset eight of the IncBricks' L7 payload would be used to invoke the appropriate application handler. The offset and length of the L7 payload is provided in the `create_compcontext` operation, and the `create_subcontext` operation binds an application handler to the provided predicate value.

Data statistics abstractions. The data statistics abstractions provide APIs to return the statistics about the operations performed by the associated handler. Data statistics are needed for services to implement algorithms (e.g., eviction algorithms, event reporting, etc.) based on the data access pattern.

3.2.2 Extension framework

Predicate and handler binding. The extension framework acts as an interface between the XPS APIs and its protocol stack by using the predicate table that binds the predicates to the corresponding application handlers. Further updates to the predicate table (*i.e.*, the predicates and their associated handlers) are performed using this framework. The predicates are standard 5-tuples.

Packet processing. During packet processing, the XPS protocol stack invokes the extension handler before delivering events to the service. The extension handler performs a predicate table lookup based on the header retrieved from the packet metadata (*e.g.*, the `sk_buff` in the Linux kernel), providing the application handler to the protocol stack if an entry is found. Similarly, the composability handler is invoked with the packet metadata.

L7 Composability handler. The composability handler is invoked after a L4 predicate is matched. The composability handler retrieves a value from an offset and length in the L7 payload, and uses the retrieved value to invoke an application handler. The composability handler uses another map, called the composability map, that binds a handler to the predicate value. For example, with IncBricks' packet format, the retrieved *increment* command (with a length of four bytes at offset eight), in the L7 payload, is looked up in the composability map to invoke the increment handler.

3.2.3 The XPS Protocol Stack

XPS provides **extensibility** by providing an interface to insert the application handlers, and invokes the extension and application handlers during packet processing. During packet processing, the application handlers provided by the extension framework are executed before delivering the events to the service. The XPS protocol stack, based on the return code of an application handler (`TX`, `DROP`, `PASS`, or `RESET`), performs further actions such as sending a response from the application handler (`TX`), dropping the packet (`DROP`), passing the packet to the service (`PASS`), or terminating the client connection (`RESET`).

The fast-path processing enables three key design aspects: first, by avoiding the socket interface for sending responses and dropping packets; second, by immediately handling requests in the protocol stack, the temporal cache locality of the packet and its metadata is retained for the application handlers and third, the immediate handling provides a zero copy interface to the application handler for receiving and sending messages. In addition, the explicit fast-path handler is executed on smart NICs, avoiding the PCIe overhead.

TCP protocol processing. In the kernel and user space TCP stack, application handlers are executed after the protocol stack processing, before delivering (if needed) the events (`EPOLLIN`/`MTCP_EPOLLIN`) to the service. Each packet invokes the application handler if the predicates match, and the application handler processes the TCP stream. Though the handlers are invoked for each packet, the message boundaries are identified by the handlers, and the actions (*e.g.*, `TX`, `DROP`) are performed only at message boundaries. Similarly, the composability handler retrieves the value from an offset and length at message boundaries, which shows the advantage of tightly integrating the service logic with the protocol stack. The necessary TCP protocol processing, such as segmentation, congestion control, and handling packet loss, is performed before the handler execution. With XPS, the return codes `TX` or `DROP` immediately updates the TCP window size and sends an acknowledgment.

We present an example to understand how the extension framework fits into the XPS kernel TCP stack. Note that with a user space stack, such as mTCP, the logical flow remains the same though the stack runs in a dedicated thread within the service's address space. We look at the packet path in detail with an service—Nginx, for restricting HTTP `POST` methods (see §2.1). As shown in Figure 3.2, the packets are first DMAed to the core that processes the requests ❶. After TCP protocol processing, such as check sum and sequence number validation, the XPS stack invokes the extension handler before adding events (`EPOLLIN`) to the socket layer ❷. Since the extension framework predicate table associates the Nginx handler with destination port 80, the Nginx handler is invoked for packets destined to port 80 ❸. The Nginx handler builds a response depending on the HTTP method parsed in the packet, containing an HTTP status code, 405 or 444, for POST or PUT methods, respectively. For the POST and PUT methods, the Nginx handler returns `TX` to the XPS TCP stack ❹, signalling the transmission of the response to the NIC after finishing the protocol processing (*e.g.*, updating the window size and congestion control), without notifying the user space Nginx service ❺ (fast-path processing). When the method does not match POST or PUT, the Nginx handler returns `PASS` to the XPS TCP stack ❻ which provides the events

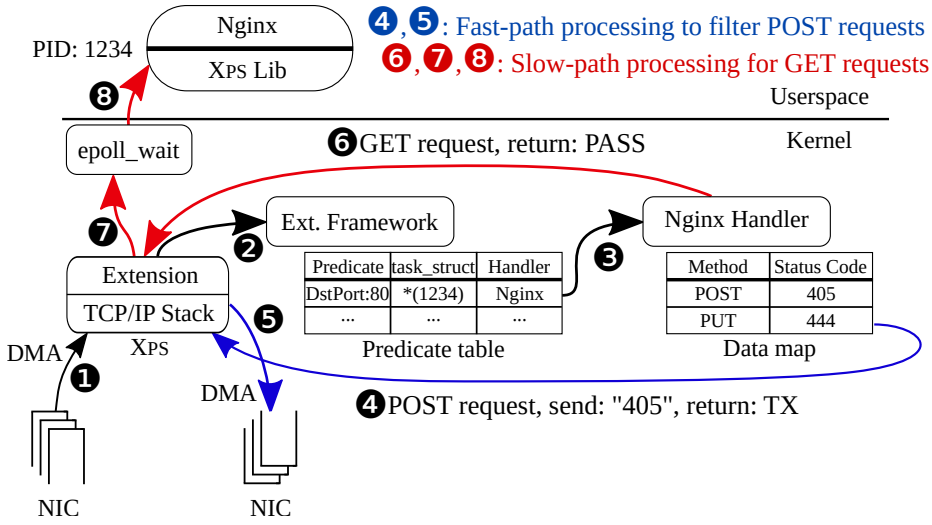


Figure 3.2: Example of the packet flow with the XPS kernel stack with Nginx. The extension framework predicate table associates the Nginx handler with destination port 80, while the Nginx handler data map associates the HTTP status code 405 and 444 with the POST and PUT methods, respectively. All other methods are passed to Nginx.

(`EPOLLIN`) to the Linux epoll layer ❼, in turn providing the read event to the Nginx service in user space ❽ (slow-path processing).

3.2.4 Stack Specific Infrastructure

The fast-path handlers specified with XPS' abstraction are flexible enough to run on three types of protocol stacks: in-kernel and user space protocol stacks, and on smart NICs. In this section we describe specific issues and characteristics of XPS' infrastructure. Table 3.3 shows details of XPS with various implementations.

In-kernel Protocol Stack

The extension framework and fast-path handlers are implemented as eBPF functions that are inserted into the kernel TCP stack. The data map, accessed by the fast-path handlers, is available in the kernel, and the service should keep an isolated copy of the data in the data map using the data abstractions. *Consistency* is provided by locking the data map entry during inserts/updates, which provides an eventual consistency model for services. For example, with services such as key-value stores, the new values updated with `SET` operation

is eventually consistent with the handler's data map, which is acceptable. In addition, to provide fairness and safe extensibility, the XPS kernel protocol stack provides two important properties: *handler accountability* and *safety*.

Handler accountability. With a kernel protocol stack, an unaccounted application handler could hurt the entire system by introducing unfairness across user processes. To address that, XPS ensures that the time taken for fast-path handler execution is accounted for in the appropriate user space service, which the stock eBPF fails to provide. To provide handler accountability, the XPS kernel stack accounts for the cycles spent in the extension framework and the application handler in a new field in the task structure. To avoid frequent accounting updates, the accumulated cycles in the new field are not accounted for in the user space process on a per-packet granularity. Instead, once the accumulated cycles amount to the cycles of one scheduler tick (1 ms in Linux and x86-64), the handler execution time is accounted for in the user space process. By accumulating the handler execution cycles, XPS avoids too frequent accounting and yet maintains accuracy with respect to the scheduler ticks. Similarly, the memory usage of each handler's data map is accounted for in the user space process during handler insertion.

eBPF improvements. In addition to the handler accountability property, XPS provides the following improvements over the existing eBPF interface. First, XPS provides an extension point in the TCP stack, unlike eBPF's existing extension points in the TC and XDP components. By using the TCP extension point, XPS avoids the need for any driver support, and reuses the existing protocol stack available in the Linux kernel.

Safety. Using the eBPF safety mechanism, XPS ensures that the kernel is secure when executing various application handlers (detailed in §2.1.3). It is worth noting that, even with these restrictions, eBPF is *expressive* enough to perform the latency-sensitive operations of an application. For example, operations such as handling `GET`s in key-value stores or restricting HTTP methods in web servers do not employ unbounded loops or complex operations. On the other hand, complex operations should be handled in the slow path. In

Table 3.3: Programming languages used for the handlers and the data structures used for the control and data abstraction in the kernel, user space, and smart NIC implementations of XPS.

Protocol stack	Handler	Handler data	Data map	Predicate table
Kernel	eBPF	isolated	eBPF map	eBPF map
User space	eBPF/C	shared	-	hash table
SmartNICs	eBPF/C	isolated	hash table	P4 tables

addition, XPS' L4 predicate ensures that the appropriate application handler is invoked, avoiding packet delivery to the wrong service.

User Space Protocol Stack

With the XPS user space TCP stack, the fast path is executed by the thread that performs the protocol processing and the slow path is executed by the service thread. The extension framework and fast-path handlers can be implemented as functions written in either C or eBPF that are simply hooked into the user space protocol stack using XPS' control APIs (see, §3.2.1). The slow path in the XPS user space stack leverages the batching mechanism, available in mTCP, amortizing the socket overhead. Since the service and the user space TCP stack directly run in the same address space, XPS' user space stack has key differences compared to the in-kernel one. First, in terms of *consistency*, the handler's data is shared between the service and the fast-path handlers, but the data should be guarded explicitly by a synchronization mechanism in the service. Second, in terms of *accountability*, the execution time taken for the fast-path handler is, by default, accounted to the service, requiring no separate accounting mechanism. Third, in terms of *safety*, any bugs in the fast-path handler are limited to the service's address space, requiring no explicit validation of the handler code.

Smart NIC

With XPS' abstractions, the fast-path handlers are invoked directly on the smart NIC, avoiding overheads associated with the PCIe and socket interfaces. The L4 and L7 predicates are implemented using P4 that parses the protocol headers, and invokes the application

Redis GET handler

```
int redis_get_handler(struct sock *sk,
                      struct sk_buff *skb) {
  redis_datakey_t k = {.len = 0};
  redis_value_t *value = NULL;

  redis_decode_payload(skb, &k);  ❶
  if (k.len) {
    value = bpf_map_lookup_elem(&datamap, &k);
    if (value) {
      if (redis_encode_payload(skb, value))  ❷
        return TX;
    }
  }
  return PASS;
}
```

❶ *Redis decode*

```
int redis_decode_payload(struct sk_buff *skb
                         redis_datakey_t *key) {
  int len = 0;
  int req_len = skb->proact_info->req_len;
  char *input = skb->proact_info->req;

  // check for GET request in the buffer
  len += redis_checkget(input, req_len);
  if (!len)
    return 0;

  // update key length and value
  redis_getkey(input, key, &len);
  return len;
}
```

❷ *Redis encode*

```
int redis_encode_payload(struct sk_buff *skb
                         redis_value_t *value) {
  int len = 0;
  char *output = skb->proact_info->res;

  // encode length into response
  len += redis_encode_len(output, len, value);
  if (!len)
    return 0;

  // encode value and end-of string CRLF
  len += redis_encode_value(output, len, value);

  return len;
}
```

Figure 3.3: An overview of the Redis `GET` handler operations using eBPF and the XPS framework. The handler contains a *decode* function that provides the key for the eBPF lookup. The *encode* function builds the response based on the outcome of the lookup.

handlers based on the predicate. The fast-path application handlers can be implemented as functions written in either C or eBPF that are provided with the parsed IP, UDP headers, and the payload as the parameters. Similar to the XPS kernel stack, the data map is explicitly isolated from the service and should be updated using XPS' data APIs that perform the updates over the PCIe interface, thus providing an eventual *consistency* guarantee. XPS supports fast-path accounting on NPU-based NICs by binding an application handler to a micro engine (ME), which does not interfere with another application handler. On ARM-based NICs, the kernel eBPF accounting mechanism provides fast-path *accounting* similar to the accounting mechanism on the host. Finally, any bugs in the fast-path handler are simply limited to the service and do not impact other services running on the host, satisfying the *safety* property. XPS' smart NIC prototype is implemented with the Netronome Agilo (2 GB memory) [7] NIC. Although we show XPS' hardware fast path with a Netronome NIC, the kernel fast-path mechanism using eBPF can be implemented on other ARM-based NIC cards, such as the Mellanox Bluefield NIC [6].

The current XPS prototype implements only a stateless UDP protocol: it allows executing the fast-path handler on the smart NIC, while the slow path is executed on the host. The fast path response, provided by the application handler, is encapsulated with the protocol headers, and transmitted from the NIC. The packets not matching the predicates are DMAed to the host for slow path processing. Because of the statelessness of UDP, the slow path

processing proceeds normally on the host even if some prior packets were processed in the fast path. However, with a stateful protocol, such as TCP, the following problem arises: packets handled by the fast-path handler on the smart NIC result in out-of-order handling for packets that are handled by the slow-path on the host. The reason for the above problem is that TCP states (*e.g.*, sequence numbers) are not synchronized between the smart NIC and the host. The TCP protocol implementation addressing these problems is part of our future work.

3.2.5 Memory Management

With the kernel and smart NIC prototype, XPS maintains a data cache in the kernel and smart NIC respectively, which is used to store the service data. The data abstractions allow an service to explicitly create, update, and delete the entries in the data cache, providing eventual consistency. However, some service need strict consistency. For such services, XPS provides strict consistency by updating the data cache before delivering the write requests to the services, similar to the consistency model provided by programmable switches [107].

For strict consistency, in addition to the fast-path handler for the read requests, a service should register a fast-path handler for processing the write requests, which updates the XPS data cache before delivering the write requests to the service. With the fast-path handler for write operations, with smart NICs, services avoid the explicit data updates over the PCIe interface.

3.3 Case Studies

We ported three types of services to take advantage of XPS, identifying appropriate operations and implementing them as fast-path handlers using XPS' abstractions; Table 3.5 describes them in detail.

Key-value stores: Redis and Memcached. With Redis, the latency-sensitive `GET` requests are handled in the XPS fast path, and the `SET` requests are handled in the slow path. Figure 3.3

shows the eBPF handler which handles the GET requests in XPS kernel protocol stack. The redis_get_handler performs the following operations: It decodes the request in the sk_buff structure via redis_decode_payload and provides the decoded key from a GET request. Subsequently, a key lookup is performed to find the value, which in turn is used to encode the response via redis_encode_payload. In the case where the value is successfully retrieved (i.e., it was decoded properly and the key was present in the data map), the redis_get_handler returns TX to transmit the response immediately, without delivering the request to the user space Redis. Otherwise, the return code PASS is used to indicate that the request has to be passed to the user space Redis. The Redis service logic is handled by the redis_get_key and the redis_encode_payload functions, while bpf_map_lookup_elem is a eBPF helper function to retrieve the handler data map. The SET requests in the kernel trigger the data updates to the GET handler's data map. With Memcached, we show the same use case using XPS' UDP stack implemented in a smart NIC.

Nginx. With Nginx, the functionality to black list requests based on the HTTP method (implemented in a Nginx service module), which should be performed with low overhead, is implemented using XPS' fast path. During initialization, Nginx inserts the fast path handler, and updates the data map with the HTTP method as the key and the HTTP status code (e.g., 405) as the value using XPS' data abstraction. Unlike Redis, Nginx slow path does not trigger an data update operation. Figure 3.2 shows an example of this functionality.

Distributed systems. In our experiment §3.5.2, **rediscluster** is unable to sustain the traffic from the client and the replica, resulting in replica heartbeat timeouts. To eliminate these timeouts, we use XPS' kernel stack to identify the process' status using its task structure and to handle cluster heartbeats in the fast path. In addition to heartbeat handling, we also handle GET requests from the clients, similar to the case of Redis with XPS. In this case, Redis Cluster inserts a GET handler on port 6379, which parses client requests and handles only GET requests. On the cluster port (16379), Redis Cluster inserts the composability handler, which retrieves the value from offset 12 to 14 (length 2 bytes) in the request, which

identifies the message type in the specific cluster message. Using the retrieved value, the composability handler invokes the heart beat handler that handles cluster heartbeats by identifying the process' status.

For **LogCabin** [114], a Raft protocol [18] leader handles the client requests and issues an `AppendEntries` RPC to the other servers for replicating log entries. On the other servers, we use XPS' kernel stack to handle the `AppendEntries` RPC in the fast path. The fast-path handler's data map is used to append the log entries. Unlike Redis, LogCabin's slow path does not trigger an data update operation.

3.4 Implementation

XPS' kernel prototype extends Linux 4.8-rc1 and modifies the kernel TCP stack to insert and invoke its handlers. `tcp_rcv_established()` and `tcp_input()` invoke the extension framework with a pointer to the `sk_buff` and `sock` structures, and, based on the return code from the handler, deliver the read/write events to the user space service. To perform proper accounting of application handlers the `task_struct` contains an additional field to keep track of cycles spent.

XPS' user space prototype extends mTCP [25] to insert and invoke the extension framework before delivering events to the services. `ProcessTCPPayload` invokes the extension framework with the TCP payload and the `cur_stream` metadata and, based on the return code from the handler, delivers the read/write events to the service. XPS' smart NIC UDP prototype implements the predicate table using P4. The application handler on the smart NIC shares memory with the host using the `__export` and `__emem` macro which the XPS library, on the host, reads and writes to.

3.5 Evaluation

In the evaluation, we answer the following four questions:

- How much effort is required for existing services to adopt XPS? (§3.5.1)

Table 3.4: The machine configurations used to evaluate XPS.

Machine Type	Commodity data center [115]	Large NUMA
Model	E5-2630 v3	E7-8870 v2
Frequency	2.40 GHz	2.30 GHz
# Cores	16	120
Layout (cores × sockets)	8 × 2	15 × 8
RAM	128 GB	768 GB
LLC (size × sockets)	20 MB× 2	30 MB× 8
L1 D-TLB entries (per core)	64	64
L2 TLB entries (per core)	1024	512
Hyperthreading	Disabled	Disabled

Table 3.5: The usage of XPS' lookup operation demonstrating the applicability of XPS for a range of services, along with the lines of code for the application handlers (eBPF) and for integrating XPS to the respective services.

Type	Application	Functionality	Data map lookup	Handler (eBPF/C)	Application (modified / baseline (%))
Web server	Nginx	Blocking POST method	Status code using HTTP method	157	149 / 118,452 (0.13%)
Key-value store	Redis-Linux	GET request handling	Value using key	266	130 / 44,069 (0.29%)
	Redis-mTCP			326	143 / 44,069 (0.32%)
	Memcached			513	154 / 17,014 (0.91%)
Composed Redis	Redis & Redis Cluster	GET and heartbeat handling	Comb. Redis & Redis Cluster	460	192 / 44,069 (0.44%)
Distributed system	Logcabin	AppendEntries RPC handling	Append log entries	162	145 / 31,126 (0.47%)

- What are XPS' benefits in terms of tail latency and throughput for services? (§3.5.2)
- What is the impact of XPS' fast-path processing on the service's slow-path processing? (§3.5.2, §3.5.2)
- What are the costs for the operations of XPS? (§3.5.3)

Experiment setup. We evaluate XPS on three machine setups, as shown in Table 4.2. The Linux kernel and mTCP are evaluated using the Mellanox ConnectX-2 NIC (40G), and IX and ZygOS are evaluated using the Intel 82599ES NIC (10G). We used and extended Redis 3.2.6, Memcached 1.5.7, Nginx 1.10.2, and the most recent version of LogCabin from Github. For the Redis experiments, we used 32-byte keys and 128-byte values with 2 million entries that are available in XPS data cache. We measured the latency and throughput using Memtier [116] with 100 TCP connections for a one-minute run of the experiment. For mTCP [25], we modified Redis to use the mTCP sockets and configured mTCP to use Intel

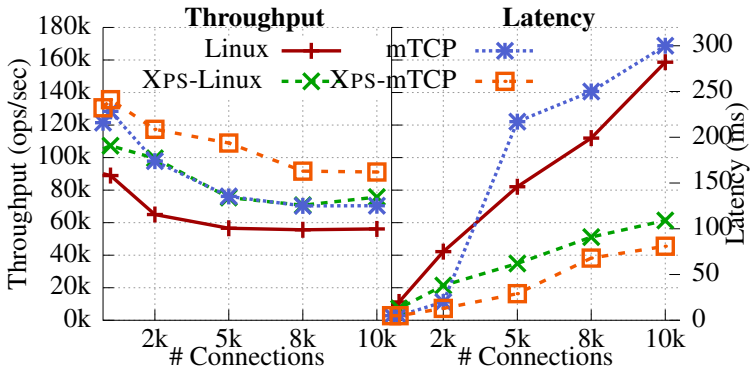


Figure 3.4: The impact of increasing connections using RedisBenchmark in the bare-metal setup. XPS retains a higher throughput and lower 99th percentile latency with increasing connections.

DPDK [24]. For IX [22] and ZygOS [23], we modified Redis to use the IX event system calls. For all experiments, the clients use Linux BSD sockets. IX, ZygOS, and mTCP use a batch size of 64, unless otherwise stated. For Nginx, we use the Wrk benchmark [117], which provides the latency and throughput for HTTP requests. Henceforth, XPS-mTCP refers to XPS' user space TCP stack (with mTCP), while XPS-Linux refers to XPS' kernel TCP stack.

3.5.1 Required Porting Efforts

XPS is easy to adopt for real-world services. For example, four types of fast-path handlers consist of around 100 to 500 LoC (see Table 3.5). To adopt these service handlers, we modified 100 to 200 LoC (less than 0.9% of changes of the original services) using XPS' user space library. The XPS library comprises 2,132 LoC and implements the APIs described in Table 3.2. We show that XPS' approach is practical and easy to adopt for real-world services.

3.5.2 Performance of Ported Services

We evaluate XPS with five services: Redis and Memcached NoSQL stores with varying read/write workload patterns, Nginx with HTTP filtering and blocking, Redis Cluster handling the cluster heartbeat messages, and LogCabin handling consensus messages.

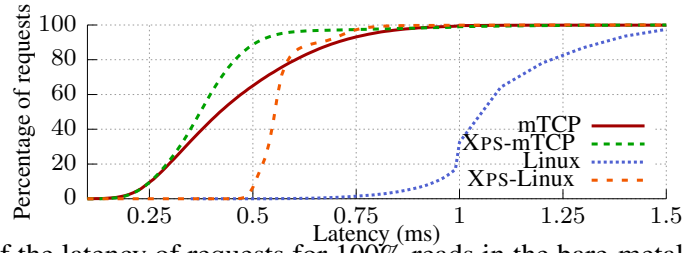


Figure 3.5: CDF of the latency of requests for 100% reads in the bare-metal setup. XPS shows a significantly lowered latency compared to Redis on Linux and mTCP.

Redis

Read performance. We first evaluate the impact of the fast path provided by XPS on read-intensive workloads after loading Redis with 2 million entries that are available in XPS data cache and by varying the number of connections using RedisBenchmark from 100 to 10,000 in Figure 3.4. We evaluate this experiment in the bare-metal setup with a read-only workload for 1024 byte values, in line with real cloud deployments [118]. The results show that with increasing connections, XPS retains the advantage of using the fast path by achieving both higher throughput and lower 99th percentile latency. In particular, with 10,000 connections, XPS-Linux improves the baseline Linux throughput by up to 34.9%, while XPS-mTCP improves the baseline mTCP throughput by up to 29.3%. Additionally, XPS-Linux reduces the 99th percentile latency of Linux by up to 61.3%, while XPS-mTCP reduces the 99th percentile latency of mTCP by up to 73.3%.

To understand XPS' performance benefits over Linux and mTCP, we measure the cache misses of Linux and mTCP with Redis (see Table 3.6) with 100 connections. mTCP shows increased L3 and L2 cache misses compared to Linux as an artifact of mTCP's batching, whereas XPS-Linux reduces L3 cache misses by 21.1% compared to Linux. Similarly, XPS-mTCP reduces the number of L3 cache misses by up to 66% compared to mTCP, demonstrating the cache locality benefits of XPS' immediate handler execution.

To further understand the latency behavior with XPS' fast-path processing, Figure 3.5 shows the cumulative density function (CDF) for read requests in the bare-metal setup. XPS-Linux shows a consistently lower latency than Redis on Linux (42.5% lower 99th percentile latency), and both XPS-Linux and XPS-mTCP provide a reduced 99th percentile

Table 3.6: Cache misses of Redis on XPS, mTCP (with a batch size of 64 and 128), and Linux for 100% reads in the bare-metal setup. XPS shows up to 66% lower cache misses compared to both mTCP and Linux.

	Linux (In kernel)			mTCP (User space)			
	Stock	XPS		mTCP-64	XPS	mTCP-128	XPS
L2 hit	0.54	0.60	(+11%)	0.45	0.55 (+22%)	0.31	0.53 (+71%)
L3 hit	0.99	0.99	(±0%)	0.87	0.95 (+9%)	0.83	0.95 (+14%)
L3 miss	114k	90k	(-21%)	630k	260k (-59%)	940k	320k (-66%)

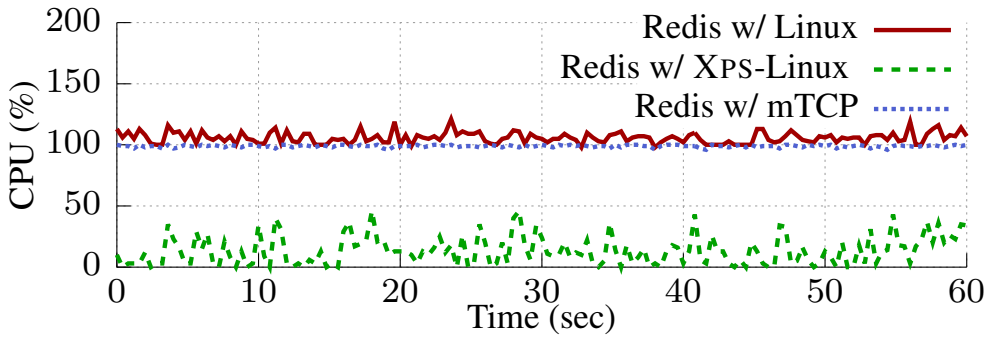


Figure 3.6: The CPU utilization, normalized to a single core, with 100% reads in the bare-metal setup. XPS-Linux shows much lower overall CPU utilization than Redis on Linux and mTCP, even though XPS improves both throughput and latency.

latency compared to mTCP. In addition, XPS-Linux reduces CPU utilization by 50-80%, as shown in Figure 3.6. By using immediate handlers, XPS eliminates the socket overhead shown in Table 2.1 in both the kernel and the user space, along with the additional service scheduling and cache pollution overheads. With XPS-Linux, the processing time is spent in the network RX `softirq` layer in the Linux kernel. With XPS-mTCP, the processing time is spent in the TCP thread. With these results, we conclude that, for a read-intensive workload that takes the fast path, XPS-Linux and XPS-mTCP provide significant improvements to both throughput and latency. In addition, XPS-Linux provides reduced CPU usage.

Read and write performance. We evaluate the throughput and 99th percentile latency of Redis and XPS-Linux with a varying percentage of reads and writes (`GET` and `SET` operations in Redis), which shows the impact of the fast path on slow-path operations. The results are presented in Figure 3.7 and show a higher throughput, not only for `GET`s, but also for `SET` operations across different workloads. This highlights the benefit of XPS, speeding up not

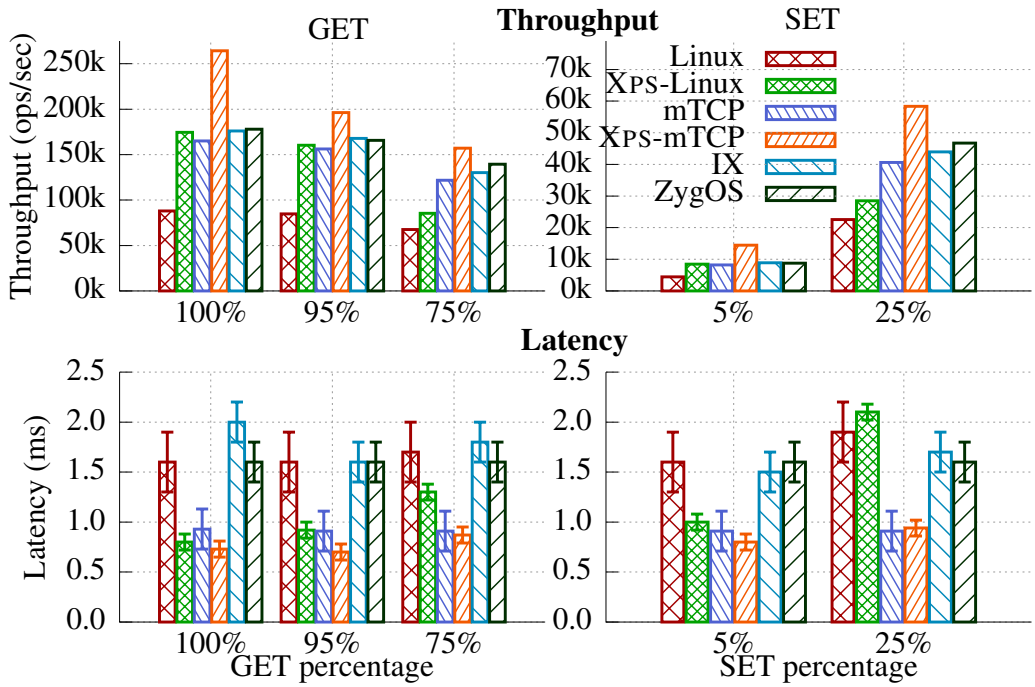


Figure 3.7: Throughput and 99th percentile latency of varying the read/write ratio of the requests. XPS outperforms the baseline for both `GET` *and* `SET` operations for all cases by up to 98.1% (`GET`) and 88.9% (`SET`), showcasing XPS' ability to improve both slow- *and* fast-path operations.

only the fast-path processing (`GET` requests), but also the slow-path processing (`SET` requests) by limiting the socket layer only for the slow path.

For a read-intensive workload (5% `SET`s), XPS improves the throughput of both `GET`s and `SET`s by up to 80%. With respect to the 99th percentile latency, read and write latency is improved by up to 40%. With a write-intensive workload that has 25% `SET`s, XPS improves the fast- and slow-path throughput by up to 25%. With respect to the 99th percentile latency, read latency is improved by up to 22%. However, the increased 99th percentile write latency stems from two factors: the handler decode cost, and the delay in delivering write events to the service when the reads are processed by the handlers. However, the handler decode cost is minimal (≈450 ns), and the queuing delay for delivering the slow-path events dominate. To understand the detailed write latency behavior, we show the CDF for 25% writes in Figure 3.8. The results show that up to the 85th percentile, XPS' write latency is lower, and beyond that the slow-path event's queuing delay dominates. The improved XPS 85th percentile write latency is the reason for the improved write throughput. With

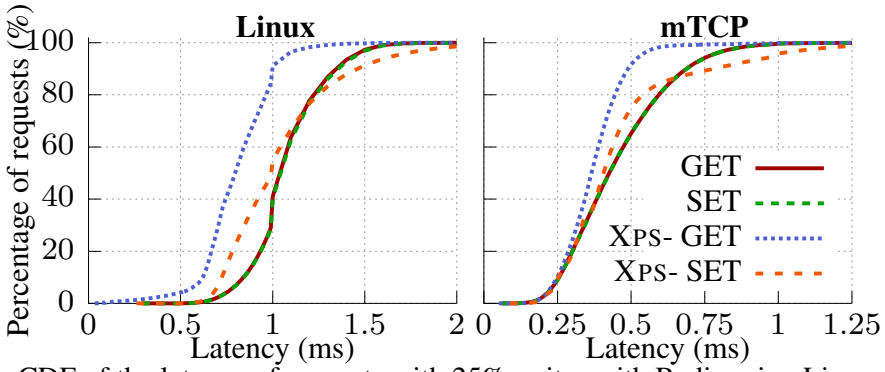


Figure 3.8: CDF of the latency of requests with 25% writes with Redis using Linux, mTCP, and XPS. XPS-Linux and XPS-mTCP show a significantly lowered latency compared to the baseline for GETs and for SETs up to the 85th percentile. Note: GET and SET overlap for both mTCP and Linux.

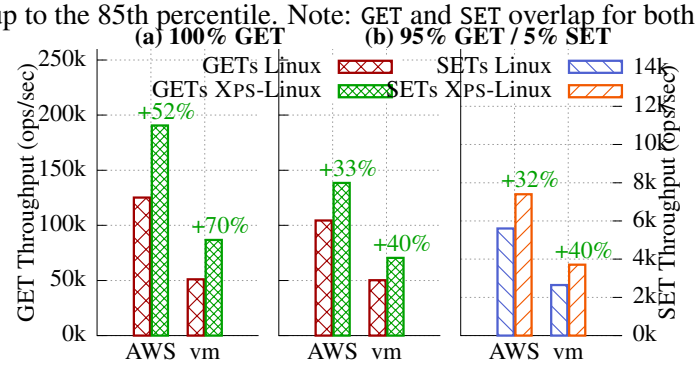


Figure 3.9: Throughput of Redis running on AWS and a self-hosted VM. XPS-Linux improves the throughput by 52.2% for a read-only workload and by 32.7% for a 95% read workload on AWS.

these results, we conclude that for write-intensive workloads (25% SETs), the co-existence of slow- and fast-path processing improves the throughput for both GETs and SETs, at the same time reducing the latency for reads and 85% of the SETs.

Figure 3.7 compares the performance of XPS-Linux and XPS-mTCP with both IX and ZygOS. Though XPS-Linux's performance improves over the baseline due to eliminating the socket interface overheads, it still suffers from synchronization and bloated metadata overheads. However, XPS-mTCP eliminates the above overheads [25] and the batching overhead, which improves the throughput and latency compared to both IX and ZygOS. With a read-intensive workload (5% GETs), XPS-mTCP improves the throughput by up to 50.1% and the latency by up to 63.5%, compared to IX and ZygOS. With a write-intensive workload (25%), compared to IX and ZygOS, XPS improves throughput of reads by up to 20.6% and throughput of writes by up to 32.7%.

Evaluation on local VMs and AWS. To show the impact of XPS-Linux in a VM using SR-

IOV, we evaluate XPS with Redis using a read-intensive workload. We show the throughput of both Redis and XPS with both 100% and 95% GETs in Figure 3.9. XPS improves the throughput by up to 80% and, in addition, improves SET operations. Additionally, XPS reduces the 99th percentile latency for both GETs and SETs by up to 51.0%.

To show the impact of XPS-Linux in a real-world cloud environment, we evaluate XPS with Redis using a read-intensive workload on Amazon AWS, using `r3.2xlarge` VMs with Intel 10G interfaces. We show the throughput of both Redis and XPS using both 100% and 95% GETs in Figure 3.9. XPS is able to improve the throughput by up to 52.2% and in addition improves SET operations. Additionally, XPS reduces the 99th percentile latency for GETs by 46.6% and SETs by 50.8%.

Comparison with Arrakis. We compare the performance of XPS and Arrakis using Redis with 100% reads. For Arrakis, we used the Barrelfish operating system that has the merged Arrakis code. We fixed bugs in Arrakis source code, such as releasing the TX buffers, to evaluate Redis with Arrakis. In addition, Arrakis does not support congestion control in the TCP stack. Though the Arrakis paper reports Redis' read performance to be 250K PPS, the Arrakis version evaluated with Barrelfish shows 154K PPS. Arrakis shows higher latency in handing over the packets to the service which reduces its throughput(as shown in Figure 3.10). Both Linux and XPS shows improved performance compared to Arrakis. In conclusion, this shows the benefit of XPS providing a mechanism for latency-sensitive operation in Linux compared to developing a new operating system.

Memcached

With Memcached, 20% of the total keys are stored in the smart NIC data cache, and the remaining keys are stored in the host. The read requests for the 20% keys are handled in the fast path, and all the other requests are handled in the slow path. In addition, L7 predicates are used to invoke the GET handler from P4. We use Mutilate [119] to measure the latency and throughput of requests using the Facebook data set with a Zipf distribution

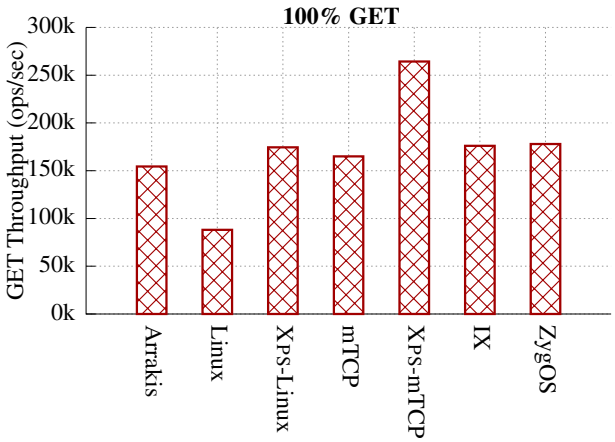


Figure 3.10: Comparison of redis' throughput with 100% gets in Arrakis, Linux, XPS-linux, mTCP, XPS-mTCP, IX and Zygos. XPS-mTCP outperforms all the other systems, and XPS-linux provides similar performance as IX and Zygos.

that considers the keys in the smart NIC as the popular keys [120]. Figure 3.11 shows Memcached's performance for different workloads. XPS' hardware fast path shows an improvement of up to 4.0× for all workloads, and for both read and write operations. In addition, the latency of both the read and write operations is reduced by up to 58.8%.

XPS' hardware fast path overcomes the overheads predominant in a software approach (XPS-Linux and XPS-mTCP). First, decoding the service payload in hardware does not incur the same overheads as the software mechanisms. Second, the PCIe overhead incurred in a software-based approach is substantially reduced by executing the fast path in hardware. Finally, the host CPUs are used only for the slow path, drastically improving the slow path performance.

Redis Cluster

To demonstrate the flexibility provided by XPS in composing L7 handlers, we evaluate Redis Cluster with XPS-linux handling both the client's `GET` requests and the cluster `PING`s (i.e., heartbeat messages) using fast-path processing in Figure 3.12. In our experiment, we configure Redis Cluster to generate a `PING` message every 2 ms, similar to [16, 92]. The master node handles the client's `GET` and the cluster `PING` messages, while the replica handles only the `PING` messages. In this experiment, with one master and one replica, the

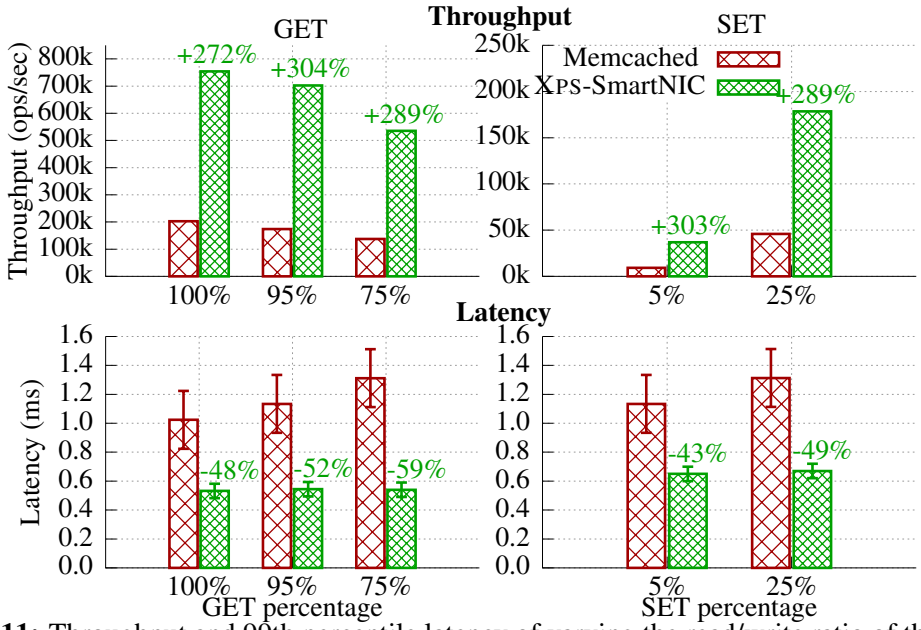


Figure 3.11: Throughput and 99th percentile latency of varying the read/write ratio of the requests for Memcached with both the host and the XPS smart NIC. XPS improves the throughput by up to 4.0× (`GET` *and* `SET`) and reduces latency by up to 58.8% (`GET`) and 49.0%(`SET`), showcasing XPS' ability to improve the slow path by running the fast path in hardware.

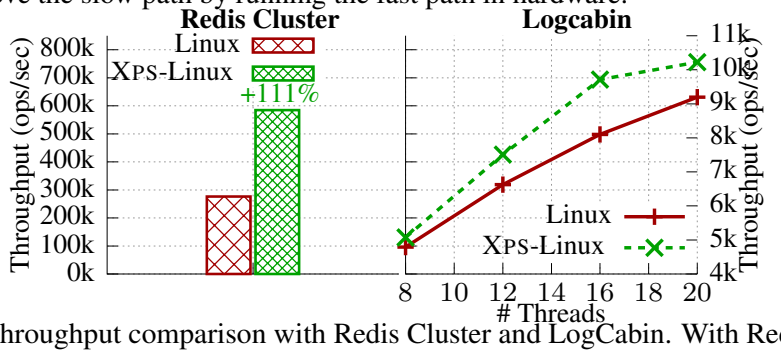


Figure 3.12: Throughput comparison with Redis Cluster and LogCabin. With Redis Cluster, XPS-Linux sustains a 2.1× higher throughput while still answering all `PING` requests. With LogCabin, XPS-Linux improves the write throughput by up to 19.9%.

Redis Cluster master node is unable to sustain both the traffic from the client and the replica, resulting in 4.6 timeouts per second during a 1-minute experiment. With XPS, the master node not only handles 2.1× more messages but also responds to all cluster `PING` messages without any timeouts.

LogCabin

We evaluate LogCabin by configuring a three-node setup with VMs, where one VM acts as the Raft leader. The client requests are served from the leader, and the leader issues `AppendEntries` RPCs to the other servers, which are handled in the fast path on the other

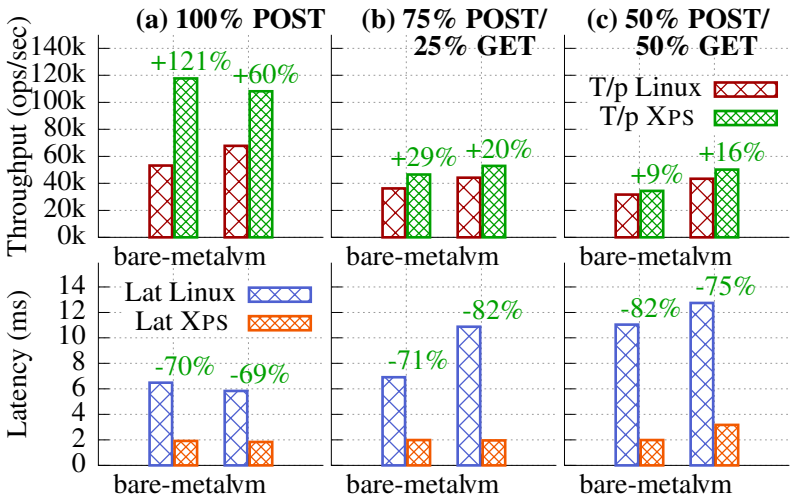


Figure 3.13: Throughput and 99th percentile latency for HTTP filtering and blocking with Nginx by blocking `POST` requests. XPS improves the throughput by up to 2.2× while reducing the 99th percentile latency by up to 82.0%.

servers. We use the LogCabin benchmark, which repeatedly writes 1,000 values to evaluate the service throughput. With XPS, the leader handles up to 19.9% more writes because the other servers respond faster to the `AppendEntries` RPC (see Figure 3.12).

Nginx

With Nginx, we show the benefits of providing HTTP filtering and blocking using XPS-Linux (as shown in §3.3). Figure 3.13 shows the results, which demonstrate that XPS improves the throughput by up to 2.2× for handling 100% `POST` requests. For 75% and 50% `POST` requests, XPS improves the throughput of both methods. Additionally, the 99th percentile latency is reduced by at least 68.6% across all cases. The latency improvement with XPS is attributed to two reasons: In user space, the entire HTTP request is parsed before taking action based on the method, whereas XPS' Nginx handler decodes only the HTTP method to perform the action. Additionally, `GET` processing in Nginx does not trigger data updates.

Table 3.7: Execution time for the XPS operations in the kernel.

Operation	Redis	Nginx	Redis Cluster	LogCabin
Framework load	3.4 ms	2.8 ms	2.4 ms	2.8 ms
`create_context()`	131.4 ms	4.1 ms	137.4 ms	130.8 ms
`update_context()`	13.3 ms	14.2 ms	16.6 ms	15.6 ms
`data_put()`	12.9 μs	10.6 μs	8.1 μs	NA
`data_update()`	9.1 μs	2.8 μs	3.3 μs	NA
`data_delete()`	5.9 μs	2.0 μs	2.2 μs	NA

3.5.3 Performance Breakdown

Table 3.7 shows the cost of control and data operations in XPS Linux. The control operation `create_context()` loads the handler in the kernel and updates the predicate table, which takes up to 140 ms, for all the handlers. For handler loading, the high overhead stems from transferring the eBPF byte code to the kernel and verifying the byte code to maintain integrity. The lightweight extension framework loads faster than the handlers, which shows that both the transfer and verification time depend on the size of the handlers. In addition, the Nginx handler loads faster without the composability handler. For `update_context`, the transfer and verification processes are already performed and only the predicate table is updated, drastically reducing the time. Due to the overhead of `create_context`, handlers are loaded during initialization and further updates are handled at run time. For the user space stack, the control operations are performed using function pointers, which is negligible. For the smart NICs, the handlers are statically loaded while loading the firmware, which takes a few hundred milliseconds.

For the data operations in the kernel, we observe a maximum of 12.9 μs for `data_put` on Redis when inserting a 1KB value, while operations using Nginx and Redis Cluster are less expensive due to smaller data sizes. With LogCabin, data is not provided from user space using data operations. Data operations are not applicable in the user space stack because the data is shared with the service.

3.6 Limitations

In our current implementation, we identify three limitations. First, the current XPS kernel implementation uses an isolated data map, which could be further improved by employing a shared memory between the service and the kernel. Second, operations such as put operation in key-value store need to update the global data which is only available in the user space which limits XPS from performing such operations in the slow-path. Third, operations in key-value stores that operate on larger sets of data ,such as SORT, LREM, SUNION, incur latency in the order of milli seconds, which are not the right fit for the fast-path operations. Such operations will block the protocol processing layer in the kernel/user-space from reading further packets from the NIC which could result in packet drops on the NIC.

Comparison with offloading approaches. Web servers such as IIS [121, 122] have an HTTP driver to provide kernel-mode caching, which serves frequently visited static pages. But, unlike XPS, the kernel HTTP drivers do not provide a generic approach, making them specific HTTP protocol that is implemented in the kernel. In contrast, XPS enables augmenting the protocol stack with any service-specific code, rendering it a generic framework applicable to the kernel stack, user space stacks, and smart NICs.

3.7 Chapter Summary

We introduced XPS, which provides a generalized solution to enable the co-existence of both slow- and fast-path processing in the kernel and the user-space protocol stacks. In addition, XPS' abstractions enable real-world services to take advantage of fast path processing on smart NICs. We demonstrated the benefits of XPS using real-world services and improved their throughput and tail latency by up to 4.0× and 82%, respectively. In addition to the protocol stack, other synchronous operations in the operating system and distributed system impact service latency. Next, We present LATR which provides an asynchronous mechanism for TLB shootdown.

CHAPTER 4

LATR: ADDRESSING LATENCY INCURRED BY SYNCHRONOUS OPERATIONS

4.1 Introduction

Translation lookaside buffers (TLBs) are frequently accessed per-core caches that store recently used virtual-to-physical address mappings. TLBs enable fast virtual address translation that is critical for service performance. Since TLBs are per-core caches, TLB entries should be kept coherent with their corresponding page table entries. The lack of hardware support for TLB coherence implies that software should provide the necessary coherence. In most existing systems, system software such as an operating system (OS) maintains the TLB coherence with the page table.

To provide TLB coherence, an OS performs a *TLB shootdown*, which is a mechanism to invalidate stale TLB entries on remote cores. TLB shootdowns are triggered by various virtual memory operations that modify page table entries, such as freeing pages (`unmap()`), page migration [123, 13], page permission changes (`mprotect()`), deduplication [124, 125], compaction [126], and Copy-on-Write operations (CoW).

Unfortunately, the existing TLB shootdown mechanism is very *expensive*—a shootdown takes up to 80 μs for 120 cores with 8 sockets and 6 μs for 16 cores with 2 sockets that is a widely used configuration in modern data centers [115]. This is mainly because most existing systems use expensive inter-processor interrupts (IPIs) [1] to deliver a TLB shootdown: *e.g.*, an IPI takes up to 6.6 μs for 120 cores with 8 sockets and 2.7 μs for 16 cores with 2 sockets. Even worse, current TLB shootdown mechanisms handle invalidation in a *synchronous* manner. That is, a core initiating a TLB shootdown first sends IPIs to all remote cores and

[1] IPIs are the mechanism used in x86 to communicate with different cores.

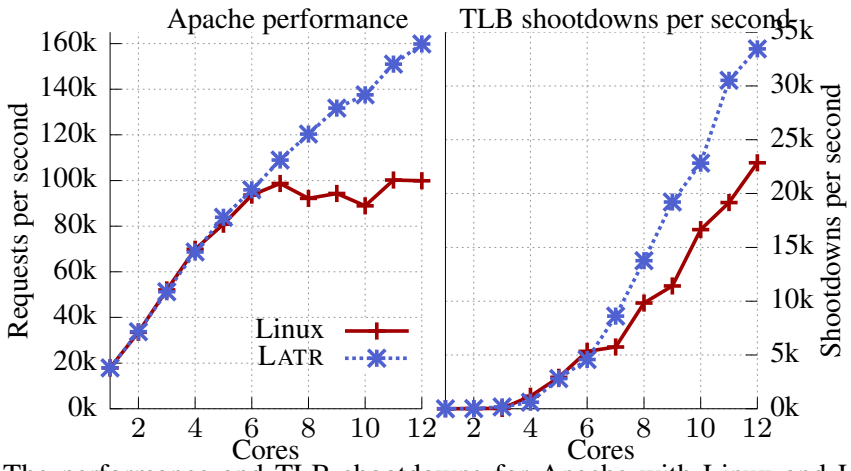


Figure 4.1: The performance and TLB shootdowns for Apache with Linux and LATR. LATR improves Apache's performance of serving 10 KB static web pages by 59.9%; it removes the cost of TLB shootdowns from the critical path and handles 46.3% more TLB shootdowns.

then waits for their acknowledgments, while the corresponding IPI interrupt handlers on the remote cores complete the invalidation of a TLB entry (see §2.2 for details).

Such an expensive TLB shootdown severely affects the overall performance of services that frequently trigger memory management operations that change the page table [38, 49], such as web servers like Apache [127] and data analytic engines like MapReduce [128, 129]. For example, a typical Apache workload serving small static web pages or files does not scale beyond six cores with the current TLB shootdown mechanism in Linux (see Figure 4.1). Once alleviated by our new mechanism, Apache can handle 46.3% more TLB shootdowns and thus improve its throughput by 59.9%. More importantly, it is all possible without any service-level modifications.

To solve this problem, there have been two broad categories of research, namely, hardware- and software-based approaches. Hardware-based approaches strive to provide TLB cache coherence in hardware (see §2.2.2), but require expensive hardware modifications and introduce additional verification challenges to the microarchitecture, which is known to be bug-prone [39, 40, 41, 42, 43, 44, 45]. Software-based approaches, on the other hand (see §2.2.3), focus on reducing the number of necessary IPIs to be sent, either by batching TLB invalidations (*e.g.*, identifying the sharing cores [49]), or using alternative mechanisms instead of IPIs (*e.g.*, message passing [50]). However, current software-based approaches

still handle TLB shootdowns *synchronously* and do not eradicate the overheads associated with the TLB shootdown. It means that, even with a message-passing alternative [50], a core initiating a TLB shootdown should wait for acknowledgments from participating remote cores. A synchronous TLB-shootdown mechanism increases the latency by several microseconds for certain virtual address operations, which is known to a culprit that contributes to the tail latency of some critical services in data centers [2].

To solve this inherent synchronous behavior of TLB shootdowns, we propose a software-based, *lazy shootdown* mechanism, called LATR, that can asynchronously maintain TLB coherence. The key idea of LATR is to use *lazy memory reclamation* and *lazy page table unmap* to perform an asynchronous TLB shootdown. By handling TLB shootdowns in a lazy fashion, LATR can eliminate the performance overheads associated with IPI mechanisms as well as the waiting time for acknowledgments from remote cores. In addition, as a software mechanism, LATR is readily implementable in commodity OSes. In fact, as a proof-of-concept, we implement LATR in Linux 4.10.

We enumerate in Table 4.1 the operations in which a lazy TLB shootdown is possible. For *free* operations, such as `munmap()` and `madvise()`[2], lazy memory reclamation enables a lazy TLB shootdown. Similarly, a lazy TLB shootdown is applicable to *migration* operations, such as AutoNUMA page migration, where page table entries can be lazily unmapped to enable a lazy TLB shootdown. However, LATR's lazy approach is not applicable to operations such as permission changes, ownership changes, and remap (`mremap()`), where the page table changes should be synchronously applied to the entire system. LATR supports common operations, such as free and migration, and improves real-world services such as Apache, Graph500, PBZIP2, and Metis. In addition, the proposed lazy migration approach can play a critical part in emerging systems using heterogeneous memory where pages are migrated to faster on-chip memory [12, 13, 14] and in emerging disaggregated memory systems in data centers where pages are swapped to remote memory using RDMA [15, 5].

[2]For example for the case of `MADV_DONTNEED` and `MADV_FREE`.

However, there are a few challenges in handling TLB coherence in a lazy manner. First and foremost, LATR should guarantee the correctness of the new, lazy approach (*i.e.*, how does LATR ensure that stale entries do not have negative, or adversarial impacts on the kernel and to service?). We laid out the correctness sketch in §4.3.2 and §4.3.3. Second, a lazy TLB mechanism should make non-trivial design decisions: how the shootdown information is communicated to the remote cores without relying on IPIs, and when the remote cores should invalidate their TLB entries (§4.3.1).

We developed LATR as a proof-of-concept in Linux 4.10, and compared it with both Linux 4.10 and ABIS [49], a recent, state-of-the-art approach to reduce the number of TLB shootdowns that can be complementary to LATR. LATR makes the following contributions:

- LATR provides a lazy TLB shootdown for *free* operations using a lazy memory reclamation mechanism, and for *migration* operations using a lazy page table unmap mechanism.
- We also reason about why such lazy operations are still correct for both free and migration operations in commodity operating systems.
- We demonstrate LATR's approach is effective on both small (2 sockets, 16 cores) and large NUMA machines (8 sockets, 120 cores) when running real-world services (Apache, PARSEC, Graph500, PBZIP2, and Metis). With a large NUMA machine, LATR reduces the cost of `munmap()` by up to 66%. In addition, LATR improves Apache's performance by up to 37.9% compared to ABIS and 59.9% compared to Linux.

4.2 Overview

LATR proposes a lazy TLB shootdown approach for virtual memory operations such as *free* (e.g., `munmap()` and `madvise()`) and *page migration* (e.g., AutoNUMA page migration), as shown in Table 4.1. The key idea that drives LATR is the *delayed* reuse of virtual and physical memory for free operations. Currently, the immediate reuse of the virtual and physical pages

Table 4.1: Overview of virtual address operations and whether a lazy TLB shootdown is possible. A lazy TLB shootdown is not possible when PTE changes should be immediately applied to the entire system for their correct behavior.

Classification	Operations	Lazy operation possible
Free	`munmap()`: unmap address range	✓
	`madvise()`: free memory range	✓
Migration	AutoNUMA [123]: NUMA page migration	✓
	Page swap: swap page to disk	✓
	Deduplication [124, 125]: share similar pages	✓
	Compaction [126]: physical pages defrag.	✓
Permission	`mprotect()`: change page permission	-
Ownership	CoW: Copy on Write	-
Remap	`mremap()`: change physical address	-

involved in a `munmap()` operation necessitates an immediate TLB shootdown, *e.g.*, via a mechanism like inter-processor interrupts (IPIs). However, considering the large virtual address space (2^{48} bytes) and the amount of RAM (64 GB and more) available in current servers, not reusing the virtual and physical pages immediately enables an asynchronous TLB shootdown.

Support for free operations. LATR relies on the following invariant for the correctness of free operations: virtual and physical pages can be reused only after the associated TLB entries have been cleared on all cores. To ensure that the invariant holds for a free operation, LATR stores the virtual and physical pages to be freed in a separate lazy-reclamation list instead of adding them to the free pool immediately, which avoids their immediate reuse across all cores. LATR issues a local TLB invalidation for the TLB entries on the current executing core. In addition, instead of sending IPIs to the other participating cores, the state needed for the TLB shootdown is recorded in per-core invalidation states (referred to as LATR states §4.3.1). During a context switch or scheduler tick, the participating cores perform a local TLB invalidation by sweeping the other cores' LATR states via regular memory reads. The context switch and scheduler tick provide a periodic transition to the OS, which provides the opportunity to perform the state sweep and a TLB invalidation. Since the

TLB invalidation is performed during a context switch or scheduler tick, the scheduler tick interval (1 ms in Linux x86) establishes an upper bound time limit for a TLB shootdown. Based on this upper bound, LATR releases the virtual and physical pages using a background thread after a scheduler tick on the cores. As these scheduler ticks are not synchronized across all the cores, LATR delays the reclamation by twice the scheduler tick interval (2 ms).

Support for page migration operations. In addition to free operations, LATR's lazy TLB shootdown mechanism is applicable to page migration (*e.g.*, AutoNUMA in Linux). For AutoNUMA, lazily changing the page table enables a lazy TLB shootdown. In LATR, the background task records the LATR state without changing the page table immediately. During the scheduler tick, the first core that reads the LATR state changes the page table, followed by local TLB invalidation. The other cores that read the LATR state during the scheduler tick, perform only the local TLB invalidation. The existing AutoNUMA page-fault handling and page-migration algorithm handle page migration, which is not modified by LATR. A similar algorithm can be used for other migration operations such as page swapping, deduplication, and compaction. For example, with a least recently used (LRU) based page swapping algorithm, the page table unmap and swap operation can be performed lazily after the last core has invalidated the TLB entry. LATR's proposed lazy AutoNUMA and page swap algorithms are important for emerging systems with heterogeneous memory [13] and disaggregated memory [15, 5].

4.3 Design

Using the idea introduced in §4.2, we describe the design of LATR for x86-based Linux machines in detail. We first introduce the states needed by LATR (referred to henceforth as LATR states), and explain the TLB shootdown operation using the LATR states. Using the LATR states component, we describe the free operations (*e.g.*, `munmap()` and `madvise()` in §4.3.2) and migration operations (*e.g.*, AutoNUMA in §4.3.3). An overview of the LATR states is given in Figure 4.2.

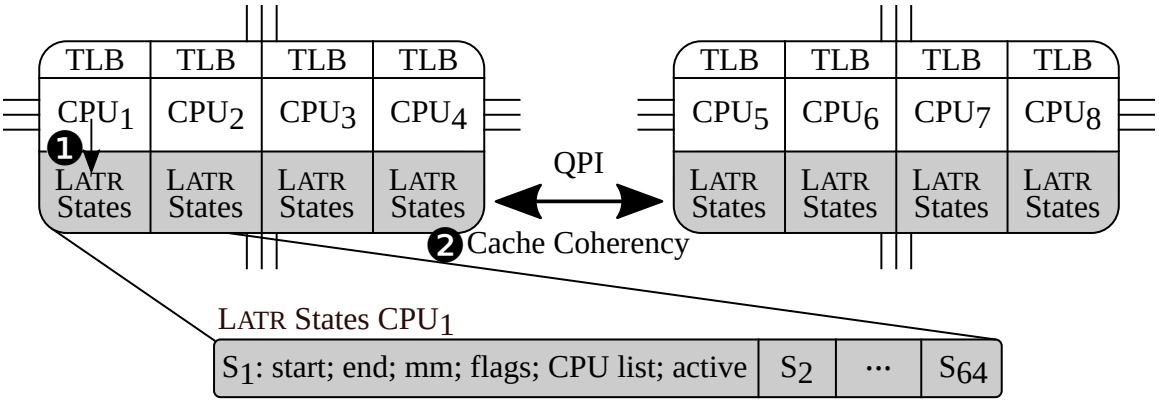


Figure 4.2: Overview of LATR's interaction with the system and its data structures. LATR uses per-CPU *states* (❶) to identify the cores included in a TLB shootdown. The states are made accessible to remote cores via the cache coherence protocol (❷) that remote cores use to clear entries in their local TLB.

4.3.1 LATR States

LATR saves the shootdown information in the LATR states, which are used for asynchronous TLB invalidation. The LATR states are a per-core cyclic lock-less queue (as shown in Figure 4.2), which is allocated from a contiguous memory region. Each entry in the LATR states holds the following information: the addresses *start* and *end* of the virtual address for the TLB shootdown, a pointer to the `mm_struct` to identify the current running process, a *bitmask* to identify the remote CPUs involved, *flags* to identify the reason for the shootdown (*e.g.*, to distinguish migration and free operations), and an *active* flag. The CPU bitmask identifies the remote cores on which the TLB invalidation should be performed for a particular entry's address. The active flag is used to identify currently valid entries. To ensure ordering between memory instructions, an entry is activated after setting all the fields using an atomic instruction coupled with a memory barrier.

Storage overhead. In LATR, each core stores 64 LATR states. The size of each state is 68 B, while all LATR states on a system with 32 cores amount to 136 KB, occupying less than 1% of the last-level caches (LLC) of recent processors (*e.g.*, at least 16 MB for an 8-core Intel CPU [130]). Even on an 8-socket, 192-core machine, the total size of all LATR states grows to only 816 KB, which corresponds to less than 1.3% of the LLC [131].

State update. The core initiating the TLB shootdown sets all fields of a LATR state,

including the CPU bitmask. Currently, Linux calculates the CPU bitmask for sending IPIs based on the cores where the process is currently scheduled. LATR uses the same logic to update the CPU bitmask in the state. Since the states are in memory, the state updates are available to all other cores using the cache-coherence protocol. We show an example in Figure 4.3, where CPU_1 initiates the TLB shootdown. It includes CPU_2 and CPU_5 in the bitmask, as the process is currently scheduled on both these CPUs (❶). The state update in LATR eliminates the overhead of sending IPIs waiting for the `ACK`s that present in Linux and existing OSes.

Asynchronous remote shootdown. During a periodic interval (scheduler tick or context switch), each core sweeps the LATR states of all available cores. The *state sweep* operation checks the LATR states from all cores, taking advantage of hardware prefetching since the states are allocated as contiguous memory blocks. Using the active flag and the CPU bitmask available in a LATR state, a core identifies states relevant to itself and invalidates the TLB entries on the core. In addition, after the TLB invalidation, the core removes itself from the CPU bitmask of the respective state. During the state sweep operation, each core updates the CPU bitmask and the active flag using an atomic operation that eliminates the need for locks. For example, in Figure 4.3, CPU_2 and CPU_5 invalidate their local TLB entries during the state sweep operation before resetting the CPU bitmask in the state (❷ and ❸). In addition, CPU_5, the last core performing the local TLB invalidation, resets the active flag in the LATR state (❸). By means of this lazy asynchronous shootdown, LATR inherently provides batched TLB invalidation without using IPIs. For example, similar to Linux where the entire TLB is flushed if there are more than 33 TLB invalidations (*i.e.*, half the size of the L1 D-TLB), LATR flushes the entire TLB during state sweep.

The LATR shootdown is performed during the scheduler tick or a context switch, whichever event happens first. The scheduler tick or context switch event provides an existing transition mechanism in the current OS, which we leverage for the state sweep and TLB invalidation. The LATR TLB invalidation during a scheduler tick or a context switch

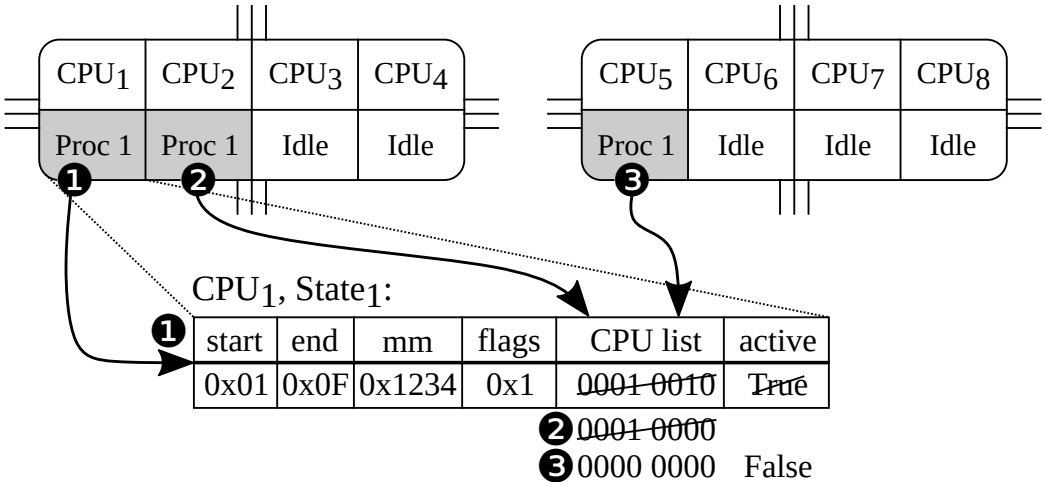


Figure 4.3: Example of the usage of a LATR state. CPU_1 unmaps a page (❶) which is also present in CPU_2 and CPU_5. At the scheduler tick, all other CPUs will use the LATR state to determine if a local TLB invalidation is needed and CPU_2 (❷) and CPU_5 (❸) will invalidate their local TLB entry before resetting the LATR state to be reclaimed.

eliminates the interrupt handling overheads associated with IPIs. In addition, it reduces the cache pollution overhead resulting from IPI interrupts (see Table 4.4).

4.3.2 Handling Free Operations

In this section, we analyze the handling of free operations using the LATR states.

Lazy memory reclamation. A key part in the design of LATR, as outlined in §4.2, is the lazy reclamation of both virtual and physical pages. LATR establishes the following *invariant*: during free operations, virtual or physical pages are released only after associated TLB entries have been cleared. In LATR, to honor this invariant, we explore the following relaxation: By allowing a delay (*e.g.*, 2 ms, twice the scheduler tick interval as introduced in §4.2) before reclaiming virtual and physical pages, we can remove both the reclamation of memory and the need for a TLB shootdown from the critical path of free operations such as `munmap()` and `madvise()`.

LATR deletes the mappings from the page table entry (PTE) during free operations; however, instead of freeing the virtual and physical pages, LATR maintains the list of virtual and physical pages to be freed lazily in the `mm_struct`. In addition, LATR maintains a global list of `mm_structs`, synchronized using a global spin lock, to identify the tasks participating

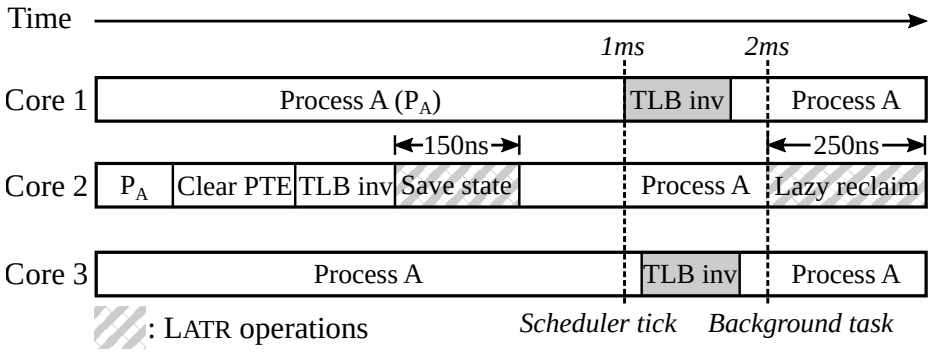


Figure 4.4: An overview of the operations involved in unmapping a page in LATR. LATR removes the instantaneous TLB shootdown from the critical path by executing it asynchronously.

in a lazy reclamation. To ensure that the virtual address is not reused, the lazy virtual address list is traversed during any memory allocation, and the addresses in the lazy list are not reused. Similarly, since the physical page reference count is non-zero, LATR ensures that the physical pages are not reused.

Handling `munmap()`. Using the *lazy memory reclamation* and the LATR states, LATR removes the instantaneous shootdown from the critical path of free operations. Instead, on execution of these operations, LATR simply records the states to shootdown a set of virtual addresses (the *state* information) but does not send an IPI immediately, as outlined in §4.3.1. In the case where there are more shootdowns per interval than there are per-core LATR states (*i.e.*, 64 LATR states per core), LATR issues IPIs as a fallback mechanism. Furthermore, LATR's lazy free operation introduces new race conditions that LATR solves, these conditions are discussed in §4.3.4.

A detailed example of LATR handling an `munmap()` operation is shown in Figure 4.4 as a timeline. Core 2 executes the `munmap()` system call resulting in a TLB invalidation followed by core 2 saving the LATR state which includes the cores 1 and 3 in the CPU bitmask. The `munmap()` execution adds the page to the lazy free list. Due to the CPU bitmask, core 1 and core 3 invalidate their local TLB entry during their respective scheduler ticks (after 1 ms) and reset their respective CPU bitmask in the LATR state. Core 2 runs the LATR background thread (after 2 ms), and frees the virtual and physical pages in the lazy list.

Lazy TLB shootdown correctness. The correctness of LATR's handling of TLB shoot-

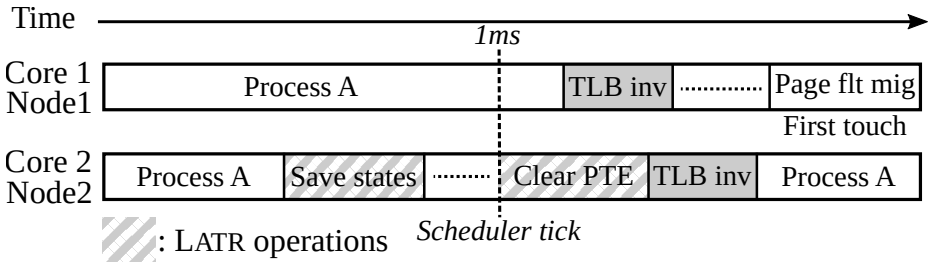


Figure 4.5: AutoNUMA page migration in LATR. LATR removes the need for an immediate TLB shootdown when sampling pages for NUMA migration.

downs for free operations relies on the invariant introduced in §4.2: Virtual and physical pages can only be reused *after* associated TLB entries have been invalidated. To fulfill this invariant, LATR waits two full cycles of TLB invalidations (*i.e.*, two scheduler ticks and 2 ms) to ensure that all associated entries have definitely been invalidated by at least one scheduler tick.

4.3.3 Handling NUMA Migration

In addition to supporting free operations, LATR's design also provides a lazy mechanism for migration operations, such as AutoNUMA page migration. We discuss the LATR mechanism for AutoNUMA page migration in this section.

The current AutoNUMA design in Linux includes a remote TLB shootdown (see §2.2), which accounts for up to 5.8–21.1% (page count ranging from 1 to 512) of the overall time in the case of a page migration. However, this TLB shootdown cost is paid even if the page fault handling decides to not migrate the page. LATR's mechanism for AutoNUMA provides a lazy TLB shootdown approach, eliminating the expensive TLB shootdown operation.

Lazy page table change. For AutoNUMA, a key part of LATR design is the lazy page table change after an interval (1 ms). LATR maintains an invariant that the pages are migrated, after the interval, only after all cores performed a TLB shootdown. LATR uses the state abstraction to maintain this invariant and to perform the lazy page table change.

AutoNUMA mechanism. We illustrate the approach taken with LATR in Figure 4.5, which exemplifies LATR's key design change (shown with two cores on two different sockets):

When the AutoNUMA daemon decides to unmap a page from the page table to test for a potential migration, LATR records only this state into a LATR *state* instead of unmapping the page immediately, delaying the TLB shootdown. This state simply informs all cores to invalidate their local TLB for the specified page at the next scheduler tick. Any memory access before the scheduler tick can proceed without interruption. In addition to invalidating the local TLB entry, the first core performs the page table unmap operation (shown as "Clear PTE") before invalidating its TLB entry. The page unmap operation results in a page fault when the page is next touched, resulting in a potential page migration, similar to the existing design in Linux.

LATR removes the need for an instantaneous and costly IPI while retaining the design of AutoNUMA. As LATR delays the page table unmap operation until the next scheduler tick (1 ms), there is no additional overhead imposed on the services. This design trades off the expensive IPI-based TLB shootdown for additional waiting time until the next scheduler tick (up to 1 ms). Furthermore, LATR's lazy migration introduces new race conditions; their handling in LATR is discussed in section §4.3.4.

Figure 4.5 shows an example of the LATR mechanism used in conjunction with the AutoNUMA page migration. The AutoNUMA background daemon on core 2 adds a state to the LATR states instead of unmapping the page from the page table. This state includes the CPU bitmask of all the cores, including core 1 and core 2. The scheduler tick on core 2 initiates the unmap operation (shown as "Clear PTE") and then invalidates its local TLB. The scheduler tick on core 1 only invalidates its local TLB. The next page access on core 1 triggers the page fault handler and subsequently the page migration due to a page access from a different NUMA node.

Correctness for AutoNUMA migration. We show the correctness of LATR's AutoNUMA migration. Memory accesses during the interval (1 ms) proceed normally. The first core performing the TLB shootdown, after the interval, will unmap the page from the page table. Memory accesses after the interval thus result in a page fault. If the page fault handler

migrates pages, LATR holding a lock until all cores perform their local TLB invalidation ensures that parallel writes are not allowed during the migration.

4.3.4 LATR Race Conditions

In this section, we discuss the possible race conditions introduced by a lazy TLB shootdown and their handling with LATR.

Reads before a TLB shootdown. For free operations, an service error can result in reading already freed memory before the scheduler tick (1 ms). On cores where the respective TLB entry is not invalidated yet, LATR serves the read from the old, not yet freed page. However, after the LATR TLB shootdown during the scheduler tick, any further reads will result in a page fault, which eventually results in a segmentation fault.

Writes before a TLB shootdown. For free operations, an service error can result when writing values to the unmapped memory before the scheduler tick (1 ms). On cores where the TLB entry is not invalidated yet, LATR allows writes to the old page that is not yet freed. However, after the scheduler tick interval, any further writes will result in a page fault, which eventually results in a segmentation fault.

For both reads and writes, LATR does not prohibit services to read or write to an unmapped page for a specific interval (until the scheduler tick, up to 1 ms) although this service behavior is the result of an service error. However, LATR prevents the consequences (*e.g.*, page corruption) of these reads or writes to impact other processes or the kernel by not releasing the physical pages before the LATR TLB shootdown is complete.

AutoNUMA balancing. With LATR's delayed page table unmap operation for the case of AutoNUMA, there is a possibility that a page fault could occur on any core before the page table unmap operation is complete, for example if a page fault occurs simultaneously with the first core unmapping the page from the page table. However, both the page fault and the AutoNUMA page table unmap operation are guarded by the `mmap_sem` semaphore, which ensures that the unmap operation is completed before the page fault handler can

proceed. Similarly, the page fault is handled only after all cores have performed the LATR TLB invalidation for the AutoNUMA migration; otherwise, cores that have not invalidated their TLB entries yet could proceed in writing to the page under migration. To avoid such a race condition, the first core that performs the page table unmap releases the `mmap_sem` only after all CPU bitmasks in the LATR state are cleared, indicating that all cores have invalidated their TLB entries. Thus, the next page fault can then trigger the NUMA page migration.

4.3.5 Approach With and Without PCID

Process-context identifiers (PCIDs) are available in x86 to allow a processor to cache TLB entries for multiple processes and to preserve it across context switches. LATR's lazy invalidation approach is applicable regardless of the OS' use of PCIDs. When PCIDs are not used (as Linux 4.10 elects to do), invalidating TLB entries pertaining to the states during the scheduler tick is important to remove stale entries within a bounded time period (*e.g.*, 1 ms). During a context switch, however, the TLB is flushed, eliminating the need for a LATR invalidation. In the case where PCIDs are being used, only TLB entries matching the currently active PCID can be invalidated. Since the invalidation should be performed before the PCID is changed, LATR's TLB invalidation during a context switch is mandatory. When using PCIDs, TLB invalidations can be triggered only for the currently active PCID. When a different PCID is active, LATR aborts migrations similar to a case where AutoNUMA aborts a page migration if the page fault does not indicate a remote socket accessing the page.

4.3.6 Large NUMA Machines

LATR eliminates the use of expensive IPIs to disseminate the TLB shootdown information. Instead, LATR requires writing the LATR states to memory, which implicitly propagate to the LLCs of all sockets via the hardware cache coherence protocol, making LATR highly scalable. The lazy approach employed by LATR eliminates the synchronous TLB shootdown

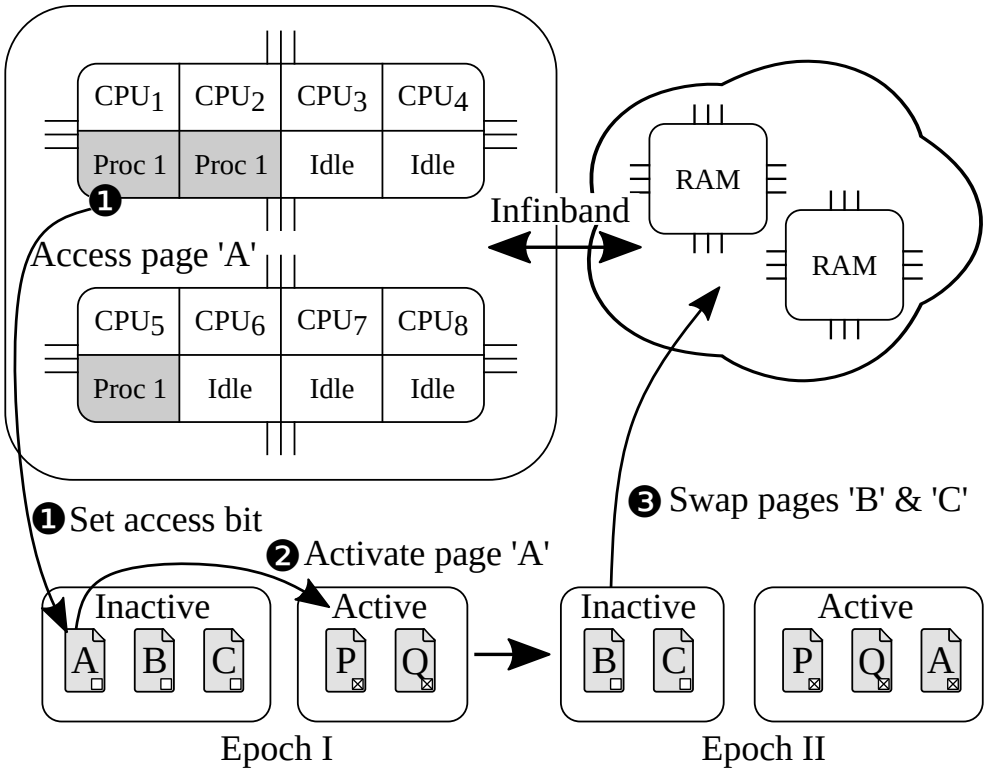


Figure 4.6: INFINISWAP with support for lazy swapping with LATR. Access bits are used to move pages on access (❶) to the *active* list (❷). After the LATR epoch is finished and all TLB invalidations have taken place, the pages are swapped out to remote memory (❸).

overhead, which amounts to up to 80 μs on an 8-socket machine (as shown in Figure 4.9).

4.3.7 Handling page swapping

In addition to supporting free operations with PCIDs, LATR's design provides a lazy mechanism for page swap operations. We discuss the LATR mechanism for page swapping in this section. The current page swap design in Linux includes a remote TLB shootdown (see Figure 2.3), which accounts for up to 18% of the overall time in the case of a page swap using INFINISWAP. LATR's mechanism for page swapping provides a lazy TLB shootdown approach, eliminating the expensive TLB shootdown operation (see Figure 4.7).

Lazy page swapping. The key part of LATR's design for background page swapping (*e.g.*, `kswapd` in Linux), is to swap pages lazily after LATR's epoch of 1 ms, after the inactive pages' TLB entries are invalidated. LATR maintains an invariant that the pages are swapped out, after an LATR epoch, only if their TLB entries are not present in any of the TLBs. The

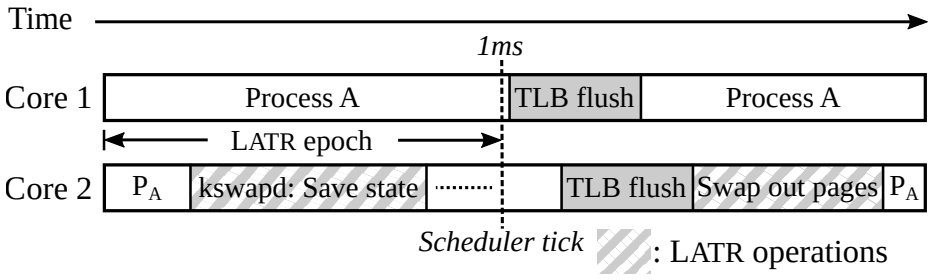


Figure 4.7: Page swapping in LATR. LATR removes the need for an immediate TLB flush after pages are swapped out.

pages accessed during the epoch, whose TLB entries are present in any of the TLBs, are not swapped out and added to the active list.

LATR extends LATR's states to record the TLB shootdown states needed by a page swap. The page swap background task instead of swapping out inactive pages immediately, adds the state to the LATR states. Due to the large inactive list, the added state indicates a full TLB flush on all CPU cores, similar to the existing page swap mechanism (handled using IPIs). After an LATR epoch, when all cores have fully flushed their TLBs, the inactive pages are swapped out to remote memory.

To swap out pages to remote memory after an epoch, LATR has to ensure that the TLB entries of inactive pages are not present in any cores. However, some cores could access the page in phase two, which would set up the TLB entry again. LATR tracks such inactive page accesses using the access bit in the PTE. For tracking accesses to inactive pages, the background task, in addition to adding the states, resets the access bit in the PTE for all the pages in the inactive list. After an LATR epoch inactive pages that have their access bit set, indicating that those pages were accessed during phase 2, are moved to the active list while other pages are (potentially) swapped out.

Page swap policy change. LATR's lazy mechanism delays page swapping by an LATR epoch, providing an additional epoch to track page accesses. By tracking accesses during an epoch, LATR changes the existing page swapping policy by not swapping out pages accessed during an LATR epoch. Using its lazy swapping policy, LATR improves, in addition to removing the TLB shootdown overhead, the temporal locality of pages accessed during

Table 4.2: The two machine configurations used to evaluate LATR.

Machine Type	Commodity data center [115]	Large NUMA
Model	E5-2630 v3	E7-8870 v2
Frequency	2.40 GHz	2.30 GHz
# Cores	16	120
Layout (cores × sockets)	8 × 2	15 × 8
RAM	128 GB	768 GB
LLC (size × sockets)	20 MB× 2	30 MB× 8
L1 D-TLB entries (per core)	64	64
L2 TLB entries (per core)	1024	512
Hyperthreading	Disabled	Disabled

an epoch by not swapping them out.

Correctness for page swapping. We show the correctness of LATR's page swapping operation during the three phases. Page accesses for inactive pages during phase one and two proceed as before, with the hardware setting the access bit when needed. In phase two, any page access to an inactive page sets the access bit, as the TLB is flushed and the access bit is reset when setting up the LATR state. Correctness for page swapping is thus maintained by not swapping out any pages with the access bit set, maintaining the invariant.

One potential race condition is in phase three: when pages are being swapped out, the access bit could be set in parallel with the page being accessed, *e.g.*, one core is performing swapping while another core accesses the page resulting in the access bit set. LATR avoids this race condition by checking the access bit immediately after resetting the present flag in the PTE. If the access bit is set, the page is moved back to the active list and is not swapped out.

4.4 Implementation

We implemented the LATR prototype in 1,012 lines of code by extending Linux 4.10. We modified the kernel's TLB shootdown handler to save the LATR states instead of sending IPIs. We extend the kernel's `munmap()` and `madvise()` handlers to perform the lazy memory

reclamation, and the AutoNUMA page table unmap handler to save the LATR states.

Lazy TLB shootdown. LATR's lazy TLB shootdown is handled in `native_flush_tlb_others`, in which sending of IPIs is replaced with saving the corresponding LATR states. The state sweep to invalidate the LATR states is handled in `scheduler_tick` and `__schedule`.

Lazy memory reclamation. The `VMA` and `page` pointers are added to a lazy list in the `mm_struct`, and the `mm_struct` is added to a global list. The background kernel thread frees the `VMA` and `page` using `remove_vma` and `free_pages`, respectively.

NUMA page migration. The function `task_numa_work` is modified to not trigger the page table unmap via `change_prot_numa`. Instead, a handler saving the LATR state is invoked while, `change_prot_numa` is invoked later during the `scheduler_tick`, along with the local TLB invalidation.

4.5 Evaluation

We implemented a proof-of-concept of LATR based on Linux 4.10. The baseline for the evaluation is Linux 4.10, while we also compare a subset of cases against ABIS [49], which is based on Linux 4.5. ABIS is a recent research prototype that aims at reducing the number of IPIs sent by tracking the set of cores sharing a page via the page table access bits [49].

Using this setup, we evaluate LATR by answering the following questions:

- Does LATR show benefits with microbenchmarks on machines with a larger number of NUMA sockets?
- What are the benefits of LATR for data center services with heavy usage of free operations (introduced in Table 4.1)?
- What are the benefits of LATR in the context of AutoNUMA page migration?
- What is the impact of LATR for services which show few TLB shootdowns and what is the overhead of LATR for memory usage and cache misses?

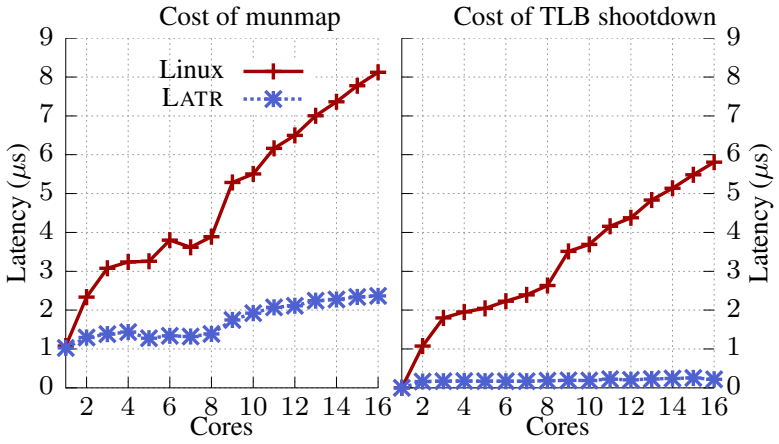


Figure 4.8: The cost of an `munmap()` call for a single page with 1 to 16 cores in our microbenchmark. TLB shootdowns account for up to 71.6% of the total time. LATR is able to improve the time taken for `munmap()` by up to 70.8% with its asynchronous mechanism.

4.5.1 Experiment Setup

We evaluate LATR on two different machine setups, as shown in Table 4.2. The primary evaluation target is the *2-socket, 16-core* machine, while we also show the impact of LATR on a large NUMA machine with *8 sockets and 120 cores* . We run each benchmark five times and report the average results.

The machines are configured without support for transparent huge pages, as this mechanism is known to increase overheads and introduce additional variance to the benchmark results [132]. Furthermore, to reduce variance in the results, all benchmarks are run on the physical cores only, without hyperthreads. We furthermore deactivate Linux's automatic balancing of memory pages between NUMA nodes, AutoNUMA, unless specifically noted, as it might introduces TLB shootdowns during the migration of a page (see §2.2).

4.5.2 Impact on Free Operations

First, we discuss the impact of LATR on operations centered around freeing virtual and physical addresses, such as `munmap()` and `madvise()` (as introduced in Table 4.1).

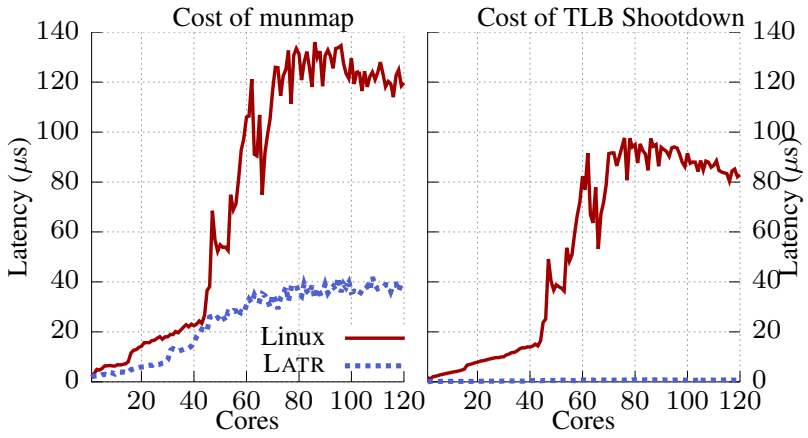


Figure 4.9: The cost of `munmap()` along with the cost for the TLB shootdown for a single page in Linux compared to LATR on an 8-socket, 120-core machine. TLB shootdowns account for up to 69.3% of the overall cost, while LATR is able to improve the cost of `munmap()` by up to 66.7%.

Microbenchmarks

To understand the scalability behavior of LATR in isolation, we compare LATR to Linux while we exclude ABIS [49], as its behavior matches Linux in a microbenchmark where all cores actually access a shared page. We devise a microbenchmark that shares a set of pages between a specified number of cores. A subsequent call to `munmap()` on this set of pages will then force a TLB shootdown on the participating cores. Each data point is run 250,000 times. The microbenchmark records the time taken for the call to `munmap()`, as well as the time taken for the TLB shootdown, excluding other overheads, *e.g.*, the page table modifications and syscall overheads.

The results of this microbenchmark using one page on our 2-socket, 16-core machine are shown in Figure 4.8 and exemplify the overheads introduced by the TLB shootdown: In the baseline Linux system, the TLB shootdowns contribute up to 71.6% to the overall execution time of `munmap()`, while a single `munmap()` call for 16 cores takes up to 8 μs. LATR on the other hand is able to reduce almost all of this overhead and improves the latency of `munmap()` by 70.8% by recording only the LATR states on the critical path of `munmap()`. LATR thus reduces the latency for `munmap()` to 2.4 μs for 16 cores.

Large NUMA machine. To investigate the behavior of LATR and the baseline Linux on a large NUMA machine, we run this microbenchmark on the 8-socket, 120-core machine and

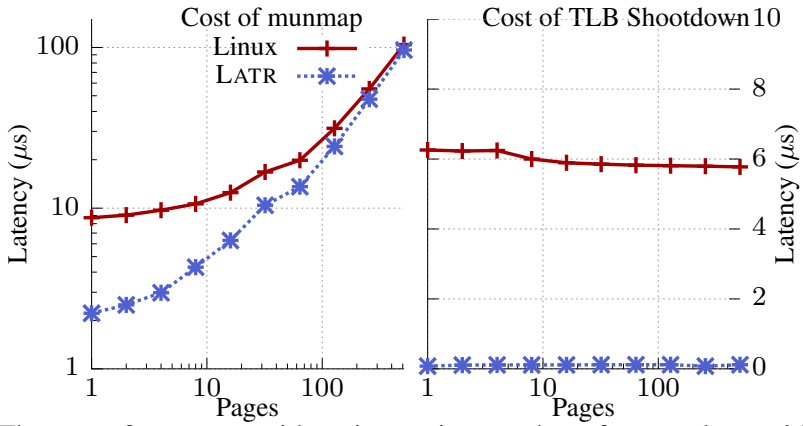


Figure 4.10: The cost of `munmap()` with an increasing number of pages along with the cost of the TLB shootdowns for Linux and LATR. For a small number of pages, LATR shows improvements of up to 70.8%, while the impact of the TLB shootdown diminishes with a larger number of pages. At 512 pages, LATR still retains a 7.5% benefit over Linux.

show the results in Figure 4.9. These results show a drastic increase in latency for `munmap()` on Linux when using more than 45 cores (more than 3 sockets), as the IPI delivered through the APIC needs two hops to reach the destination CPU. At 120 cores, the latency for a single `munmap()` rises to more than 120 μs, with the TLB shootdown accounting for up to 82 μs or 69.3%. LATR on the other hand is able to efficiently use the cache coherence protocol to complete the `munmap()` operation in less than 40 μs on 120 cores, reducing the latency by 66.7% compared to Linux, as LATR's `munmap()` does not rely on expensive IPIs and eliminates the `ACK` wait time on the initiating core.

Increasing number of pages. We investigate the behavior of LATR compared to Linux when using more than one page; the results for up to 512 pages on 16 cores are shown in Figure 4.10. The impact of the TLB shootdown diminishes with a larger number of pages, as the overhead of clearing TLB entries is amortized by more costly operations, such as changing the page table. Furthermore, Linux elects to fully flush the TLB when more than 32 pages are being invalidated at once, which furthermore limits the maximum possible overhead. Even though the impact of the TLB shootdown reduces, LATR still improves the performance at 512 pages by 7.5% while showing larger benefits with fewer pages. Furthermore, services can use *huge pages* (either 2 MB or 1 GB pages on x86), either directly or via transparent huge pages support in the OS [132], to mitigate the effects of

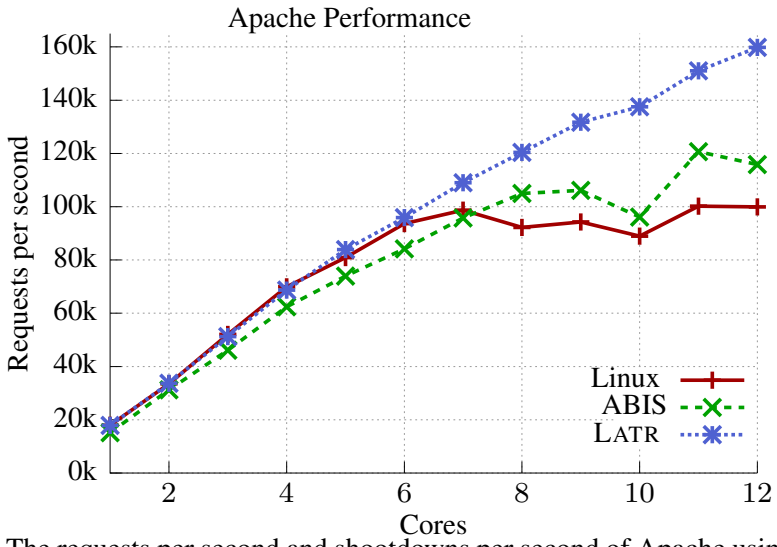


Figure 4.11: The requests per second and shootdowns per second of Apache using LATR, Linux and ABIS [49]. LATR shows a similar performance as Linux for lower core counts while outperforming it by 59.9% on 12 cores. ABIS initially shows overhead from frequent unmapping, while LATR performs up to 37.9% better at 12 cores than it.

unmapping many pages at once.

Impact on data center services

We evaluate LATR by quantifying its impact on free operations with real-world services, using both Apache and the PARSEC [133] benchmark suite.

Apache webserver. We compare the requests per second of Apache with Linux, ABIS [49], and LATR on the 2-socket, 16-core machine. We use the Wrk [117] HTTP request generator, using four threads with 400 connections each for 30 seconds, to send requests to Apache, which hosts a static, 10 KB webpage. Wrk and Apache run on the same machine (to avoid the network stack becoming the bottleneck) but on a distinct set of cores to isolate the two services. This configuration leaves up to 12 cores available for Apache. We disable logging in Apache and use the default `mpm_event` module to process requests. This module spawns a (small) set of processes that in turn spawn many threads to handle the requests. To serve an individual request, Apache `mmap()`s the requested file to serve a request and `munmap()`s the file after the request has been served. This behavior generates many TLB shootdowns due to the frequent unmapping of (potentially) shared pages. The results are shown in Figure 4.11

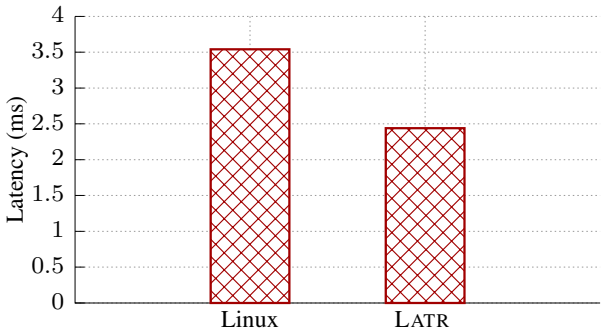


Figure 4.12: Comparison of Apache's latency with LATR and Linux using wrk benchmark. LATR improves the latency of Apache by up to 26.1%.

and show both the requests per second served by Apache, as well as the TLB shootdowns per second. LATR outperforms Linux by up to 59.9% and ABIS by up to 37.9%. ABIS shows a reduced performance on lower core counts (¡ 8 cores) because of the overhead of frequent changes to access bits while outperforming Linux for larger core counts because of the significantly reduced number of TLB shootdowns. LATR outperforms both Linux and ABIS because of its efficient asynchronous handling of TLB shootdowns, even though the rate of shootdowns is up to 46% higher due to the increased performance of LATR. Furthermore, LATR does not need to use any fallback IPIs (see §4.3.2) during the execution of this test.

Latency of Apache. In addition to throughput, we evaluate apache latency with LATR and Linux using the same benchmark used for throughput. LATR improves apache latency by 26.1% compared to Linux (as shown in Figure 4.12). By optimizing the synchronous TLB shootdown, LATR reduces the CPU overhead on all the cores which enables the reduction in latency. Apache latency evaluation shows the latency improvement by optmizing system services such as virtual memory.

Parsec benchmark. We show the performance of LATR compared with Linux across a wider range of PARSEC benchmarks. The normalized runtime (with respect to Linux) along with the TLB shootdowns per second is shown in Figure 4.13. LATR shows improvements of up to 9.6% on cases with a larger number of TLB shootdowns (*e.g.*, `dedup`) due to frequent calls to `madvise()` and shows small improvements for most of the other benchmarks. The

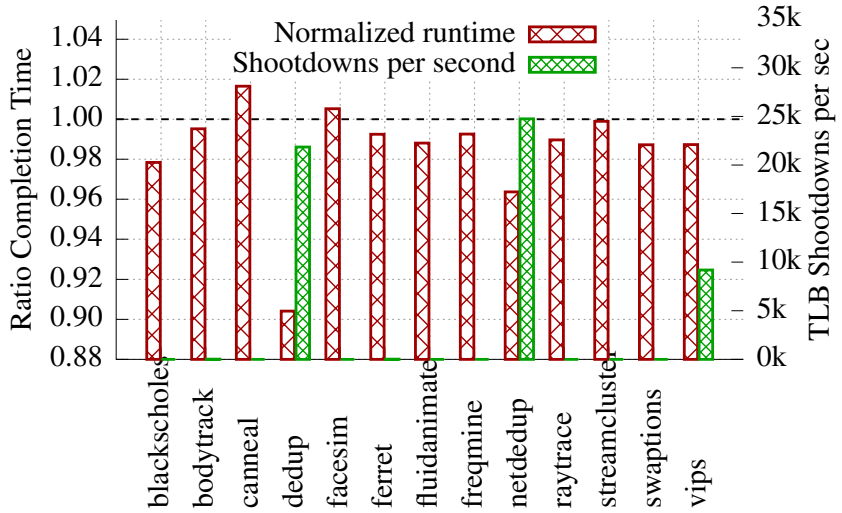


Figure 4.13: The normalized runtime and the rate of shootdowns for the PARSEC benchmark suite, comparing LATR and the Linux baseline using all 16 cores. LATR imposes at most a 1.7% percent overhead while improving the runtime on average by 1.5% and by up to 9.6% for `dedup`.

reason for the small improvement for most benchmarks is that LATR optimizes background operations of the system such as reading files via `mmap()`/`munmap()`. For one benchmark, `canneal`, LATR shows a slight degradation of 1.7% due to frequent context switches for this benchmark, which triggers frequent state sweeps. Overall, LATR shows an improvement of 1.5% on average over Linux across all PARSEC benchmarks.

4.5.3 Impact on NUMA Migration

In contrast to previous benchmarks, we enable AutoNUMA for the following experiments. We evaluate the impact of LATR on AutoNUMA with a subset of services (`fluidanimate` from PARSEC and `ocean_cp` from the SPLASH-2x benchmark suite) that benefit from enabling NUMA memory balancing. Additionally, we evaluate LATR with three real-world services: Graph500 [134], PBZIP2 [135], and Metis [136, 56, 137]. Graph500 is a graph analytics workload running a breadth-first search on a problem of size 20. PBZIP2 allows compressing files in parallel and in memory. We compress the Linux 4.10 tarball while splitting the input file among all processors. Finally, Metis is a Map-Reduce framework optimized for a single-machine, multi-core setup.

The results are given in Figure 4.14 and show the normalized runtime of LATR compared

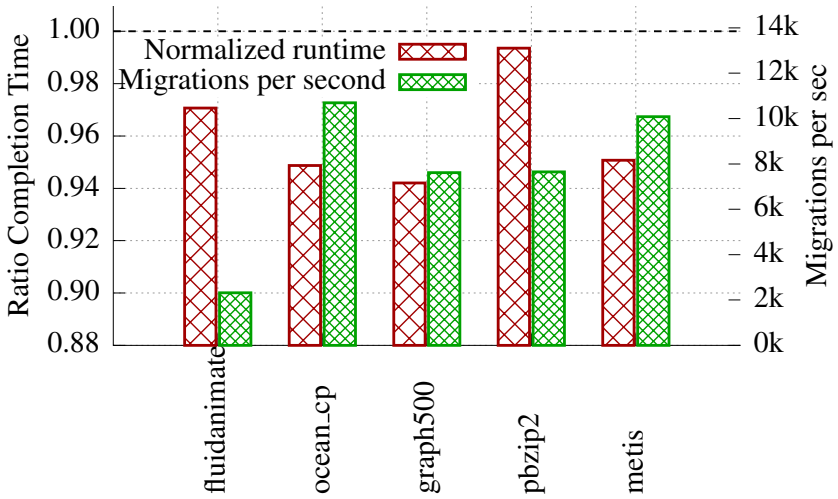


Figure 4.14: Impact of NUMA balancing on the overall runtime as well as the overall number of page migrations of LATR compared to Linux on 16 cores. LATR performs up to 5.7% better, showing a larger improvement with more page migrations.

to Linux, as well as the number of page migrations per second. The results show an improvement of up to 5.7% for the Graph500 benchmarks while showing similar benefits on other benchmarks that show a large number of migrations. On PBZIP2, LATR improves only marginally compared with Linux, as for this service other overheads dominate the runtime. As AutoNUMA migrates one page at a time, this result aligns with the cost breakdown of a migration operation showing a 5.8% (one 4 KB page) to 21.1% (512 4 KB pages) overhead for the TLB shootdown.

4.5.4 Impact on Page Swapping

We evaluate LATR's benefits with page swapping using INFINISWAP [15] that uses remote memory as the swap device. We constrain the service inside an `lxc` container and set the soft-memory limit of the container to about half of the services working set to induce swapping via `kswapd`. Overall, the TLB shootdown accounts for up to 20% of the swapping time with INFINISWAP when running Memcached with 5M keys using a recently published workload (`ETC`) by Facebook [118]. This workload shows around 100,000 TLB shootdowns per second from background swapping, as the kernel has to ensure that dirty pages are unmapped and not being written to on all cores before swapping them out.

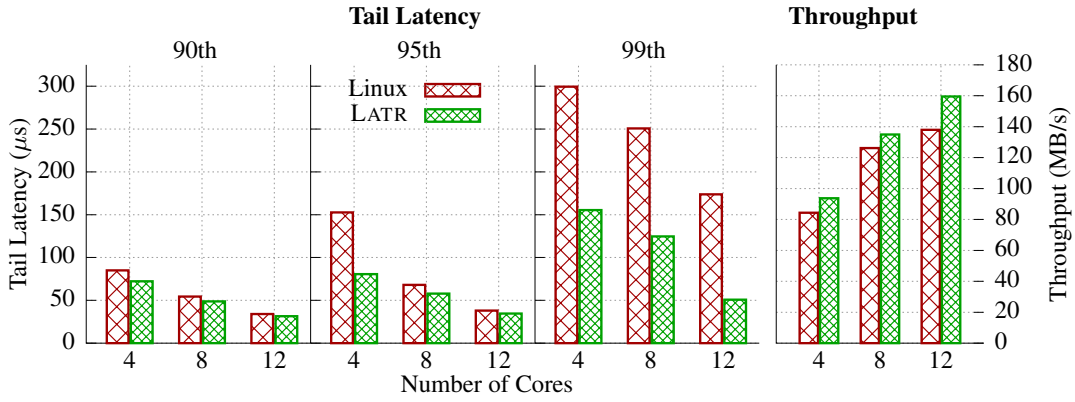


Figure 4.15: The impact of swapping using INFINISWAP with Linux and LATR for Memcached in terms of both latency and throughput for a varying number of cores. LATR improves the 99th percentile tail latency by up to 70.8% and the throughput by up to 13.5% by reducing the impact of synchronous TLB shootdowns.

Memcached. We use the Mutilate tool [138] to send requests to Memcached [139], constraining Memcached and Mutilate to separate NUMA nodes to minimize interference effects. We show the results with a differing number of cores when running Memcached with INFINISWAP, on Linux and LATR, for both tail latency and throughput in Figure 4.15. We show the tail latency for the 90th, 95th and 99th percentile, using 4, 8, and 12 cores. LATR shows benefits for all of these percentiles, with a larger benefit being visible for higher percentiles (up to a reduction of 14.8%, 47.2%, and 70.8% for the 90th, 95th, and 99th percentile, respectively). LATR helps in reducing the tail latency for in-memory caches, which is a critical goal to many data-center services [2]. LATR also shows a larger benefit when Memcached is run on less cores (*e.g.*, on four cores) as Memcached doesn't scale well when increasing the number of cores [140], thus allowing for more idle CPUs when using more cores which in turn results in a lower tail latency. LATR is also able to improve the throughput of Memcached by up to 13.5% and 9.8% on average across all three configurations tested.

LATR's deferred TLB shootdown algorithm allows pages to move from the *inactive* list to the *active* list (with the help of the active bits in the page table entry) during the epoch before the TLB shootdown is completed on all cores (*e.g.*, 1 ms). For example, this *swap policy change* allows LATR to move around 180 pages per second from the *inactive* to

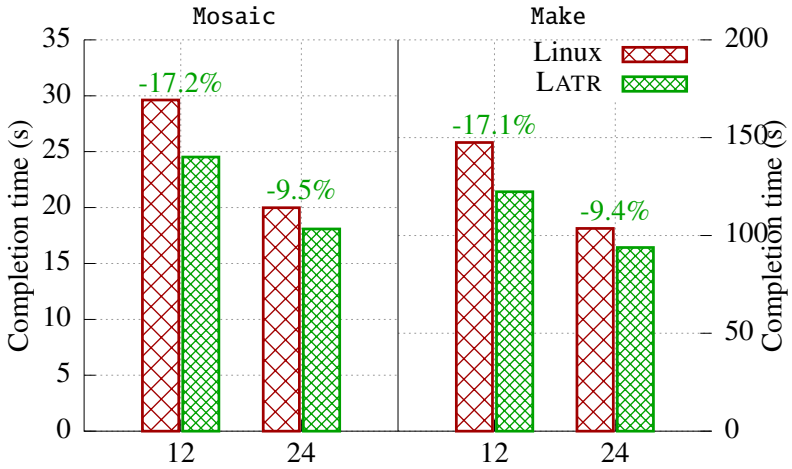


Figure 4.16: The impact of swapping using INFINISWAP with Linux and LATR for `Mosaic` and `Make`. LATR is able to improve the service's completion time by up to 17.2% as a result of the lazy swapping approach.

the *active* list when running Memcached on 12 cores, thus saving the overhead of having to swap out a page that actually would have been used soon after and would need to be swapped in again.

`Make` and `Mosaic`. We demonstrate LATR's benefits when swapping with INFINISWAP with two more services, building a Linux kernel with `Make` and running a graph processing engine with `Mosaic`, focusing on the services's completion time. In more detail, `Make` compiles the Linux 4.10 kernel on a specified number of cores in a massively parallel fashion, loading the source files into memory in the process. On the other hand, `Mosaic` [141] is a graph processing engine, running in an in-memory mode, executing the pagerank algorithm algorithm on the twitter graph [142]. `Mosaic` initially loads the graph into memory before executing 10 iterations of the pagerank algorithm. We report the overall time taken for all 10 iterations.

For both services, we run two configuration for both Linux and LATR, using 12 and 24 cores. The results are given in Figure 4.16 and show that LATR achieves a speedup of up to 17.2% for `Make` and 17.1% for `Mosaic`. LATR shows a smaller benefit (of 9.5% and 9.4%, respectively) for the 24 core configuration as the system has to use all physical and virtual cores for that configuration which results in only marginal speedup, thereby reducing the impact of LATR's improved handling of swapping via INFINISWAP.

4.5.5 Overheads of LATR

We investigate the overheads imposed by LATR in three aspects: what is the memory overhead of LATR, how does LATR impact services with few TLB shootdowns, and how does LATR impact the cache locality of service?

Memory utilization. We perform an analysis based on the microbenchmarks presented to show LATR's overhead in terms of memory utilization. In each time period (*e.g.*, 1 ms), LATR shows an overhead of up to 21 MB of physical and virtual pages (for the case of 16 cores and 512 pages per `munmap()` call). If fewer cores and pages are being used, the overhead ranges from 3 MB (for 2 cores sharing a single page) to 1.5 MB (for 16 cores sharing a single page). Using more pages, the memory overhead stays bounded by 21 MB, as the overhead of page table modifications and related operations dominates the cost of the TLB shootdown (as seen in Figure 4.10). Considering the large virtual address space (2^{48} bytes) and the amount of RAM (64 GB and more) available in current servers, the memory overhead is not high (smaller than 0.03%) and is released back within a short time interval (2 ms).

Overhead on services. We show the overhead imposed by LATR on real-world services with few TLB shootdowns in Figure 4.17. This focuses on services being run only on a single core (two webservers: Nginx and Apache) and a subset of PARSEC benchmarks which show very few TLB shootdowns. LATR shows a maximum overhead of 1.7% due to a larger number of context switches while imposing a smaller overhead on other services. LATR is even able to improve the performance of some of the benchmarks due to optimizing various background activity in the system, as well as the general benefit of faster unmapping and freeing of shared memory.

An interesting analysis is the percentage of LLC cache misses for various services and core counts. These results are shown in Table 4.4 and show that LATR *improves* cache misses for a number of services while only imposing a maximum overhead of 0.8% on

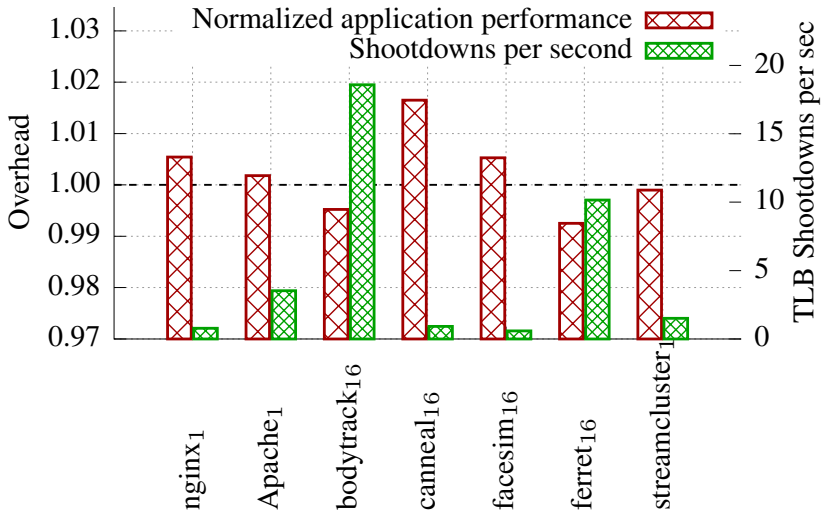


Figure 4.17: The overhead imposed by LATR on services with few TLB shootdowns; subscripts indicate the number of cores. LATR shows small overheads of up to 1.7% for one benchmark.

Table 4.3: A breakdown of operations in LATR compared to Linux when running the Apache benchmark. LATR reduces the time taken for a single shootdown by up to 81.8% due to its asynchronous mechanism.

Operation	Time spent
Saving a LATR state	132.3 ns
Performing single state sweep with LATR	158.0 ns
Single TLB shootdown in Linux	1594.2 ns

other cases. LATR's improvements in cache misses are due to the removed handling of IPI interrupts on remote cores. These benefits outweigh the increased cache utilization of the LATR states, which, however, occupy only a small portion of the LLC of modern processors (less than 1%, see §4.3.1).

Breakdown of operations. We show a breakdown of operations in LATR compared to Linux when running Apache on 12 cores in Table 4.3. This breakdown shows the two separate phases of LATR: saving the LATR states on a per-core basis as well as invalidating local TLB entries based on the other's LATR states. This breakdown only includes the time taken for the TLB shootdown, excluding other effects such as modifying the page table or the syscall overhead, allowing for a fair comparison between Linux and LATR. Overall, LATR reduces the time taken for the TLB shootdown by up to 81.8%.

Table 4.4: The ratio of L3 cache misses between Linux and LATR; subscript indicates the number of cores the benchmark ran on. LATR shows cache misses to be very close (or better) to the Linux baseline due to the minimal cache footprint of LATR's states.

Application	Cache Misses		Relative Change
	Linux	LATR	
Apache_1	6.08%	6.13%	+0.84%
Apache_6	1.60%	1.55%	-3.27%
Apache_{12}	1.23%	1.22%	-1.32%
canneal_{16}	80.51%	79.94%	-0.71%
dedup_{16}	18.33%	18.14%	-1.09%
ferret_{16}	48.02%	48.21%	+0.38%
$\text{streamcluster}_{16}$	95.42%	95.25%	-0.18%
swaptions_{16}	47.48%	47.23%	-0.54%

4.6 Chapter Summary

We presented LATR, a lazy software-based TLB shootdown mechanism that is readily implementable in modern OSes, for significant operations such as *free* and *page migration*, without requiring hardware changes. In addition, the proposed lazy migration approach can play a critical role in emerging heterogeneous memory systems and in disaggregated data centers. LATR reduces the cost of `munmap()` by up to 70.8% on multi-socket machines while improving the performance of services like Apache by up to 59.9% compared to Linux and 37.9% compared to ABIS. In addition to synchronous operations in operating systems, such operations impede service performance in distributed systems where a lazy mechanism (eventual consistency) is not applicable. Next, we present DYAD that untangles the logically coupled consensus mechanism from the service.

CHAPTER 5

DYAD: UNTANGLING LOGICALLY COUPLED CONSENSUS

5.1 Introduction

Consensus algorithms [74, 76, 78, 18] are commonly required in distributed services such as lock managers [16, 17], key-value stores, and timestamp servers [81]. To guarantee high availability, consensus algorithms perform state machine replication on multiple replicas. Consensus algorithms, such as leader-based Paxos [74], Viewstamped replication (VR) [77], or Raft [18], are used by these distributed services.

Since services optimize their execution to reduce latency and need to classify client requests to orchestrate consensus for specific operations, existing consensus algorithms tend to be entangled with the services. For example, services such as Etcd and Cassandra need to classify the client operations after decoding the packet and then orchestrate the PREPARE message (see Table 5.1). In addition, Zookeeper [17] speculatively checks the quorum logic for faster execution of operations, reducing latency, which entangles consensus with service logic. Unfortunately, such a design practice of entangling the service and consensus logic incurs two problems. First, it results in an unpredictable, conflicting resource allocation between an service and a consensus algorithm, in which the consensus algorithms tends to dominate the resource usage with an increasing number of replicas or when it is contended (see, indirect cost in §2.1), second, it results in a weak fault domain in which the consensus states are lost when a service fails.

In this work, we attempt to address these problems by untangling the service and consensus logic via *delegation*. Delegation, in this context, means that the consensus logic is separated from the service logic, and the entire consensus part is executed on a separate processing element. Unlike offloading techniques that run the entire service with consensus

Table 5.1: Services need to optimize latency and orchestrate consensus for specific operations after decoding the packet, which entangles the service with the consensus algorithm.

App.	Algo.	Consensus operations intertwined in app.
Zookeeper	ZAB	The application, to reduce latency, speculatively verifies the quorum logic and the sequence number for faster execution of operations (`create`, `delete`, and `setData`)
Cassandra	Paxos	The storage proxy needs to classify the client operations (`insert` and `delete`), after decoding the packet, and then orchestrates the PREPARE † message
Etcd	Raft	The application needs to classify put operations, after decoding the packet, then the put handler orchestrates the PREPARE † message

† : PREPARE shown in Figure 5.1. Paxos and Raft: PREPARE ⇔ PROPOSE

on special-purpose hardware, this approach delegates part of the service—consensus—to a separate entity. In addition, delegation isolates the consensus states, eliminating the weak isolation problem. In normal cases, the delegated consensus logic handles the high-overhead coordination needed with replicas before delivering the client requests to the service in consensus order. The service ensures *linearizability* by executing the client requests in the order received from the delegated consensus logic. With delegation, service-level coordination is only needed for events such as recovery and view change.

Existing research approaches can be classified into two categories. The first category attempts offloading to FPGAs [92], which is limited in two aspects: first, the service is also running with the consensus algorithm in an FPGA, and second, the service is limited by the hardware resources available on an FPGA. In addition, such an attempt to offload does not eliminate the two problems incurred by a logically-coupled consensus (*i.e.*, unpredictable resource allocation and a weak fault domain is not eliminated). The second category of approaches, such as NoPaxos [81], Speculative Paxos [82], and NetPaxos [143, 85], attempts partial delegation by pushing certain functionality to the network. However, because of partial delegation, the network-based approaches impose strict ordering guarantees on the network to decouple consensus. In addition, the approaches that rely on the programmable switch suffer from high latency and low throughput because of the ordering constraints

needed in the switches [143, 85, 92, 86].

The key idea of DYAD is to address the problems incurred as a result of logically-coupled consensus by delegation of consensus algorithms, which are naturally integrated as part of network operations, using SmartNICs as a target platform. Current SmartNICs provide hardware filtering, which enables delegation by filtering certain services' operations, *e.g.*, the write operations in Zookeeper, Etcd, and Cassandra are filtered and delivered to the delegated consensus logic on the SmartNIC. By delegating the consensus algorithm on the SmartNIC, the consensus coordination with replicas is handled closer to the network I/O, eliminating the PCIe and CPU [79, 80] overheads (direct and indirect cost §2.1). DYAD eliminates the unpredictable resource-allocation problem by processing only the client requests on the host processor. In addition, DYAD eliminates the weak fault domain by storing the ordered logs and the consensus states on the SmartNIC (currently up to 32 GB [7] supported). DYAD ensures *linearizability* [144] by processing the client requests, on the host processor, in the order delivered by the SmartNIC.

DYAD improves service recovery for consensus algorithms, such as VR, by using the ordered logs available on the SmartNIC. In addition, DYAD provides an adaptive fault-detection mechanism that identifies service failures and supports service parallelism by appending a logical timestamp to the client requests. DYAD's evaluation with increasing replicas shows that it improves the throughput and tail latency of service, such as timestamp server and key-value store, by up to 8.2x and 90%, respectively. In addition, DYAD reduces the CPU usage by up to 62% by processing only the client request on the host processors. By delegation, DYAD improves service recovery by up to 67% using the logs on the SmartNIC and demonstrates ease-of-use with real-world service such as Memcached.

In DYAD, we make the following contributions:

- DYAD provides delegation of consensus, eliminating the unpredictable resource-allocation and weak fault domain problems in a logically-coupled consensus.
- We demonstrate delegation using SmartNICs by leveraging the hardware packet

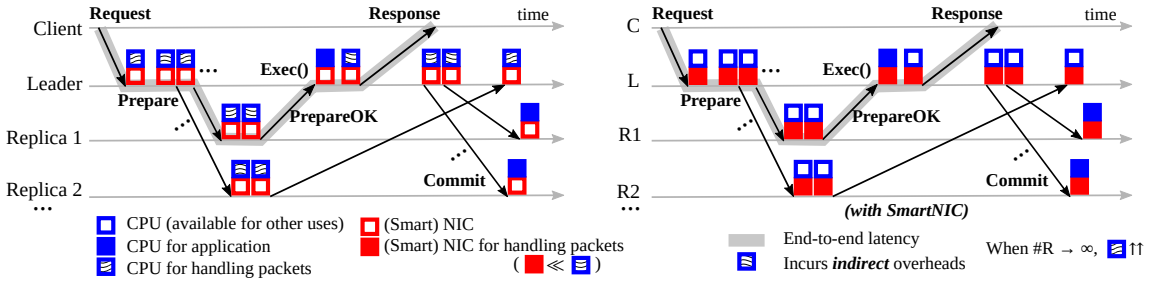


Figure 5.1: Normal-case execution of VR with and without Dyad. The existing VR algorithm incurs high direct and indirect costs, which increases with increasing replicas. Dyad reduces the direct and indirect costs by untangling VR, which is executed on the SmartNIC.

filtering and custom packet handling which is supported on commodity SmartNICs.

- We evaluate DYAD on a cluster with three, five, and seven replicas using services such as timestamp server and key-value stores.

5.2 Design

5.2.1 DYAD Overview

DYAD provides a software architecture which leverages the emerging SmartNICs to build leader based consensus algorithm such as Viewstamp Replication (VR) protocol that uses replicated state to provide persistence. By leveraging the SmartNICs, DYAD physically isolates the consensus operation performed on the SmartNIC and the services running on the host processor(). To provide the physical isolation, DYAD defines the abstractions provided by the SmartNIC and the operations performed by the on the host processor (shown in Figure 5.1). Other than leveraging the SmartNIC, DYAD does not impose any additional ordering constraints on the network. We assume an asynchronous network, where there are no guarantees that packets will be received in a timely manner, in any particular order, or even delivered at all. As a result, the consensus algorithm which is running on the SmartNIC is responsible for both ordering and reliability. Figure 5.2 shows the system architecture with the host processor containing the service and the SmartNIC containing the consensus component.

The SmartNIC provides an ordered client request abstraction to the service running on

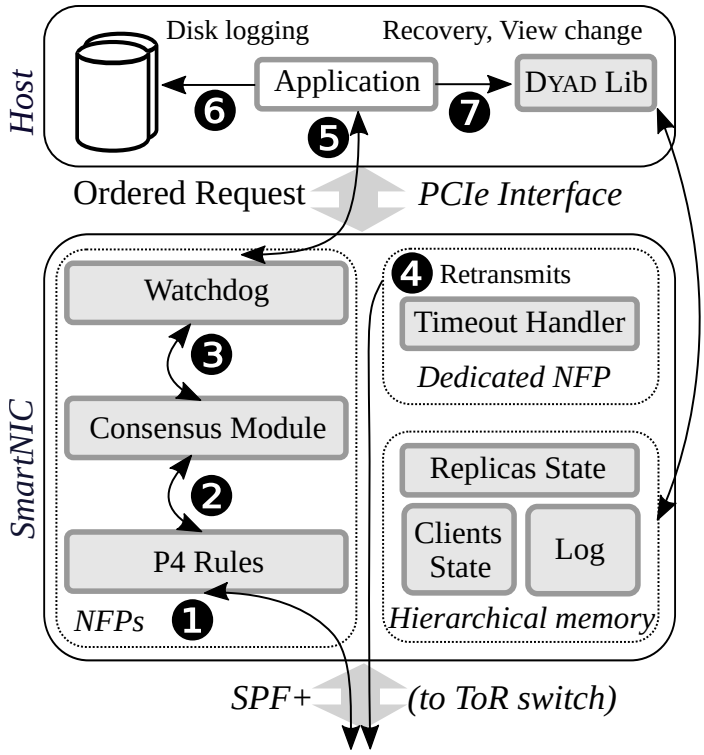


Figure 5.2: Architecture overview of DYAD. Packets are classified by the P4 rules (❶) providing them to the consensus module (❷) that coordinates with the replicas. The watchdog (❸) monitors the host RTT for each packet and the timeout handler processes retransmits (❹). The application (❺) runs on the host processors executing the ordered requests. In addition, the application performs disk logging (❻) for protocols such as Raft, and uses DYAD library (❼) for recovery and view change. The DYAD library reads the ordered logs in SmartNIC memory over the PCIe interface.

the host(§5.2.2). With such an abstraction, the client requests are serialized on the SmartNIC and delivered in the log order to the host processor after executing the consensus algorithm. The host processor executes the requests, either using a single or multiple thread, in the order received from the SmartNIC which ensures *linearizability*. For multi-threaded services, the SmartNIC appends *logical timestamp* to the client requests which is used by the service to ensure *linearizability*(§5.2.3). In addition to processing client requests, the ordered client request abstraction enables the service running on the host processor to recover after a software failure(§5.2.5). However, to provide physical isolation, fault tolerance is an important property needed in DYAD. *i.e.*, the service failure on the host processor should be detected by the SmartNIC, and vice versa(§5.2.4). For service failures, DYAD provides agnostic fault tolerance by using the round-trip time (RTT) measurements on the SmartNIC. And for SmartNIC failures, DYAD relies on the heart-beat messages that are part of the

consensus protocol. We explain the hardware capabilities available on the SmartNICs that enable DYAD's abstractions.

Hardware packet filtering. DYAD leverages the hardware packet filtering mechanism available in the SmartNICs to filter client and consensus messages. Existing commodity SmartNICs, with FPGAs or network processors, support L7 packet decoding and filtering mechanism by providing P4 language support. In addition to packet decoding and filtering, SmartNICs support custom *application handlers* that executes specific action (process) on a received packet. In addition to processing a received packet, the application handlers can either drop a packet or DMA a packet to the host processor or forward a packet back to the network. DYAD implements custom actions for the client requests and consensus messages to handle a consensus protocol on the SmartNIC. DYAD classifies the client messages and the various consensus messages by using the *Message type* in the DYAD packet format.

SmartNIC processing. The software on the SmartNIC is the core component of DYAD which is responsible for providing the ordered logs and executing the consensus algorithm. SmartNICs expose one (or more) virtual interface to the host, which is treated as a network interface by the operating system for sending and receiving packets. DYAD leverages the hardware filtering mechanism available in SmartNICs to filter and process the packets delivered to or received from a virtual interface. The packets destined to the host processors are processed on the SmartNIC before delivering the packets to the host processor, and similarly the packets leaving the host processors are processed by the SmartNIC before delivering the packet to the network. Only the filtered packets are processed on the processing elements available on the SmartNIC, while the other packets are treated with standard NIC functionality.

5.2.2 Ordering client requests

The SmartNIC maintains the ordered logs to execute the consensus algorithm on the SmartNIC. The ordered logs is a linear memory on the SmartNIC where the requests

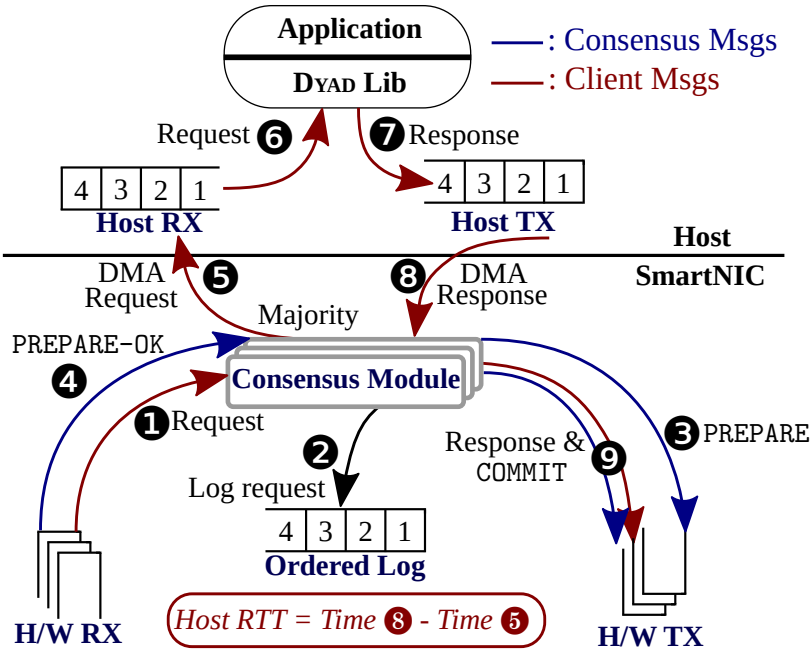


Figure 5.3: Packet flow of consensus messages and client messages in the leader node with DYAD. The client requests (❶) are ordered and logged on the SmartNIC (❷) before executing the VR algorithm (❸ ,❹). After the majority of replicas respond (❺), the client request is forwarded to the host for application processing (❻). The response message from the host (❼ ,❽) is used to send the COMMIT messages to the replicas (❾). With three replicas, for handling one client message, the NIC processes eight messages (request + response + 2 PREPARE + 2 PREPAREOK + 2 COMMIT).

from clients are serialized using an atomically increasing sequence number. The ordered log is used to deliver the client request in the logged order, and are stored on the DRAM available on the SmartNICs which support upto 32 GB of DRAM capacity. With such a DRAM capacity, millions of log entires with 1 KB size can be stored. In addition, the ordered log is available on the SmartNIC after an service failure on the host processor which enables service recovery. Figure 5.3 shows the packet flow on the leader.

Ordering on leader. The SmartNIC on the leader node filters the client requests and logs them on the smartnic. Instead of immediately sending the requests to the service running on the host processor, the consensus PREPARE request are sent to all the replicas. After majority of the replicas respond with the PREPAREOK message for an entry in the head of the log, the logged client request is sent to the host processor. The PREPARE messages received from the other replicas (after the majority) are dropped by the SmartNIC.

Consensus messages can arrive out-of-order from the network. i.e., the PREPAREOK

message from the majority of the replicas can arrive for a log entry beyond the first log entry. In such a scenario, DYAD ensures delivering the client requests in log order by forwarding the requests to the host processor only when the majority of replicas respond to the first log entry. In addition to sending the first log entry to the host, DYAD sends further log entries for which consensus is reached. This ensures that the SmartNIC only delivers packets in the log order to the host even if the network reorders the packet.

In addition to reordering, the network is not reliable which means that the consensus messages can be dropped. The dropped messages could result in a condition where the first log-entry does not receive a PREPAREOK message from the majority of replicas. DYAD relies on timers and back-logs to reinitiate the consensus messages for the log entries that did not receive the PREPAREOK messages. Though some NICs that use network processors do not support timers, they can be overcome by using dedicated network processing that trigger retransmisson at periodic intervals. In addition to timers, retransmissions are triggered when the log entires are back-logged beyond a particular threshold. Both the above mechanisms ensures that DYAD provides best-effort delivery when the underlying network in unreliable.

The response message sent by the service is filtered and processed on the SmartNIC. The response message is processed on the SmartNIC for two reason: the first reason is to send the COMMIT message to all the replicas and the second reason is to measure the RTT which is used for fault tolerance. After the SmartNIC processing, the response message is forwarded to the client over the network. As an optimization, the SmartNIC identifies in-progress consensus operations and piggybacks the sequence number on the PREPARE message instead of sending a COMMIT message which reduces the number of messages processed on the replicas.

Ordering on replicas. In addition to maintaining the ordered logs on the leader SmartNIC, the ordered logs abstraction is useful on the replicas other than the leader. In the replicas, the PREPARE message from the leader SmartNIC is decoded , filtered, and append to the log on the SmartNIC. The PREPARE message is handled and dropped on the SmartNIC

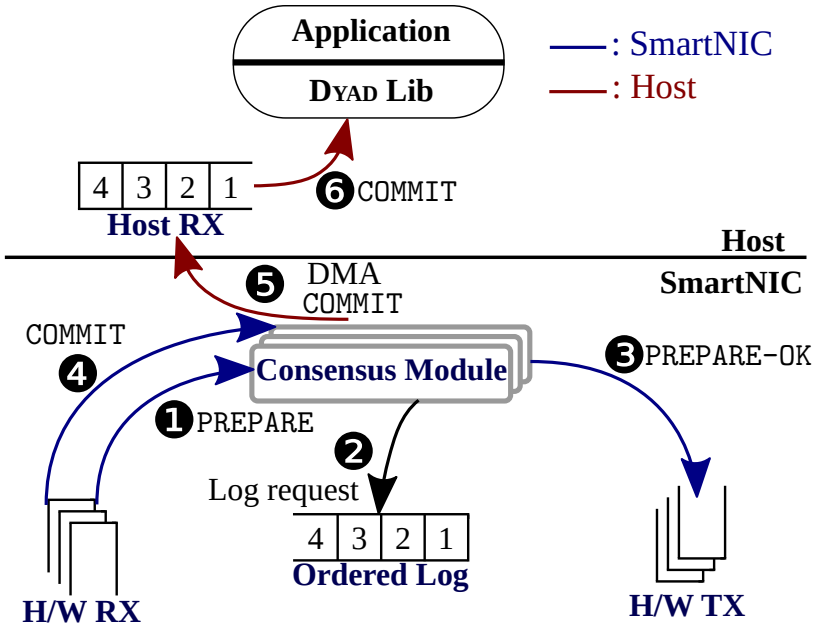


Figure 5.4: Packet flow of consensus messages in the replica with DYAD. The PREPARE messages (❶) received from the leader are logged in the SmartNIC memory (❷) using the sequence number received, and the PREPAREOK message (❸) is sent to the leader. The COMMIT received (❹) from the leader is appended with the request and forwarded to the application running on the host (❺ ,❻).

without forwarding it to the host. In addition to appending the log, the replica SmartNIC responds with the PREPAREOK message to the leader. The COMMIT message received from the leader is forwarded to the host along with the corresponding logged data which results in committing the operation on the replica host. Similar to the ordered client request delivery on the leader, the COMMIT messages are forwarded to the host processor in the same log order. Request (client messages on the leader and PREPARE messages on the replicas) delivered to the host in log order ensures *linearizability*. Figure 5.4 shows the packet flow on the leader.

5.2.3 Service parallelism

A single threaded service, running on the host, is linearizable by processing the requests in the order they are received. However, the same deterministic execution order can not be maintained if the service is multi-threaded, i.e., a multi-threaded service can not ensure that the execution order is same as the arrival order. Such a multi-threaded service does not

ensure *linearizability* when the requests are delivered in log order. Multi-threaded service require deterministic execution to provide linearizability, which is provided by DYAD using logical timestamps.

A logical timestamp is the *sequence number* which is used to identify a request in the ordered log on the SmartNIC, which defines the total order for the client requests. DYAD's SmartNIC component appends a logical timestamp to client requests that require consensus to support multi-threaded service. For example, the read operation in a key-value store requires consensus for which the logical timestamp is appended by the SmartNIC, and the logical timestamp is not appended to operations such as write. A multi-threaded service running on the host is linearizable by executing the client requests in the logical timestamp order. The appended logical timestamp acts as an interface between the SmartNIC and the host to support multi-threaded services.

DYAD services running on the host use the logical timestamp to ensure total order for the client commands that require consensus. The logical timestamp is similar to a ticket used in a ticket lock where the locks are serviced in the order of the ticket issued. However, unlike a ticket lock, the logical timestamp (ticket) is issued by the SmartNIC. The commands that does not require consensus, such as the read operations, could be placed on a separate queue in the host avoiding head-of-line blocking due to linearizable operations. Though the command execution is serialized, other operations such as receiving the request and sending the response are handled in parallel.

5.2.4 Fault tolerance

DYAD enables fault tolerance by using the ordered logs available on the SmartNIC. In addition, DYAD provide failure detection mechanisms to identify services and SmartNIC failures.

Log persistence. The ordered log maintained by the SmartNIC consensus component is available after a service failure. In Netronome SmartNICs, the lifetime of the ordered log

is till an power failure or till the firmware on the SmartNIC is reloaded. In SmartNICs, such as Mellanox Bluefield, that run a separate operating system (Linux) on the SmartNIC, the SmartNIC operating system is not affected by the system reboot. In such SmartNICs, the logs persist a service failure and system reboot, and are only lost upon a power failure. Consensus algorithms, such as VR, fetch the logs from other replicas during a power failure, which is a fall back mechanism in DYAD. In addition,DYAD enables disk logging if needed by the protocol.

Failure detection. To provide physical isolation of the consensus algorithm and the service, DYAD identifies service failures on the SmartNIC and SmartNIC failures on the host. Identifying such failures allow the service and the SmartNIC to recover the failed subsystem. In case of an service failure, the SmartNIC stops executing the consensus algorithm and the service recovery is initiated using the logs on the SmartNIC and the replicas. In case of the SmartNIC failure, the SmartNIC should be recovered depending on the SmartNIC used. *e.g.*, In Netronome SmartNICs, the firmware is reloaded to instantiate the SmartNIC subsystem. The service running on the host processor sends the heart-beat message to all the replicas which is only needed when there are no client requests to be handled.

Service failures. Identifying an service failure on the SmartNIC is important, otherwise the SmartNIC would continue to execute the consensus algorithm and forward the ordered client requests to the host processor where the service is not running. The other replicas will not identify the leader service failure because the consensus messages are received from the leader node which identifies the healthy status of the leader node. However, though the SmartNIC on the leader node is healthy, the service on the host may not be running.

DYAD measures the RTT for handling each client request from the host processor. i.e., an RTT constitutes the time from sending the client request to the host to the time the client response is received on the SmartNIC. DYAD calculates the weighted moving average of the RTT, similar to the TCP RTT calculation, and uses this metric to detect service failures. Equation 5.1 shows the RTT calculation where α value can be between zero and one, and

DYAD used 0.8 which is a recommended value for TCP RTT calculation. When a client request does not receive a response for a threshold that is a function of the RTT, then the SmartNIC identifies the service failure. The threshold should be decided considering the occasional outlier that could falsely identify a service failure. The different multiplier values and their corresponding false positive value is further discussed in the evaluation. When there are no client requests handled by the system, DYAD identifies service failures using the heart beats sent/received by the service. Since the service on the replicas do not respond to COMMIT messages, service failure on the replicas is detected using the heart-beat messages.

$$Host_{RTT} = \alpha * Host_{RTT} + (1 - \alpha) * current_{RTT} \tag{5.1}$$

SmartNIC failures. The client requests received on the host identifies the health status of the SmartNIC. In addition, When there are no client request, the heart-beat messages from the other replicas is used to identify the health status of the SmartNIC. The above fault tolerance mechanism is applicable on both the leader and the replicas.

5.2.5 Recovery

Service recovery for consensus protocol such as VR involves fetching the replicated logs from the other replicas, after the service is restored from a checkpoint. DYAD handles the state-transfer messages that retrieve the replicated logs from the replicas on the SmartNIC. After the state-transfer messages are decoded and filtered on the SmartNIC, the logs available on the SmartNIC are used to respond to the state-transfer message. In addition to handling the state-transfer message, the logs on the SmartNIC provide a first-level recovery mechanism before fetching the states from the other replicas. The first-level recovery mechanism is applicable for recovery resulting from software failure which is the major cause of failures in data center services. In case of hardware or power failure resulting in the entire system failure, DYAD falls back to recovering the logs from other replicas.

Replica recovery. DYAD proposes a two stage recovery mechanism to recover the service

Table 5.2: Recovery and view change interface provided by DYAD

Function	Description
`clientRequest(request)`	Provide client request format at compile time
`clientResponse(response)`	Provide client response format at compile time
`getNicLog(data, length, dstport)`	Retrieves the ordered log from the SmartNIC
`updateReplica(replicas, dstport)`	Updates the replica's state on the SmartNIC

.

on the replicas. In the first stage, the service fetches the logs from the respective SmartNIC over the PCIe interface. And, in the second stage, it fetches the remaining logs from the other replicas in the network which is the recovery mechanism proposed by VR protocol. Without the first stage, all the logs should be retrieved over the network from all the replicas increasing the network bandwidth needed for recovery.

Leader recovery. Similar two stage recovery mechanism is used to recover the service on the leader node. The logs on the leader SmartNIC identify the last committed log entry for which a response is sent to the client. By recovering the last committed operation, DYAD ensures the "execute-once semantics" after crash recovery [145]. Without the SmartNIC logs, the last committed entry which sent a response to the client is not trackable because the leader node could crash after sending the client response and before sending the COMMIT messages.

Recovery interface. DYAD provides a recovery API that fetches the logs from the SmartNIC over the PCIe interface(as shown in Table 5.2). As part of the API, the port number used by the service is used to multiplex the logs used by various services. With the current interface provided by NIC vendors such as Netronome, the PCIe read throughput is 16 MB/s with a single thread. However, DYAD increases the PCIe read throughput by sharding the log and reading the logs with multiple threads. With 16 threads, the PCIe read throughput is increased to 256 MB/s. The PCIe read throughput for the logs is discussed further in the evaluation with various log size and threads.

5.2.6 View change and leader election

View change and leader election are handled by the service running on the host. DYAD considers these operations as control operations which change the consensus state on all the replicas. During the view change and leader election operation, the SmartNICs subsystem stops processing messages, and forwards the messages between the host and the network. After a view change and leader election, the service running the host performs a service checkpoint and updates the replica details on the SmartNIC.

DYAD provides `UpdateReplica` API to update the replica details to the SmartNIC Table 5.2. Similar to the log recovery interface, DYAD multiplexes the replica details using the service port number.

5.2.7 Supporting reliable connection

unlike the VR protocols, there are certain consensus algorithms designed with the assumption of the reliable network transportation layer. One of such examples is the Raft consensus protocol: it relies on TCP to communicate with the replicas and to log the client commands to storage for persistence. DYAD SmartNIC component has a basic TCP protocol that allows the Raft leader to communicate with the replicas. DYAD TCP stack specifically decodes Raft headers and payload. And since packet reordering is infrequent in data centers, DYAD processes only in-order packets and relies on TCP retransmission on the replicas [146]. There are existing approaches proposed in the Linux kernel community to process the out-of-order packets on the CPU core instead of the SmartNICs, which is applicable to DYAD TCP stack. In addition, the service running on the host processor logs the requests to disk before executing the command and sending the response. DYAD demonstrates that the SmartNIC design allows disk logging if needed by the protocol. Logging to disk, for persistance, can be eliminated when the SmartNICs support non-volatile memory in future. Running the basic TCP stack reduces the latency of Raft consensus drastically which in turn reduces the end-to-end latency. In addition, with Raft, the network packet drops and

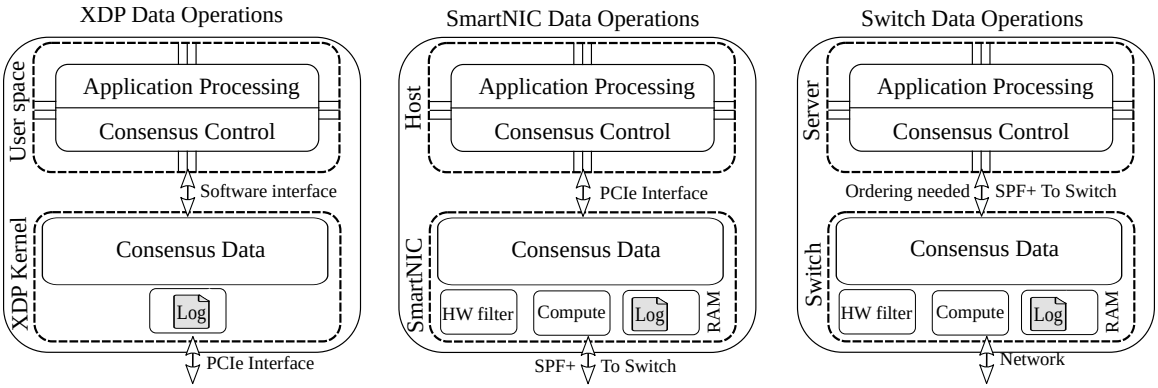


Figure 5.5: The consensus data operations could be executed on three different infrastructure, XDP, SmartNICs, and programmable switches.

reordering are taken care by the TCP protocol.

5.3 Dyad applicability - infrastructures

In this section, we discuss the applicability of DYAD's design to other infrastructures such as express data path, SmartNICs with RDMA, and programmable switches. Figure 5.5 shows the pictorial representation of consensus data operation that is executed on various infrastructures. Figure 5.6 shows the three phases of consensus where ordering and replication is untangled. With such a untangled algorithm, programmable switches need coordination with the service running on the host to provide ordered execution. In addition, we discuss the direct and indirect cost of consensus in these infrastructures. Table 5.3 shows the way DYAD's components fit in different infrastructures. In addition, the table shows the protocol used between the replicas and the need for ordering in these infrastructures. Table 5.4 shows the direct and indirect cost of consensus with various infrastructures.

5.3.1 Express Data Path

Express data path (XDP) is a new mechanism to reduce the protocol processing and context switch overhead in the kernel. XDP allows custom user-space functionality to be executed closer to the NIC drivers which provides the performance benefits needed for consensus consensus algorithms.

Table 5.3: The table shows where the control and data operations are performed and the applicability of DYAD's fault tolerance mechanism on various infrastructures such as XDP, SmartNICs, and Switches. In addition, it shows the protocol possible between the replicas in the respective infrastructure and the need for network ordering in these infrastructures.

Systems	Operations		App Failure		Replica	Network
	Data	Control	Detection	Recovery	**Protocol**	**Ordering**
eXpress Data Path (XDP)	Kernel XDP	Host	App RTT	XDP Logs	UDP/TCP	Not Needed
SmartNIC	SmartNIC	Host	App RTT	SmartNIC Logs	UDP/TCP	Not Needed
SmartNIC-RDMA	SmartNIC	Host	App RTT	SmartNIC Logs	RDMA	Not Needed
Programmable Switch	Switch	Host	App RTT	Switch Logs	UDP/TCP	Needed

DYAD's data operations can be executed inside XDP, and the control operations can be executed with a library in the service. The ordered log needed for storing the client requests is stored in the XDP maps that can store up to 2 million entries. DYAD with XDP needs failure detection mechanisms to identify service crashes which is possible by tracking the host RTT. The ordered logs in DYAD XDP is isolated from the service which enables the service to recover the logs after failures. `getNicLog` API should be extended to retrieve the logs from XDP instead of the NICs. However, during the system failure, the entire logs are lost and the service recovers the logs from other replicas. `updateReplica` API is needed to update the XDP module with view changes performed as part of the control operations.

We next analyze the direct and indirect cost with XDP. With respect to the direct cost, XDP reduces the context switch and protocol processing cost for consensus operations. However, the PCIe overhead is not reduced by executing consensus with XDP in a normal NIC. Though, XDP is more useful for System on Chips (SoCs), such as Mellanox Bluefield, where the PCIe overhead is eliminated, and XDP reduces the system overhead for consensus messages. With respect to the indirect cost, XDP does not reduce the cache pollution and CPU overhead completely. In summary, XDP is a more flexible packet processing framework for consensus which will improve consensus performance. However, the indirect cost of consensus is not reduced by XDP.

Limitations. The packet parsing and filtering is done in software which adds additional overhead compared to hardware filtering and parsing. In addition, such parsing should be

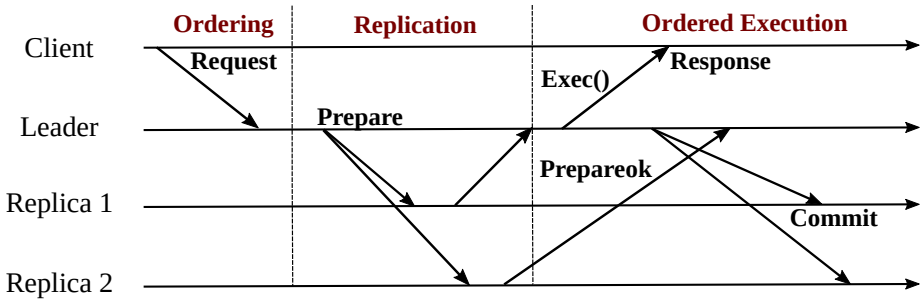


Figure 5.6: The three phases of consensus – ordering, replication, and ordered execution – are shown in this figure. The cost of ordering and replication is reduced by untangling the operations in XDP, SmartNICs, or programmable switches. However, programmable switches need coordination with the service to perform ordered execution.

done for every packet which will induce additional latency for packets which are not filtered.

5.3.2 SmartNICs

In addition to the applicability of DYAD to SmartNICs which we have discussed before this section, we discuss the applicability of DYAD to SmartNICs with RDMA in this section. The main difference is that the SmartNICs with RDMA can update the logs on the remote replicas using RDMA writes which could eliminate the protocol overhead incurred in non-RDMA SmartNICs. In addition, the RDMA SmartNICs need support for hardware packet filtering and processing packets which enable data operations on the SmartNIC. With respect to the SmartNICs, ordering is ensured on the SmartNIC without the need of network ordering.

5.3.3 Programmable Switches

In this section, we discuss the applicability of building leader based consensus mechanisms, such as VR, Raft and Zab, on programmable switches. Programmable switches such as Barefoot Network's Tofino ASIC chip provide data path programming which enable building the DYAD data operations on the switches. DYAD's data operations which perform ordering and replication is possible on the switches. In addition, the service executes on a server which handles the control operations. However, an important restriction with respect to the switches for supporting data operations is that they should ensure ordered delivery of

Table 5.4: The table shows the impact of the direct and indirect cost on various infrastructures, such as XDP, SmartNICs, and Switches, where DYAD's design is applicable. In addition, the table shows network ordering which is needed in programmable switches.

Systems	Direct Cost			Indirect Cost		Network
	PCIe	Control	Protocol	Cache	CPU	Ordering
eXpress Data Path (XDP)	Applicable	Removed	Applicable	Applicable	Applicable	Not needed
SmartNIC	Removed	Removed	Reduced	Removed	Removed	Not Needed
SmartNIC-RDMA	Removed	Removed	Removed	Removed	Removed	Not Needed
Programmable Switch	Removed	Removed	Reduced	Removed	Removed	Needed

these packets from the switch to the server running the service. In Figure 5.6, the ordered execution phases requires coordination between the service and the programmable switches. In case of XDP and SmartNICs, this is ensured in a single node using the software layer and PCIe interface respectively.

Fault detection is needed on the switches to identify service failures on a server which is possible using DYAD's host RTT. However, the response should traverse through the same switch, that runs the data operation, to identify service failures on the switch. Similarly, service recovery is possible using the logs available on the switches. However, unlike DYAD using SmartNICs, the logs need to be retrieved over the network for recovering the service. In addition, the switches could use either TCP or UDP to communicate with the other replicas.

With respect to the direct cost and indirect cost, data operations running on the SmartNIC provides the same benefits as SmartNIC. However, switches add an additional overhead of routing the packets through the leader switch which might not be the optimal route to reach the server.

Limitations. Programmable switches are limited by resources, both memory and compute capabilities, which limit the number of logs stored and the amount of per-flow states [86, 87]. In addition, placing the leader on the switches in data center impose addition requirements such as the leader switch should provide consensus data operations to all the services with the limited compute and memory resource available. In addition, the client requests should be routed to the leader switch before send it to the host which might not be the optimal route.

An important design level requirement is that the requests from the leader switch to the host should be ordered to ensure *linearizability* which was provided by XDP and SmartNICs. In addition, the service running on the host recovers the logs from the leader switch over the network.

With switches, another conceptual limitation is the number of nodes used for replication. Leader based consensus algorithms, such as Raft, Zab, or VR, use (2f + 1) nodes to support f failures. With the switch and the hosts, the number of nodes used increases to 2 * (2f+1) with (2f + 1) switches and (2f + 1) hosts.

5.4 Implementation

DYAD prototype is implemented using Netronome SDK RTE version 6.1.0.1 with around 5000 lines of C (called as Micro-C) and P4 code written for the SmartNIC subsystem. Out of the 5000 lines of code, 1700 lines constitute the leader handling, 1000 constitute the replica handling, 2000 lines constitute the TCP handling, and the remaining constitute the P4 code. In addition, DYAD implements a time-stamp server and key-value store that runs as an service on the host processor, and the library that provides the recovery and view-change interface. DYAD provides an library to the service running on the host which provides the interface for recovering logs from the SmartNIC and updating the replica status on the SmartNIC.

Forwarding requests. The netronome SDK provides APIs to generate packets and to calculate hardware checksum, which is used to generate the consensus messages to the replicas. However, there are no APIs currently available to generate packets to the host processor. This limitation is overcome in DYAD by modifying any message received from the network with the message that should be sent to the host processor and forwarding it to the host processor. DYAD sends the client requests by modifying the prepare-ok messages from the last replica, that constitutes the majority, with the client request that is logged in the SmartNIC, and forwards the modified message to the host processor. The prepare-ok

messages from the other replicas are dropped on the SmartNIC.

Hierarchical memory. The memory on the netronome NIC is hierarchical, consisting of 5 levels, with various size and access times. The closest memory to the processor is 4 and 64 KB with access times of 13 to 15 cycles whereas the farthest memory is available in GBs with access times of 150 to 500 cycles. The decoded packet data, the remaining payload, and the packet metadata are stored in the closest memory, and the ordered log are stored in the farthest memory. The other data such as the replica and client states are store in one of the middle hierarchy. The data structures are explicitly allocated in one of the hierarchical memory at compile time using unique keywords.

Handling messages. The Netronome SmartNIC contains many compute elements called as micro engines (ME) that handle messages in parallel. i.e., the client messages and the consensus messages are processed by multiple MEs in parallel. With such parallel processing, the client requests are ordered by using the atomic operations such as `mem_test_add` and `mem_incr64` that are available on the Netronome SmartNIC. The packet buffers for sending the consensus messages are received from `pkt_ctm_alloc` which allocates the buffers on the nearby memory in the hierarchy, and the messages are transmitted to the outside network using `pkt_nbi_send`. The timers for sending retransmits is implemented using the me_tsc_read() and the sleep() functions, which are executed on dedicated MEs.

5.5 Evaluation

In the evaluation, we answer the following four questions:

- What are DYAD's benefits in terms of tail latency and throughput for services?
- What is the impact of DYAD's performance with parallelism?
- What is DYAD's impact on fault tolerance?
- What are DYAD's benefits in recovering the replicas after software failures?

Experiment setup. We evaluate DYAD using five servers that are connected by a Mellanox SN2700 switch that handles only the traffic generated by the experiments. All clients and

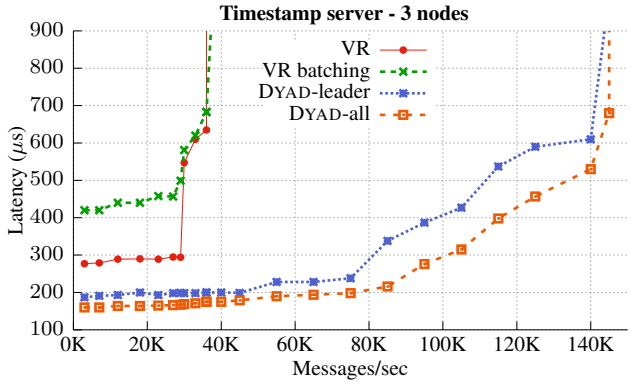


Figure 5.7: 99th percentile latency as a function of throughput for a 3-node testbed deployment of VR protocol running on the CPU vs the SmartNIC for a time-stamp server. DYAD-leader increases the throughput by up to 4x and reduces the latency by up to 68.5% for 36K PPS. In addition, DYAD-all reduces the latency by another 5% for 36K PPS.

replicas ran on the servers with atleast two 2.0 GHz Intel Xeon processors with 16 cores in total and 64 GB of RAM. For replicas that ran the SmartNIC subsystem, the servers are configured with a Netronome Agilio CX 40GbE SmartNIC that has 2GB of RAM. The other replicas and clients, that do not use a SmartNIC, are configured with Mellanox ConnectX-2 (40G) or Intel 82599ES (10G) NIC. All machines ran Ubuntu LTS 16.0.4 with the 4.15.0 Linux kernel. All experiments used three replicas that handled requests from 20 clients unless otherwise stated. DYAD is compared to two other replication protocols that are executed on the host processors: Paxos and Paxos with batching. VR with batching, run with batch size 64 unless stated, shows the overall benefits provided by kernel-bypass mechanisms such as DPDK that rely on batching. DYAD modified the NOPaxos benchmark to use structures as messages instead of using protobuf. For all the experiments, We measured the latency and throughput using a benchmark that sends request for one-minute run of the experiment. All the services are single threaded in a bare-metal setup unless otherwise stated. Henceforth, DYAD leader refers to using the SmartNIC subsystem only on the leader, and DYAD replica refers to using the SmartNIC subsystem on the leader and all the other replicas.

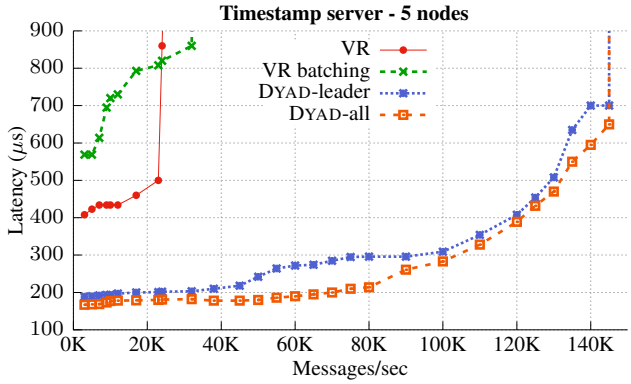


Figure 5.8: 99th percentile latency as a function of throughput for a 5-node testbed deployment of VR protocol running on the CPU vs the SmartNIC for a time-stamp server. DYAD-leader increases the throughput by up to 5.8x and reduces the latency by up to 76% for 24K PPS. In addition, DYAD-all reduces the latency by another 3% for 24K PPS.

5.5.1 Service performance

DYAD shows the end-to-end performance using two services. First, a *timestamp server* that generates monotonically increasing timestamps or globally unique identifiers used for distributed concurrency control. Each replica maintains its own counter which is incremented after consensus with a majority of replicas. Second, a *key-value store* that replicates the put operation on all the replicas before executing the operation on the leader which sends the response to the client. The other replicas execute the put operation after receiving the commit message from the leader.

Timestamp server. We first evaluate the throughput vs latency of DYAD for a timestamp server with multiple replicas (three, five, and seven nodes). With three replicas, the results in Figure 5.7 show that DYAD-leader improves the throughput by up to 4x and the tail latency by up to 68.5%. DYAD-leader SmartNIC subsystem handles sending and receiving the consensus messages which frees up the host CPU to handle only the client requests which is the reason for DYAD's performance improvement. Particularly, the SmartNIC subsystem handles 1 Million messages from the clients and the replicas, which eliminates such overhead on the CPU. DYAD-all further reduces the latency of the timestamp server by handling the prepare messages on the replica SmartNIC. Though the reduction in latency is 5% for lower packets per second (PPS), the improvement increases by up to 36% (compared

to DYAD-leader) with increased PPS (80K - 115K). With increased PPS on the replicas, the host processing overheads on the replicas increase which is eliminated by DYAD-all. When the PPS increases further (beyond 115K), the service processing overhead on the leader node increased which increases the latency in DYAD-all. In addition, the baseline VR with batching shows that batching increases the latency of DYAD operations without impacting the throughput.

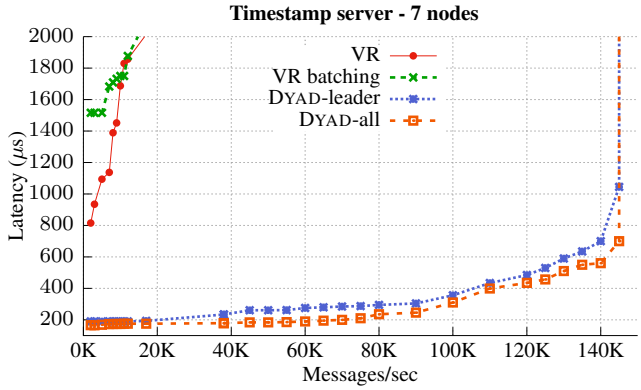


Figure 5.9: 99th percentile latency as a function of throughput for a 7-node testbed deployment of VR protocol running on the CPU vs the SmartNIC for a time-stamp server. DYAD-leader increases the throughput by up to 8.2x and reduces the latency by up to 90% for 17K PPS. In addition, DYAD-all reduces the latency by another 25 μs for 17K PPS

We evaluate the throughput and latency of timestamp server using DYAD with five and seven replicas. The results in Figure 5.8 and Figure 5.9 show that the baseline throughput drops by up to 33% and 52% respectively, which shows the overhead induced to handle messages to/from increased number of replicas. DYAD improves the throughput by 5.8x for a five-node setup and the latency by up to 79% compared to the baseline. And with the seven-node setup, DYAD improves throughput by up to 8.2x and the latency by up to 90% compared to the baseline. With five and seven replicas, the number of messages handled on the SmartNIC is 1.9 and 3 Million respectively. with DYAD-leader and DYAD-all, the increased messages are handled by the SmartNIC which does not increase the end-to-end latency considerably (up to 8 μs). Since the service on the host processor only handles the client requests, the throughput and latency is approximately similar with the various number of replicas.

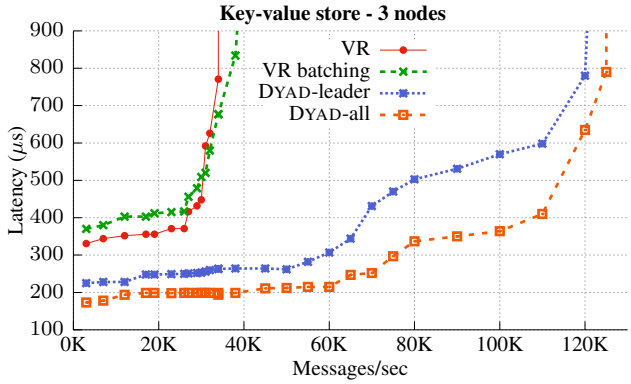


Figure 5.10: 99th percentile latency as a function of throughput for a 3-node testbed deployment of VR protocol running on the CPU vs the SmartNIC for a key-value store. DYAD-leader increases the throughput by up to 3.6x and reduces the latency by up to 65% for 34K PPS. In addition, DYAD-all reduces the latency by another 10% for 34K PPS.

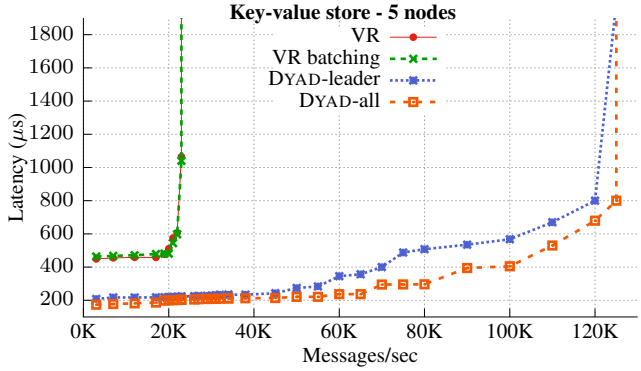


Figure 5.11: 99th percentile latency as a function of throughput for a 5-node testbed deployment of VR protocol running on the CPU vs the SmartNIC for a key-value store. DYAD-leader increases the throughput by up to 5.3x and reduces the latency by up to 79% for 22K PPS. In addition, DYAD-all reduces the latency by another 20 μs for 22K PPS.

Key-value store. We next evaluate the throughput vs latency of DYAD for a key-value store with multiple replicas (three, five, and seven nodes). The key-value store is evaluated by performing put operation with 64 byte data. With three-replicas, the results in Figure 5.10 show that DYAD-leader improves the throughput by up to 3.6x and the tail latency by up to 65%. Similar to the time-stamp server, handling 1 Million messages on the SmartNIC improves the throughput and latency of a key-value store. However, compared to the timestamp sever, the put operation takes longer processing time on the replicas which is evident in DYAD-all performance. i.e., DYAD-all reduces the latency by up to 10% which shows that replica's contribution to end-to-end latency.

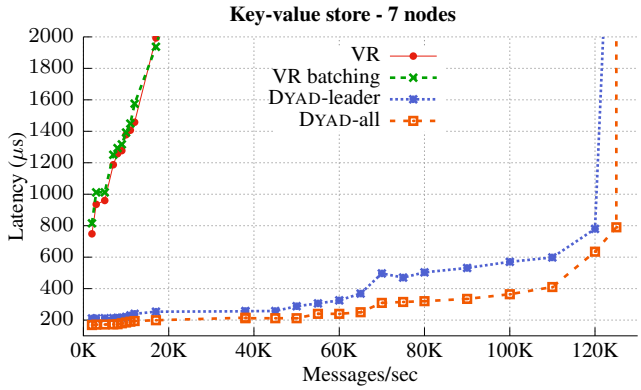


Figure 5.12: 99th percentile latency as a function of throughput for a 7-node testbed deployment of VR protocol running on the CPU vs the SmartNIC for a key-value store. DYAD-leader increases the throughput by up to 7.3x and reduces the latency by up to 87% for 17K PPS. In addition, DYAD-all reduces the latency by another 3% for 17K PPS.

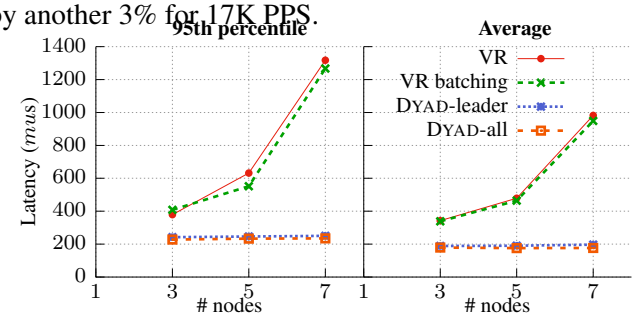


Figure 5.13: The 99th percentile and average latency of a key-value store with increasing number of replicas. DYAD-leader improves the tail and average latency by up to 80%. DYAD-all reduces the latency by another 20 μs.

We evaluate the throughput and latency of key-value store using DYAD with five and seven replicas. The performance of the baseline key-value reduces drastically with increased number of replicas which is due to increased number of messages. DYAD improves the throughput by 5.3x for a five-node setup and the latency by up to 79% compared to the baseline. And with the seven-node setup, DYAD improves throughput by up to 7.3x and the latency by up to 87% compared to the baseline. With five and seven replicas, the number of messages handled on the SmartNIC is 2 and 3 Million respectively.

The 99th percentile and average latency of the key-value store with increasing replicas is shown in Figure 5.13. Both the 99th and 50th percentile latency increase by up to 71% and 62% respectively for the baseline. DYAD reduces both the 99th and 50th percentile latency by up to 80%, and sustains the lower latency with increasing number of nodes

Increasing clients. We evaluate the timestamp server running on DYAD with increasing

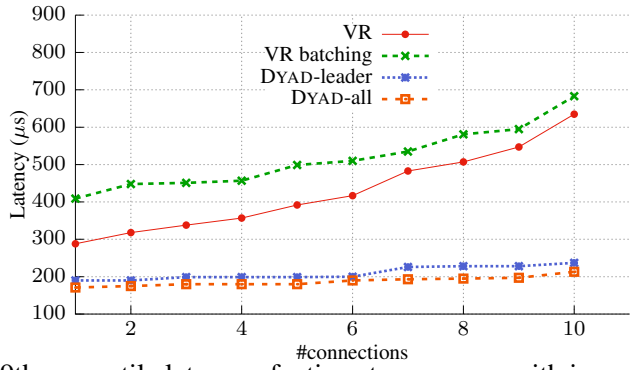


Figure 5.14: The 99th percentile latency of a timestamp server with increasing connections. DYAD-leader improves the tail latency by up to 62% and DYAD-all reduces the tail latency by another 4%.

number of clients and with three replica nodes. Figure 5.14 shows the 99th percentile latency with increasing client connections. The cost of handling more client connections increases linearly in the baseline due to decoding the five-tuple connection identifier and hash lookup on the host processors. With DYAD, the message decoding is done in SmartNIC hardware, using P4, which reduces the cost of identifying the five-tuple needed for connection lookup. DYAD-leader reduces the 99th percentile latency by up to 62% and DYAD-all further reduces the 99th percentile latency by up to 4%. In addition, DYAD sustains the lower latency with increasing number of connections.

CPU usage. We evaluate the CPU usage of the timestamp server running on DYAD to the VR implementation on the host. Figure 5.15 shows the throughput vs % CPU usage on the host processor. The baseline VR reaches 100% CPU usage with 36K PPS which limits the total throughput. In baseline VR, most of the time is spent of service context switches which is 300K on the leader and 200K on the replicas. Kernel-bypass mechanisms such as DPDK do not help in reducing the CPU usage because of their poll-mode drivers which runs always at 100%. DYAD reduces the CPU usage by up to 62% for handling 36K PPS by only handling the service messages on the host processor. With DYAD, the host CPU usage only increases with increasing number of client requests.

Raft latency. In addition to evaluating the latency of VR protocol with timestamp server and key-value store, we evaluate the latency of timestamp server with Raft protocol which is shown in Figure 5.16. We evaluate Raft with in-memory logging where the replicas

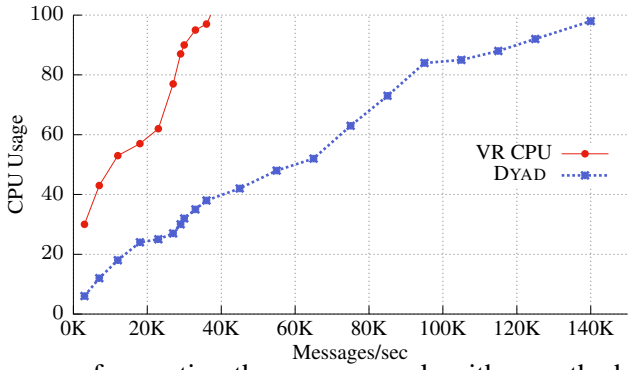


Figure 5.15: CPU usage of executing the consensus algorithm on the host vs the SmartNIC. By handling the VR protocol on the SmartNIC, DYAD reduces the CPU usage by up to 62%.

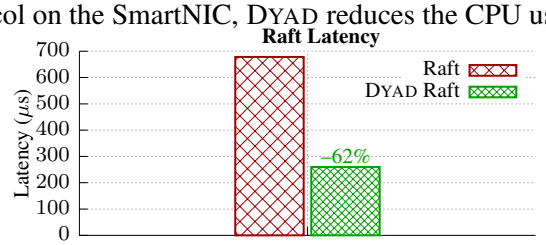


Figure 5.16: 99th percentile latency of Raft consensus executed on the host vs the SmartNIC. DYAD improves Raft latency by up to 61%.

communicate using TCP protocol. DYAD improves Raft latency by up to 61% by handling the consensus messages on the SmartNIC. The latency improvement is attributed to running the TCP protocol on the SmartNIC which reduces the end-to-end latency of 64 byte TCP ping-pong messages by up to 50%. In addition, the other benefits observed in VR protocol such as handling more client connections, increased number are replicas, are applicable to Raft.

5.5.2 Service parallelism

DYAD enables service parallelism by the SmartNIC subsystem appending logical timestamp to the client requests. We evaluate the throughput vs latency of timestamp server with parallelism, and compare DYAD-all-parallelism to DYAD-leader and DYAD-all. With 3 service threads processing the client requests, DYAD-all-parallelism improves throughput by up to 2.19x compared to DYAD-leader. The latency of DYAD parallelism is 15 μs greater than the latency of DYAD-all because of the synchronization needed with the atomic operations. However, the client request are read in parallel from all the threads having the

next client request ready to execute before the current request is processed. By having the next request ready, DYAD-parallelism avoids the read latency incurred after processing the current request. In addition, compared to baseline VR, DYAD improves the throughput by up to 8.3x and the latency by up to 68.5%.

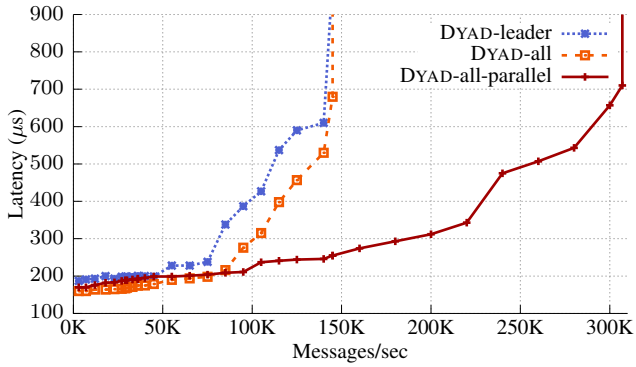


Figure 5.17: 99th percentile latency as a function of throughput for a 3-node testbed deployment of VR protocol running on the SmartNIC with parallelism for a time-stamp server. Parallel execution leveraging the logical timestamp improves the throughput of timestamp server by up to 2.19x.

5.5.3 Fault tolerance

We evaluate DYAD's effectiveness of identifying service failures on the SmartNIC with different multiples of RTT values. First, we measure the adaptive RTT by timestamping the operations on the SmartNIC. Table 5.5 shows the breakdown of the end-to-end latency including the RTT calculated on the SmartNIC with different number of replicas. The results, which includes the time on wire, show that the DYAD-leader reduces the latency of consensus operation to 52 μs, and DYAD-all reduces the latency of consensus by another 36 μs. The latency of service processing on the host, which includes the PCIe and DMA latency, is up to 98 μs. We use the calculated RTT from the host to set the threshold for identifying service failures without false positives.

The fault tolerance threshold is set to multiples of the RTT values, and evaluated to detect the false positives. We send million client requests and identify the false positives that detect service failure when the timestamp server is still running. Figure 5.18 shows the outliers that are detected as false positives with the respective configured threshold. With a

Table 5.5: Breakdown of the end-to-end latency for a timestamp server. In DYAD-leader, the latency of consensus operation is up to 52 μs, and DYAD-all reduces the latency of consensus by another 36 μs. The latency of service processing on the host is up to 41 μs, and the rest of the time is sent in client processing and network latency which is up to 42 μs.

System	Nodes	Consensus	Host	Other	Total - 99th
DYAD-Leader	3	45 μs	95 μs	53 μs	193 μs
	5	48 μs	98 μs	51 μs	197 μs
	7	52 μs	93 μs	54 μs	199 μs
DYAD-All	3	16 μs	95 μs	53 μs	164 μs
	5	17 μs	98 μs	51 μs	166 μs
	7	16 μs	93 μs	54 μs	163 μs

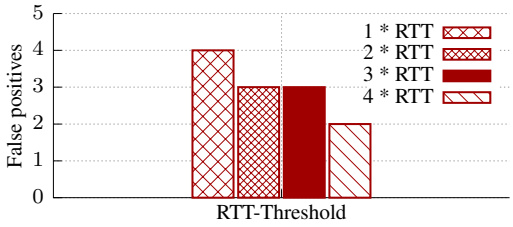


Figure 5.18: Number of false positives with various multiples of RTT values. Multiples below five result in false positives.

multiple less than five, the system identifies false positives where there are requests that are outliers whose response is received after the threshold. The reason for such outliers could be the first packet that triggers an interrupt, whereas the rest of the packets are received using the polling mode in the Linux kernel. Such false positives are not seen with multiples higher than five.

5.5.4 Recovery

We evaluate service recovery by using the ordered logs available on the SmartNIC. First, we evaluate the latency of reading the logs from the SmartNIC using the PCIe interface. Figure 5.19 shows the PCIe read latency with varying log sizes. The read throughput of a single threaded log reader is only 16 MB. However, the results show that the PCIe read throughput increased linearly with increasing number of threads. DYAD shards the logs and issues a multi-threaded read operation with each thread reading an independent shard. In our system with 16 cores, multi-threaded read increases the throughput by up to 256 MB/s. On a

system with higher core count, the read throughput can be further improved by increasing the thread count. In addition, the log read operation is performed before recovering/starting the service. So, the service performance is not impacted by the multi-threaded read operation.

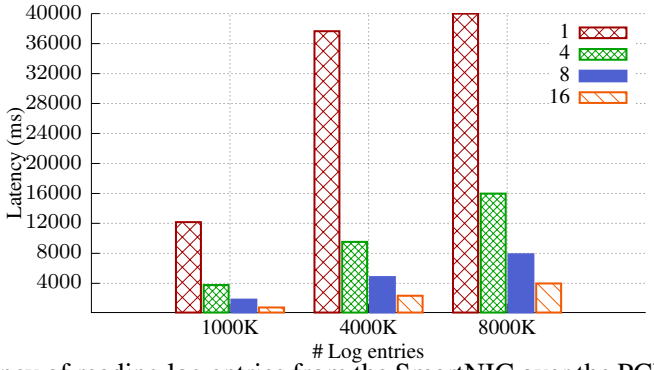


Figure 5.19: Latency of reading log entries from the SmartNIC over the PCIe interface for increasing log entries. A single thread PCIe throughput is 16 MB/s, which is increased to 256 MB/s with 16 threads.

We next evaluate the entire service recovery time that includes reading the logs from the SmartNIC, service running the recovery routine, and the finally syncing with all the replicas. We evaluate the time-taken for recovery using 1000K logs on the SmartNIC by terminating and restarting the service immediately. Figure 5.20 shows the normalized throughput over a 30 second window where the service is terminated and restarted. DYAD recovers the logs in 800 ms, and service recovers with the logs in 5 ms. The DYAD library strips the network (Ethernet, IP, and UDP) headers that were logged on the SmartNIC, and provides only the client request payload to the service. The total recovery time includes the process spawn time when the service is restarted.

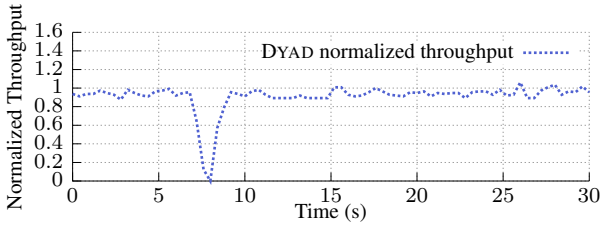


Figure 5.20: Recovering timeserver after service failure. The service around 800 ms to recover the logs from the SmartNIC, and around 5 ms to recover the service from the logs.

5.6 Chapter Summary

We introduced DYAD, which provides a physically isolated consensus by leveraging SmartNICs available in data centers. By physical isolation, DYAD eliminates the consensus component from competing for system resources with the service which improves the service performance. Apart from the normal-case consensus, DYAD provides mechanism to recover the service using the ordered logs after a software failure, and provides fault tolerance to detect service and SmartNIC failures. We demonstrated the benefits of DYAD with services such as timeserver and key-value store which shows that it improves the the throughput and tail latency by up to 8.2x and 90% respectively. In addition, DYAD reduces the CPU usage by up 62% by processing only the client request on the host processors.

CHAPTER 6

DISCUSSION

The previous chapters (§3, §4, §5) focused on three lower level mechanisms that reduce latency: XPS that reduces the protocol stack overhead, LATR that reduces the TLB shootdown overhead, and DYAD that reduces the consensus algorithm overhead. However in the previous chapters, we detailed and analyzed the importance of these mechanisms in isolation. In this chapter, we analyze the impact of latency when the three mechanisms are used in tandem. In addition, we finally summarize the lessons learned while working on this dissertation.

6.1 Lower level mechanisms in tandem

In this section, we analyze services such as web servers, log sequencers, and lock managers that benefit from a combination of the three mechanisms detailed in this dissertation. We will discuss the use cases for these services where more than one mechanism is useful.

6.1.1 Web Servers

With web servers [113], we analyze the applicability of XPS and LATR mechanisms to reduce the service latency. Web servers process HTTP GET methods by serving them from the files available in the system. For serving such files, web servers `map` the corresponding files into memory, send the response using the mapped files, and then `munmap` the file, which triggers a TLB shootdown for every HTTP request. In addition to serving HTTP requests, web servers filters and block HTTP methods that are not supported in their system which protect them from L7 DDoS attacks. For example, a Nginx web server that is configured to server HTTP GET methods is also configured to block HTTP POST methods, which prevents the sophisticated L7 attacks using HTTP POST methods. In such a case, the HTTP

GET methods should be served with less TLB shootdown overhead. In addition, the HTTP POST methods should be blocked with less system overhead.

In web servers, the mechanisms developed in LATR can be used to reduce the TLB shootdown overhead for serving HTTP GET methods, and the mechanism developed in XPS can be used to reduce the system overhead for blocking HTTP POST methods. LATR reduces the latency of serving HTTP GET methods by eliminating the synchronous shootdown overhead which is up to 16 μs in a 2 socket machine. In addition, the DDoS filtering and blocking can be implemented using XPS fast patch on SmartNIC which reduces the system overhead for blocking HTTP POST methods. XPS slow path is processing the HTTP GET methods using the existing socket interface, which is optimized by LATR. Throughput and latency of the HTTP GET method can be improved further (compared to using the mechanisms in isolation) because the CPU is only available for processing GET methods (POST methods are dropped on the SmartNIC by using XPS fastpath) and the latency incurred due to TLB shootdown is reduced by LATR.

6.1.2 Log sequencers

Log sequencer is used in distributed services, such as CORFU [147], to assign a 64-bit token that provides a unique entry in a distributed log. The performance of a log sequencer is important for the performance of services such as CORFU. Fault tolerance for a log sequencer is provided using consensus algorithms such as VR.

XPS and DYAD mechanisms together can provide the performance improvement for log sequencers. DYAD can be used to implement the consensus algorithm for the log sequencer, and XPS fast path can be used to implement the sequencer functionality. With such an implementation, the service's latency with consensus can be reduced approximately to 20 μs with a 3 three node replica. In a log sequencer where the service's functionality is simple, the host processor need not be used for executing any operation.

6.1.3 Distributed lock services

Distributed lock services, such as Zookeeper [17] and Google Chubby [16], provides lock service to other services such as Google File System (GFS) [148] and Bigtable [149]. Distributed lock services are replicated using consensus algorithms such as Paxos and Zookeeper Atomic Broadcast (ZAB). Write operations to the lock service are replicated using a consensus algorithm. The lock service stores the lock entries in the host memory which increases with the number of lock entries.

DYAD and LATR can provide the performance improvements for distributed lock services. DYAD's hardware filtering and processing mechanism can filter the write operations and execute the consensus operations on the SmartNIC. The write requests are forwarded to the host processor for storing them on the host memory. In memory constrained environments, using containers or virtual machines, page swap is triggered when the number of entries in the lock manager increases. LATR's lazy mechanism can reduces the TLB shootdown cost, in such memory constrained environment, when the page swap operation triggers a TLB shootdown.

6.2 Case study

In this section, we demonstrate the impact of these three lower-level mechanism with Memcached, a key-value store.

6.2.1 Memcached

Memcached is an in-memory key-value store which supports read (`GET`) and write (`SET`) operations. In key-value stores, the `GET` operations are latency critical operations and the `GET` operations are not latency critical. In real-world deployments, `GET` operations are more frequent than `SET` operations. For example, the `ETC` workload in Facebook, using Memcached as the key-value store, has a `GET`/`SET` ratio of 30:1 [118, 120]. In addition to the

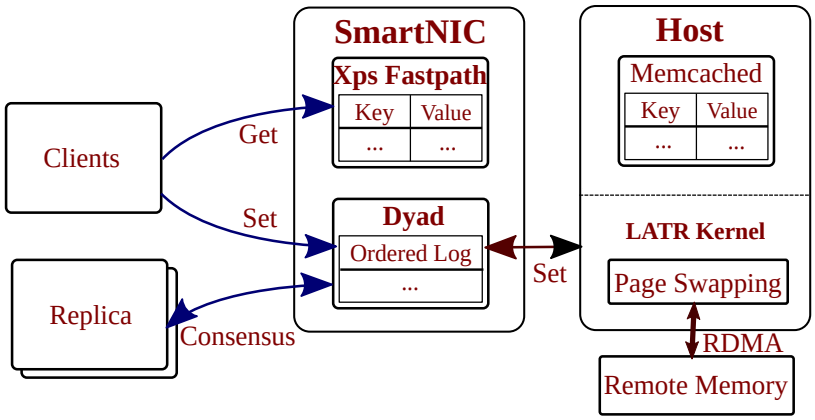


Figure 6.1: Demonstration of Memcached that uses XPS fastpath for the GET operations, uses DYAD consensus component for the SET operations and the SET operations are sent to the service running on the host after consensus, and LATR kernel provides lazy TLB shootdown for page-swap operation.

service operations, the operating system performs operations such as page swapping and migration which triggers TLB shootdown.

The GET operations are handled using XPS' fast path in the SmartNIC, which handles the GET request using the GET handler on the SmartNIC and sends a response using the handler data. The SET operations are handled by DYAD consensus, which filters the SET requests and executes the consensus operations on the SmartNIC. After the majority of replicas respond, the SET requests are sent to the host processor where Memcached executes the SET operations. The set operations executed on the host processor increases the memory used by Memcached which triggers the page-swap operation. The LATR kernel running on the host processor provides the lazy shootdown mechanism for Memcached reducing the TLB shootdown overhead. Figure 6.1 shows Memcached with XPS, DYAD, and LATR mechanisms.

Evaluation. We evaluated Memcached using INFINISWAP [15] that uses remote memory as the swap device. The evaluation setup is provisioned with two NIC cards, where the client messages are processed using the Netronome SmartNIC and swapping is done over the other RDMA NIC card. In addition, we constrain the service inside an `lxc` container on the host processor and set the soft-memory limit of the container to about half of the services working set to induce swapping via `kswapd`. We run the memaslap benchmark with 80%

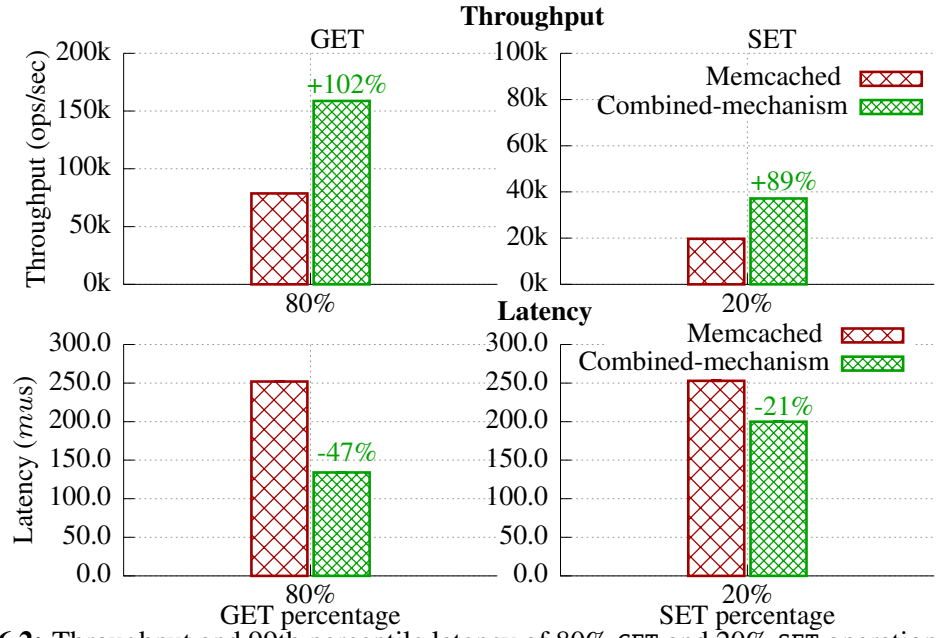


Figure 6.2: Throughput and 99th percentile latency of 80% `GET` and 20% `SET` operations. The `GET` operations throughput and latency is improved by 102% and 47%, respectively. The `SET` operations throughput and latency is improved by 89% and 21%, respectively.

Gets and 20% Sets, where the Gets are handled in XPS fast path in SmartNIC and the Sets are first handled by DYAD (as shown in Figure 6.1). After consensus, the `SET` operations are executed on the host processor which triggers the page swap operation in the LATR kernel (as shown in Figure 6.1). The performance improvement of `GET` operation is due to the XPS fast path operations running closer to I/O devices. The performance improvement of `SET` operation (shown in Figure 6.2) is due to three reasons, the fast consensus operation provided by DYAD, the lazy TLB shootdown provided by LATR, and the advantage of slow path provided by XPS.

6.3 Lessons learned

We next present a summary of general lessons we learned while working on this dissertation.

6.3.1 The need for disaggregating the functionality of services

Single machines are built as distributed systems containing various compute elements, such as GPUs, smarter NICs, and smarter NVMe, which shows that compute elements are

disaggregated over the PCIe interface. In a conceptual way, various computing devices are connected over the PCIe bus in a single machine [95]. However, current system services are developed as a monolithic entity running on the host processor. Research approaches have accelerated such services to run on many-core accelerators, such as GPUs and FPGAs [92]. In our process of reducing the latency of services, we stumbled upon the fact that disaggregating (breaking down) services into parts that leverage different compute device is important in future architectures. For example, XPS executing Redis reads on the SmartNIC and the write on the host processor. Similarly, DYAD executing consensus data operations on the SmartNIC and the control operations on the host processor. With multiple processing elements in a single machine, breaking a general service such as key-value store into micro-service which run on different processing elements is an important direction. In addition, the generic micro services could be combined to provide other services. For example, the consensus micro service running on the SmartNIC can be composed and used by various services such as key-value stores, databases, lock managers, and timestamp servers. We learned that breaking such monolithic functionality needs both the knowledge about the service and the system architecture.

Service redesign and placement. Service redesign is required to leverage the various compute elements available in a single machine. Identifying and placing the operations on different compute elements is a challenge. During the process of building XPS, we overcame this challenge by identifying the latency sensitive operations and placing them closer to the I/O devices. Though, we reused the service running on the host processor for other operations. Programming languages, such as P4 and eBPF, enable breaking down the service into operations and placing such operations transparently on various compute elements or even on the host processor.

Data consistency of operations. Placing various operations of a service on different computing elements is one part of the challenge, the data consistency provided to the entire service is another part of the challenge. When operations are performed on different compute

elements, consistency of the data provided by the entire service should not be sacrificed. In most cases, similar to distributed systems, eventual consistency is an easier consistency model to provide. However, services need strict consistency models than eventual which is important to provide. For example, research approaches try to reduce the performance difference between eventual and strong consistency in geo-distributed services [150], which is also important and applicable in a single system. We learned that design principles, drawn from computer architectures, such as write back and write through caches enable developing the needed consistency models in the system. XPS data operations in the kernel and SmartNIC (§3.2.4 and §3.2.5), where the data is isolated from the service, shows that consistency can be provided when operations are executed on different processing elements.

6.3.2 Impact of cross-layer optimization

Abstractions provide generality and simplicity to develop services that can run on various operating systems and hardware. However, software abstractions, such as the POSIX and VFS layer, induce high latency due to the indirections they create. For example, every file system access should traverse the VFS layer which adds additional system overhead.

In the initial stage of our research, we first focused on implementing an efficient protocol stack which reduces latency. In the process, we figured out that optimizing POSIX interface that interacts with the user-space is much more important than the protocol stack. For example, the socket interface developed in even a state-of-the-art user-space protocol stack incurs high overhead similar to the kernel protocol stack [25]. An approach that we learned, in the process, is the impact of *cross-layer optimization*, that allows a service to execute operations in a lower layer (closer to the protocol stack). Cross layer optimization is not a new idea [8, 9, 27], Ashs [8] and Plexus [27] provide mechanisms to reduce the software layering overheads in protocol stacks which are not available in current operating systems. However, such mechanism are useful in the era of optimizing micro-second latency. XPS' fast path is inspired from Ashs and Plexus, however XPS provides a generalized and practical

mechanism compared to the other systems. Instead of strictly adhering to the POSIX or VFS interface, cross-layer optimization enhances such interface to reduce their indirection overhead.

Similar to the POSIX and VFS interface, the Linux Application Binary Interface (ABI) enforces certain strict guarantees that an operating system should provide. For example, in an munmap system call, the ABI enforces synchronous TLB shootdown which imposes latency of up to 80 μs. A *cross-layer optimization* to relax such strict ABI and develop lazy mechanism reduces the latency of munmap operation drastically. Similar to POSIX and VFS interface, the Linux ABI enforced latency can be overcome with cross-layer optimizations.

In summary, we learned the following from cross-layer optimizations:

- The layered protocol stack in the kernel and user-space have clear interfaces which enables cross-layer optimization. For example, the TCP protocol layer has an interface which forwards packets to the socket layer (POSIX), which enabled running service logic (L7) immediately.
- The ABI restrictions could be overcome by introducing new flags in system call interfaces, such as `mmap` and `munmap`, that provide lazy mechanisms. Such a new interface enables providing lazy mechanisms to other system call interfaces such as `mprotect`.

6.3.3 The importance of low-level mechanisms with fast I/O devices

Low level system operations, such as synchronization, scheduling, page swapping and migrations, need to operate at lower latency to catch up with the fast I/O devices. For example, instead of swapping pages to disk, recent research systems, such as INFINISWAP [15], advocate for the usage of remote memory using Infiniband RDMA, which reduces the tail latency of page swapping by up to 61x. Due to the reduced remote paging latency, the TLB shootdowns needed for swapping become an important contributor to the cost of page swapping (contributing up to 18% for a Memcached workload using INFINISWAP).

The above example shows the shift in paradigm (With RDMA, NVM, and NVMe) where the low-level operations, which were not the major contributors with older I/O devices, are currently the major contributors to latency. Similar results were seen with 100 Gbps NIC cards, where the packet processing overhead in software should be reduced to few nanoseconds to saturate such NIC cards. With our experiments, we learned the impact of such lower-level mechanisms that become the major contributors of latency and the need for optimizing such lower-level mechanisms.

Hardware support. As discussed in this dissertation, hardware support is important for reducing latency in new I/O devices. We learned about two possible direction: first, the software utilizes the hardware efficiently (LATR) and second, the most important direction is software (DYAD and XPS) defining the direction for hardware optimizations that improves system service latency. For example, in DYAD, the latency of consensus operations can be further improved by reducing the memory access latency in SmartNICs. Similarly, XPS fast path will benefit from SmartNIC optimizations that provide fast remote memory access from the host processors.

CHAPTER 7
CONCLUSION AND FUTURE WORK

This dissertation covers three lower level mechanisms that help in reducing the latency of system services. We summarize key contributions before briefly discussing future work.

7.1 Conclusion

Server network bandwidth is growing rapidly from 10/25 Gbps to 100/200 Gbps, and even 400 Gbps will become a reality soon. However, the compute power in modern processors is not growing fast enough to handle these large network bandwidth. In addition, the lower layer mechanisms used in modern systems are not efficient enough for handling these large network traffic. We address the inefficiencies in the lower layer mechanisms through the research presented in this dissertation.

We started with showing the inefficiencies in current protocol stacks, and their overheads on current system services such as Redis and Nginx. To address the protocol stack overhead, we presented an *extensible* protocol stack, XPS, that allows an application to specify its latency-sensitive operations and to execute them *inside* the kernel, user-space protocol stacks and smart NICs, providing higher throughput and lower tail latency by avoiding the socket interface. For all other operations, XPS retains the popular, well-understood socket interface. XPS' approach is practical: rather than proposing a new OS or removing the socket interface completely, our goal is to provide stack extensions for latency-sensitive operations and use the existing socket layer for all other operations.

Our evaluation showed XPS' benefits for real-world system services, such as caching in a key value store, filtering and blocking HTTP requests in a web server, and processing messages in distributed systems: XPS improves their throughput and tail latency by up to 4x and 82%.

Next, we showed the inefficiencies in a synchronous TLB shootdown and their impact of web servers such as Apache. To address the synchronous TLB shootdown overhead, we presented LATR—lazy TLB coherence—a software-based TLB shootdown mechanism that can alleviate the overhead of the synchronous TLB shootdown mechanism in existing operating systems. By handling the TLB coherence in a lazy fashion, LATR avoided expensive IPIs which are required for delivering a shootdown signal to remote cores, and the performance overhead of associated interrupt handlers. Therefore, virtual memory operations, such as *free*, *page migration*, and *page swapping* operations, can benefit significantly from LATR's mechanism. For example, LATR improves the latency of `munmap()` by 70.8% on a 2-socket machine, a widely used configuration in modern data centers.

Real-world, performance-critical system services such as web servers can also benefit from LATR: without any application-level changes, LATR improves Apache by 59.9% compared to Linux, and by 37.9% compared to ABIS, a highly optimized, state-of-the-art TLB coherence technique. In addition, LATR improves Apache latency by up to 26.1%

Finally, we showed the overhead of consensus algorithm on distributed services such as time-stamp server and key-value stores. To overcome the overheads of a consensus algorithm, DYAD untangles the tightly coupled consensus mechanism from the application by physically isolating them, allowing the high-overhead consensus component to run on the SmartNIC and the application to run on the host processor. By physical isolation, DYAD eliminates the consensus component from competing for system resources with the application which improves the application performance. Apart from the normal-case consensus, DYAD provides mechanism to recover the application using the ordered logs after a software failure, and provides fault-tolerance to detect application and SmartNIC failures.

DYAD evaluation with three, five and seven replicas showed that it improves the throughput and tail latency of distributed services, such as timestamp server and key-value store, by up to 8.2x and 90% respectively. In addition, DYAD reduces the CPU usage by up 62% by

processing only the client request on the host processors.

This dissertation provides hard evidence about the inefficiencies in lower-level mechanisms, such as protocol stack, TLB shootdown, and consensus algorithms, that increase the latency of system services. In addition, this dissertation demonstrates how innovative operations such as lazy shootdown, extensible protocol stack, and untangling consensus, address the inefficient implementation of lower-level mechanism and they improve the latency of system services.

7.2 Future work

With the end of Dennard scaling, the compute power in modern processors is not growing fast enough to handle the 100/200 Gbps of traffic generated in next-generation data centers. As presented in this dissertation, to handle such large volume of data, efficient lower-level mechanisms both in software and hardware are needed to reduce the overhead of system services in data centers. The insights gathered from this dissertation work provides suggestions for promising future directions.

Low-level mechanisms in virtualized environments. Addressing the software overheads in virtualized environments is challenging due to the various indirections used in these environments. For example, the VM exits triggered due to operations such as a synchronous TLB shootdown incurs high overhead compared to a baremetal environment. Similarly, even the overhead of an asynchronous TLB shootdown mechanism is significant due to the VM exits incurred. Apart from the TLB shootdown, protocol stacks in VMs incur significant overheads due to the scheduling overheads incurred on VMs, which can be addressed by providing protocol as a service in the hypervisor. In addition, system services running on VMs need abstractions to leverage future hardware technologies, such as programmable switches and SmartNICs, which will reduce the overhead incurred in virtualized environments.

Improving lower-layer mechanisms with hardware. Hardware changes enable software mechanisms to further reduce the latency of system service. Improving the latency of mem-

ory accesses helps lower level mechanisms such as a lazy TLB shootdown and consensus algorithms. For example, LATR could benefit from the availability of a global coherent scratchpad memory which will further reduce the TLB shootdown latency. Similarly, such a memory access in SmartNICs will further reduce (from 17 μs provided by DYAD) the latency of consensus algorithms. In addition, leveraging hardware features such as Intel's cache allocation technology (CAT) is an interesting direction to reduce the latency incurred due to memory accesses. In summary, building efficient lower level mechanisms coupled with hardware optimizations will further reduce latency of system services.

Latency of operating system operations. In addition to the mechanisms analyzed in this dissertation, other mechanisms in operating system operations, such as context switch, meta-data allocation, lock contention, induce high and unpredictable latency in system services in data centers. In particular, the context-switch overhead in the operating system is the reason for unpredictable latency in data centers [80]. Similarly, the meta-data allocation overhead (SKB in Linux) induces high latency to packet processing in the kernel. In addition, traditional inter-core communication mechanism using IPI limits the latency of operating system operations spanning across large number of cores. The above overheads are some of the reasons that impact the latency of services in micro-seconds scale, and which presents an opportunity for further improvement.

We open sourced LATR (`https://github.com/sslab-gatech/latr`) which enables further research using our software artifact . In addition, we will open source the remaining software artifacts which will enable future research detailed in this dissertation.

REFERENCES

[1] *Starting the Avalanche*, https://medium.com/netflix-techblog/starting-the-avalanche-640e69b14a06, 2017.

[2] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the Killer Microseconds," *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, Mar. 2017.

[3] R. Myslewski, *Google Research: Three things that MUST BE DONE to save the data center of the future*, https://www.theregister.co.uk/2014/02/11/google_research_three_things_that_must_be_done_to_save_the_data_center_of_the_future/, 2014.

[4] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's Time for Low Latency," in *13th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, Napa, CA, May 2011.

[5] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network Requirements for Resource Disaggregation," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[6] *BlueField SmartNIC*, http://www.mellanox.com/page/products_dyn?product_family=275&mtag=bluefield_smart_nic, 2018.

[7] *Agilio SmartNIC*, https://open-nfp.org/resources/, 2018.

[8] D. A. Wallach, D. R. Engler, and M. F. Kaashoek, "ASHs: Application-Specific Handlers for High-Performance Messaging," in *Proceedings of the 7th ACM SIGCOMM*, Palo Alto, CA, Aug. 1996.

[9] G. R. Ganger, D. R. Engler, M. F. Kaashoek, H. M. Briceño, R. Hunt, and T. Pinckney, "Fast and Flexible Application-level Networking on Exokernel Systems," *ACM Transactions on Computer Systems*, vol. 20, no. 1, Feb. 2002.

[10] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo, "EbbRT: A Framework for Building Per-Application Library Operating Systems," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[11] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The Operating System is the Control Plane," in *Proceedings*

of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, Colorado, Oct. 2014.

[12] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb, "Die Stacking (3D) Microarchitecture," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Orlando, FL, Dec. 2006, pp. 469–479.

[13] M. Oskin and G. H. Loh, "A Software Managed Approach to Die-Stacked DRAM," in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, San Francisco, CA, Sep. 2015, pp. 188–200.

[14] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories," in *Proceedings of the 21st IEEE Symposium on High Performance Computer Architecture (HPCA)*, San Francisco, CA, Feb. 2015, pp. 126–136.

[15] G. Juncheng, L. Youngmoon, Z. Yiwen, C. Mosharaf, and S. Kang, "Efficient Memory Disaggregation with Infiniswap," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Apr. 2017.

[16] M. Burrows, "The Chubby Lock Service for Loosely-Coupled Distributed Systems," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10, Boston, MA: USENIX Association, 2010, pp. 11–11.

[18] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14, Philadelphia, PA: USENIX Association, 2014, pp. 305–320, ISBN: 978-1-931971-10-2.

[19] *Meltdown fix impact on Redis performances in virtualized environments*, `https://news.ycombinator.com/item?id=16079457`, 2018.

[20] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.

[21] L. Soares and M. Stumm, "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls," in *Proceedings of the 7th ACM SIGCOMM*, Palo Alto, CA, Aug. 1996.

[22] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[23] 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)*, Shanghai, China, Oct. 2017, pp. 325–341.

[24] *DPDK*, `http://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html`, 2017.

[25] E. Jeong, S. Woo, M. 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 USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, Apr. 2014.

[26] I. Marinos, R. N. Watson, M. Handley, and R. R. Stewart, "Disk—Crypt—Net: rethinking the stack for high-performance video streaming," in *Proceedings of the 2017 ACM SIGCOMM*, Los Angeles, CA, Aug. 2017, pp. 211–224.

[27] M. E. Fiuczynski and B. N. Bershad, "An Extensible Protocol Architecture for Application-Specific Networking," in *Proceedings of the 1996 USENIX Annual Technical Conference (ATC)*, San Diego, CA, Jan. 1996.

[28] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[29] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, Dec. 1995.

[30] *eBPF: extended Berkley Packet Filter*, `https://www.iovisor.org/technology/ebpf`, 2017.

[31] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *Proceedings of the Winter 1993 USENIX Annual Technical Conference (ATC)*, San Diego, CA, Jan. 1993.

[32] A. Begel, S. McCanne, and S. L. Graham, "BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture," in *Proceedings of the 10th ACM SIGCOMM*, Cambridge, MA, Aug. 1999.

[33] *LinuxBPF classifier and actions in tc*, `http://man7.org/linux/man-pages/man8/tc-bpf.8.html`, 2017.

[34] *eXpress Data Path*, `https://www.iovisor.org/technology/xdp`, 2017.

[35] Intel, *Multiprocessor Specification*, 1997.

[36] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy, "UNified Instruction/Translation/Data (UNITD) Coherence: One Protocol to Rule Them All," in *Proceedings of the 16th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Bangalore, India, Jan. 2010, pp. 1–12.

[37] Z. Yan, J. Veselý, G. Cox, and A. Bhattacharjee, "Hardware Translation Coherence for Virtualized Systems," in *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, Jun. 2017, pp. 430–443.

[38] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Ünsal, "DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Galveston Island, TX, Oct. 2011, pp. 340–349.

[39] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying Memory Ordering at the Hardware-OS Interface," in *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016, pp. 233–247.

[40] B. F. Romanescu, A. R. Lebeck, and D. J. Sorin, "Specifying and Dynamically Verifying Address Translation-aware Memory Consistency," in *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, Mar. 2010, pp. 323–334.

[41] L. Anaczkowski, *Linux VM workaround for Knights Landing A/D leak*, `https://lkml.org/lkml/2016/6/14/505`, 2016.

[42] C. Covington, *arm64: Work around Falkor erratum 1003*, `https://lkml.org/lkml/2016/12/29/267`, 2016.

[43] L. K. D. Database, *CONFIG ARM ERRATA 720789*, http://cateee.net/lkddb/web-lkddb/ARM_ERRATA_720789.html, 2017.

[44] A. L. Shimpi, *AMD's B3 stepping Phenom previewed, TLB hardware fix tested.* http://www.anandtech.com/show/2477/2, 2008.

[45] T. Valich, *Intel explains the Core 2 CPU errata.* http://www.theinquirer.net/inquirer/news/1031406/intel-explains-core-cpu-errata, 2007.

[46] B. Pham, D. Hower, A. Bhattacharjee, and T. Cain, "TLB Shootdown Mitigation for Low-Power, Many-Core Servers with L1 Virtual Caches," *IEEE Computer Architecture Letters*, vol. PP, no. 99, Jun. 2017.

[47] ARM, *ARM Compiler Reference Guide: TLBI*, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0802b/TLBI_SYS.html, 2014.

[48] R. Arimilli, G. Guthrie, and K. Livingston, *Multiprocessor system supporting multiple outstanding TLBI operations per partition*, US Patent App. 10/425,425, Oct. 2004.

[49] N. Amit, "Optimizing the TLB Shootdown Algorithm with Page Access Tracking," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017, pp. 27–39.

[50] A. Baumann, P. Barham, P.-E. Dagand, T. 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)*, Big Sky, MT, Oct. 2009, pp. 29–44.

[51] L. Torvalds, *Linux Kernel*, https://github.com/torvalds/linux, 2017.

[52] M. Gorman, *TLB flush multiple pages per IPI*, https://lkml.org/lkml/2015/4/25/125, 2015.

[53] D. L. Black, R. F. Rashid, D. B. Golub, C. R. Hill, and R. V. Baron, "Translation Lookaside Buffer Consistency: A Software Approach," in *Proceedings of the 3rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, Apr. 1989, pp. 113–122.

[54] M. Y. Thompson, J. Barton, T. Jermoluk, and J. Wagner, "Translation Lookaside Buffer Synchronization in a Multiprocessor System," in *Proceedings of the Winter 1988 USENIX Annual Technical Conference (ATC)*, Dallas, TX, 1988.

[55] P. J. Teller, R. Kenner, and M. Snir, "TLB Consistency on Highly-Parallel Shared-Memory Multiprocessors," in *Proceedings of the 21st Annual Hawaii International*

Conference on System Sciences. Volume I: Architecture Track, vol. 1, 1988, pp. 184–193.

[56] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "RadixVM: Scalable Address Spaces for Multithreaded Applications," in *Proceedings of the 8th European Conference on Computer Systems (EuroSys)*, Prague, Czech Republic, Apr. 2013, pp. 211–224.

[57] J. K. Peacock, S. Saxena, D. Thomas, F. Yang, and W. Yu, "Experiences from Multithreading System V Release 4," in *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems*, ser. SEDMS III, Newport Beach, CA, 1992, pp. 77–91.

[58] R. Balan and K. Gollhard, "A Scalable Implementation of Virtual Memory HAT Layer for Shared Memory Multiprocessor Machine," in *Proceedings of the Summer 1992 USENIX Annual Technical Conference (ATC)*, San Antonio, TX, Jun. 1992, pp. 107–115.

[59] P. Teller, "Translation-Lookaside Buffer Consistency," *Computer*, vol. 23, no. 6, pp. 26–36, Jun. 1990.

[60] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An Operating System for Many Cores," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008, pp. 43–57.

[61] FreeBSD, *FreeBSD - PCID implementation*, `https://reviews.freebsd.org/rS282684`, 2015.

[62] D. Lustig, A. Bhattacharjee, and M. Martonosi, "TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, 2:1–2:38, Apr. 2013.

[63] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-Level TLBs for Chip Multiprocessors," in *Proceedings of the 17th IEEE Symposium on High Performance Computer Architecture (HPCA)*, San Antonio, TX, Feb. 2011, pp. 62–73.

[64] S. R. Thomas Barr Alan Cox, "SpecTLB: a Mechanism for Speculative Address Translation," in *Proceedings of the 38th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, San Jose, California, USA, Jun. 2011, pp. 307–318.

[65] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB Reach by Exploiting Clustering in Page Translations," in *Proceedings of the 20th IEEE*

Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, USA, Feb. 2014, pp. 558–567.

[66] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Vancouver, Canada, Dec. 2012, pp. 258–269.

[67] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *Proceedings of the 22st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017, pp. 435–448.

[68] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal, "Energy-Efficient Address Translation," in *Proceedings of the 22nd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Barcelona, Spain, Mar. 2016, pp. 631–643.

[69] A. Bhattacharjee, "Translation-Triggered Prefetching," in *Proceedings of the 22st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017, pp. 63–76.

[70] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008.

[71] J. Cowling and B. Liskov, "Granola: Low-overhead distributed transaction coordination," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12, Boston, MA: USENIX Association, 2012, pp. 21–21.

[72] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, pp. 223–234.

[73] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12, Hollywood, CA, USA: USENIX Association, 2012, pp. 251–264, ISBN: 978-1-931971-96-6.

[74] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[75] ——, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001.

[76] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '88, Toronto, Ontario, Canada: ACM, 1988, pp. 8–17, ISBN: 0-89791-277-2.

[77] B. Liskov and J. Cowling, "Viewstamped replication revisited," Jul. 2012.

[78] A. Medeiros, "Zookeeper ' s atomic broadcast protocol : Theory and practice," 2012.

[79] N. Rolf, A. Gianni, Z. J. Fernando, A. Yury, L.-B. Sergio, and M. A. W., "Understanding pcie performance for end host networking," in *Proceedings of the 2018 ACM SIGCOMM*, Budapest, Hungary, Aug. 2018, pp. 327–341.

[80] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18, Budapest, Hungary: ACM, 2018, pp. 297–312, ISBN: 978-1-4503-5567-4.

[81] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say no to paxos overhead: Replacing consensus with network ordering," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[82] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing distributed systems using approximate synchrony in data center networks," in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Oakland, CA, Apr. 2015.

[83] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15, Santa Clara, California: ACM, 2015, 5:1–5:7, ISBN: 978-1-4503-3451-8.

[84] E. Sakic and W. Kellerer, "Response time and availability study of raft consensus in distributed sdn control plane," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 304–318, Mar. 2018.

[85] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, M. Canini, N. Zilberman, F. Pedone, and R. Soulé, "Series in informatics p4xos : Consensus as a network service," 2018.

[86] N. K. Sharma, A. Kaufmann, T. Anderson, C. Kim, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17, Boston, MA, USA: USENIX Association, 2017, pp. 67–82, ISBN: 978-1-931971-37-9.

[87] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13, Hong Kong, China: ACM, 2013, pp. 99–110, ISBN: 978-1-4503-2056-6.

[88] M. Poke and T. Hoefler, "Dare: High-performance state machine replication on rdma networks," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15, Portland, Oregon, USA: ACM, 2015, pp. 107–118, ISBN: 978-1-4503-3550-8.

[89] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "Apus: Fast and scalable paxos on rdma," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17, Santa Clara, California: ACM, 2017, pp. 94–107, ISBN: 978-1-4503-5028-0.

[90] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast Crash Recovery in RAMCloud," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[91] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, Apr. 2014.

[92] Z. István, D. Sidler, G. Alonso, and M. Vukolić, "Consensus in a Box: Inexpensive Coordination in Hardware," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, Apr. 2016.

[93] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Renton, WA, Apr. 2018, pp. 51–66.

[94] D. Firestone, "VFP: A virtual switch platform for host SDN in the public cloud," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, 2017, pp. 315–328, ISBN: 978-1-931971-37-9.

[95] M. Flajslik and M. Rosenblum, "Network Interface Design for Low Latency Request-Response Protocols," in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, San Jose, CA, Jun. 2013.

[96] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[97] ——, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[98] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving Network Connection Locality on Multicore Systems," in *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, Bern, Switzerland, Apr. 2012.

[99] *OSv - Network Channels*, `http://osv.io/nfv`, 2015.

[100] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov, "OSv—Optimizing the Operating System for Virtual Machines," in *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, Philadelphia, PA, Jun. 2014.

[101] *Van Jacobson's network channels*, `https://lwn.net/Articles/169961/`, 2006.

[102] J. Corbet, *The current state of kernel page-table isolation*, `https://lwn.net/Articles/741878/`, 2017.

[103] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA Efficiently for Key-Value Services," in *Proceedings of the 2014 ACM SIGCOMM*, Chicago, IL, Aug. 2014.

[104] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-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*, ser. ICPP '11, 2011.

[105] S.-Y. Tsai and Y. Zhang, "LITE Kernel RDMA Support for Datacenter Applications," in *Proceedings of the 26th Symposium on Operating Systems Principles*, Shanghai, China, Oct. 2017, pp. 306–324.

[106] C. Kulkarni, A. Kesavan, T. Zhang, R. Ricci, and R. Stutsman, "Rocksteady: Fast Migration for Low-latency In-memory Storage," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017, pp. 390–405.

[107] J. Xin, L. Xiaozhou, Z. Haoyu, S. Robert, L. Jeongkeun, F. Nate, K. Changhoon, and S. Ion, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017, pp. 121–136.

[108] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamuthy, and K. Atreya, "IncBricks: Toward In-Network Computation with an In-Network Cache," in *Proceedings of the 22st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.

[109] *Nginx L7 DDOS protection*, https://www.nginx.com/blog/mitigating-ddos-attacks-with-nginx-and-nginx-plus/, 2015.

[110] S. Rombauts, *Nginx restrict method*, https://stevenrombauts.be/2017/04/deny-http-request-methods-in-nginx/, 2017.

[111] *Configuring HTTP filtering*, https://technet.microsoft.com/en-us/library/cc995081.aspx, 2017.

[112] *Redis Key-Value Store*, https://redis.io/, 2017.

[113] *Nginx Web Cache*, https://www.nginx.com/resources/wiki/, 2017.

[114] *Logcabin projetc*, https://ramcloud.atlassian.net/wiki/spaces/logcabin/overview, 2017.

[115] T. P. Morgan, *AMD Disrupts The Two-Socket Server Status Quo*, https://www.nextplatform.com/2017/05/17/amd-disrupts-two-socket-server-status-quo/, 2017.

[116] *memtier benchmark: A High Throughput Benchmarking Tool for Redis and Memcached*, https://redislabs.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/, 2013.

[117] *WRK HTTP benchmark*, https://github.com/giltene/wrk2, 2015.

[118] B. Atikoglu, Y. Xu, and E. Frachtenberg, "Workload Analysis of a Large-Scale Key-Value Store," in *Proceedings of the 12th ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, Jun. 2012.

[119] J. Leverich and C. Kozyrakis, "Reconciling High Server Utilization and Sub-millisecond Quality-of-Service," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014, 4:1–4:14.

[120] 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 USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Lombard, IL, Apr. 2013.

[121] *HTTP Protocol Stack (IIS 6.0)*, `https://docs.microsoft.com/en-us/iis/manage/managing-performance-settings/configure-iis-7-output-caching`, 2017.

[122] P. Joubert, R. B. Kingy, R. Neves, M. Russinovichz, and J. M. Traceyx, "High-Performance Memory-Based Web Servers: Kernel and User-Space Performance," in *Proceedings of the 2001 USENIX Annual Technical Conference (ATC)*, Jun. 2001.

[123] J. Corbet, *AutoNUMA: the other approach to NUMA scheduling*, `https://lwn.net/Articles/488709/`, 2012.

[124] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002, pp. 181–194.

[125] B. Pham, J. Veselý, G. H. Loh, and A. Bhattacharjee, "Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways?" In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Waikiki, Hawaii, Dec. 2015, pp. 1–12.

[126] J. Corbet, *Memory compaction*, `https://lwn.net/Articles/368869/`, 2010.

[127] Apache, *Apache HTTP Server Project*, `https://httpd.apache.org/`, 2017.

[128] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004, pp. 137–150.

[129] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in *Proceedings of the 13th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Phoenix, AZ, Feb. 2007, pp. 13–24.

[130] *Intel Xeon Processor E5-4610 v2*, `http://ark.intel.com/products/75285/Intel-Xeon-Processor-E5-4610-v2-16M-Cache-2_30-GHz`, 2014.

[131] *Intel Xeon Processor E7-8894 v4*, http://ark.intel.com/products/96900/Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2_40-GHz, 2017.

[132] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and Efficient Huge Page Management with Ingens," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016, pp. 705–721.

[133] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Toronto, Canada, Oct. 2008, pp. 72–81.

[134] *Graph500 Reference Implementations*, http://graph500.org/?page_id=47, 2017.

[135] J. Gilchrist, "Parallel Compression with BZIP2," in *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Cambridge, MA, Nov. 2004, pp. 559–564.

[136] Y. Mao, R. Morris, and F. Kaashoek, "Optimizing MapReduce for Multicore Architectures," MIT, Tech. Rep. MIT-CSAIL-TR-2010-020, May 2010.

[137] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An Analysis of Linux Scalability to Many Cores," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010, pp. 1–16.

[138] J. Leverich, *Mutilate: High-performance memcached load generator*, https://github.com/leverich/mutilate, 2017.

[139] *A high-performance, distributed memory object caching system*, http://memcached.org/, 2017.

[140] ScyllaDB, *Memcached Benchmark*, https://github.com/scylladb/seastar/wiki/Memcached-Benchmark, 2015.

[141] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a Trillion-Edge Graph on a Single Machine," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, SR, Apr. 2017, pp. 527–543.

[142] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a Social Network or a News Media?" In *Proceedings of the 19th International World Wide Web Conference (WWW)*, Raleigh, NC, Apr. 2010, pp. 591–600.

[143] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 2, pp. 18–24, May 2016.

[144] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[145] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout, "Implementing linearizability at large scale and low latency," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, Monterey, California: ACM, 2015, pp. 71–86, ISBN: 978-1-4503-3834-9.

[146] B. Pismenny, I. Lesokhin, L. Liss, and H. Eran, "TLS Offload to Network Devices," 2016.

[147] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis, "Corfu: A shared log design for flash clusters," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, San Jose, CA: USENIX Association, 2012, pp. 1–1.

[148] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, Bolton Landing, NY, USA: ACM, 2003, pp. 29–43, ISBN: 1-58113-757-5.

[149] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 4:1–4:26, Jun. 2008.

[150] H. Gupta and U. Ramachandran, "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access," in *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '18, Hamilton, New Zealand: ACM, 2018, pp. 148–159, ISBN: 978-1-4503-5782-1.