Kernel vs. User-Level Networking: Don't Throw Out the Stack with the Interrupts

PETER CAI, University of Waterloo, Canada
MARTIN KARSTEN, University of Waterloo, Canada

This paper reviews the performance characteristics of network stack processing for communication-heavy server applications. Recent literature often describes kernel-bypass and user-level networking as a silver bullet to attain substantial performance improvements, but without providing a comprehensive understanding of how exactly these improvements come about. We identify and quantify the direct and indirect costs of asynchronous hardware interrupt requests (IRQ) as a major source of overhead. While IRQs and their handling have a substantial impact on the effectiveness of the processor pipeline and thereby the overall processing efficiency, their overhead is difficult to measure directly when serving demanding workloads. This paper presents an indirect methodology to assess IRQ overhead by constructing preliminary approaches to reduce the impact of IRQs. While these approaches are not suitable for general deployment, their corresponding performance observations indirectly confirm the conjecture. Based on these findings, a small modification of a vanilla Linux system is devised that improves the efficiency and performance of traditional kernel-based networking significantly, resulting in up to 45% increased throughput without compromising tail latency. In case of server applications, such as web servers or Memcached, the resulting performance is comparable to using kernel-bypass and user-level networking when using stacks with similar functionality and flexibility.

CCS Concepts: • **General and reference** → **Performance**; • **Software and its engineering** → **Input / output**; • **Networks** → **Network servers**.

Additional Key Words and Phrases: network stack performance; interrupt mitigation; locality

ACM Reference Format:
Peter Cai and Martin Karsten. 2023. Kernel vs. User-Level Networking: Don't Throw Out the Stack with the Interrupts. *Proc. ACM Meas. Anal. Comput. Syst.* 7, 3, Article 49 (December 2023), 23 pages. https://doi.org/10.1145/3626780

1 INTRODUCTION

Recent literature (cf. Section 2) as well as reports from practitioners [27, 36, 41, 45] attest to significant performance gains arising from user-level networking in comparison to using the kernel network stack. However, user-level networking is no panacea and presents serious challenges to the design and deployment of network-centric applications. We are not aware of a comprehensive performance study that quantitatively describes which elements of kernel-bypass and user-level networking are responsible for the reduction in processing overhead and by how much. We surmise that the performance improvements are caused by two general characteristics:

(1) Customization: Reduced functionality and flexibility of a user-level stack can directly lead to a corresponding reduction in memory footprint and processing overhead.

Authors' addresses: Peter Cai, peter.cai@uwaterloo.ca, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada; Martin Karsten, mkarsten@uwaterloo.ca, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2476-1249/2023/12-ART49 $15.00
https://doi.org/10.1145/3626780

(2) Alignment: User-level network stacks cannot directly receive hardware interrupts and thus use polling to interact with the network interface card (NIC). This leads to both spatial (locality) and temporal (synchronous) alignment of network- and application-level processing, both of which are known to improve performance.

A customized network stack presents opportunities for optimization, but comes with caveats. Network protocols and their internals continue to evolve. High-efficiency and low-latency implementations are complex and error-prone. While well-known and mature network stacks, such as Linux or *BSD, typically receive timely updates and undergo extensive testing, this is not necessarily true for customized stacks. The application programming interface (API) of a custom stack might need to differ from established APIs to fully realize the customization benefits. This makes it hard to integrate existing and diverse applications. Similarly, user-level network stacks are not integrated with the operating system's scheduler and interrupt delivery mechanisms. Therefore, applications using such custom stacks typically require dedicated resource allocation, bypassing the kernel's resource management system, which reduces overall system efficiency and utilization.

The main contribution of this work is demonstrating that much better alignment of network- and application-level processing is possible without requiring massive changes or additions to a vanilla Linux system using the regular kernel network stack. The key improvement is reducing the overhead caused by asynchronous hardware interrupt requests (IRQ) and their handling. Because this overhead is impossible to measure directly at high system load, we present and evaluate two kernel configuration setups that are not useful for general deployment, but corroborate the conjecture about the impact of IRQs. Based on these observations, one small and feasible kernel code change is devised that delivers substantial benefits compared to a default system setup.

These findings have implications for best practices and research methodology. When studying the performance of network stack processing, high-level design decisions as well as experimental methodology affect IRQ handling, but the resulting effects are often not properly attributed. This can lead to an overly optimistic assessment of proposals for re-architecting network stack processing. Therefore, it is important to understand and document the effects of IRQ handling when evaluating network stack performance. Also, the findings reported here need to be taken into account when assessing the baseline performance of a vanilla network stack in comparison with novel proposals.

The rest of the paper first presents background and related work in Section 2. This is followed by formulating a simple performance model and providing a preliminary experimental analysis of network stack overhead in Section 3. Section 4 presents proposals for favourably aligning network- and application-processing, while Section 5 presents an experimental evaluation of these proposals. These sections present strong evidence that substantiates our conjecture about IRQ handling and alignment, along with a practical proposal to improve performance significantly with little to no negative side effects. The paper is wrapped up with a summary and conclusion in Section 6.

2 BACKGROUND

2.1 Kernel- vs. User-Level Networking

Traditionally, most of the network protocol stack is processed inside the operating system kernel. The kernel normally ingests memory-mapped buffers of network data from the NIC after receiving a notification via an IRQ, and then passes these buffers through the link (typically Ethernet), network (IP), and transport (such as UDP or TCP) protocol layers. The buffers ultimately become part of a queue called *socket buffer* [5] (one per transport instance), from where the data is made available to applications through standard system calls that provide specific semantics.

Commensurate with the increase in link transmission speeds and NIC capabilities (now reaching 100s of Gbps), kernels, such as Linux, have been continually improving the performance of their

network stacks. This includes efforts to streamline the network stack, such as reducing data copies and avoiding interrupts when possible, scaling multiple transmit (TX) and receive (RX) queues to multiple processor cores, and efforts to optimize communication and cooperation between kernel and application, such as the `kqueue` [21], `epoll` [26] and `io_uring` [4] system interfaces.

A recent line of work, on the other hand, seeks to abandon the kernel network stack, which is often considered dated and inefficient, in its entirety. This work includes library-based network protocol implementations that are executed in the context of user-level application processes. Depending on the level of kernel involvement in network processing, the term *kernel-bypass* is used to denote minimal involvement (or none) of the kernel in the data path. In this operating mode, a user program takes control of a NIC or a virtual function exposed by the NIC, either through hardware-assisted virtualization, or through special provisions made in the kernel driver for the NIC. In both cases the user-level application has direct access to a subset of the TX and RX queues.

When network traffic is processed in user space, it is often the case that the user-level process must run in a constant polling loop on dedicated CPU cores. This requirement stems from the fact that typical platforms and operating systems available today provide no mechanism for directly routing interrupts to user-level processes. Depending on the exact execution model chosen, it is sometimes possible to designate only a subset of cores to run in polling mode, and to rely on user-level communication for the rest of the application to receive data. Such a model is nevertheless very different from the fundamentally asynchronous interrupt handling in the kernel.

2.2 Components

Data Plane Development Kit (DPDK) [24] is a popular framework for implementing customized user-level (kernel-bypass) network stacks. A program using DPDK handles data in a polling loop with direct memory-mapped access to the NIC's TX/RX queues, and processes and transmits data in the same polling loop. DPDK provides abstractions and common software infrastructure required by typical user-level network stacks. For example, its *poll mode drivers* expose a common interface for all mechanisms employed to access NICs in user space, regardless of hardware support and driver-specific access methods. The *environment abstraction layer* hides platform-specific details of memory allocation, memory mapping, thread creation, and the polling loop that is run on each designated core. DPDK also includes libraries for memory pool management and synchronization primitives. Although DPDK was originally intended for raw packet processing on software switches and routers, the wide coverage of its abstractions has brought about a plurality of user-level network stacks both in research and industry.

The *eXpress Data Path* (XDP) [13] subsystem is an emerging Linux kernel mechanism to attain the benefits of DPDK-like packet processing frameworks within the kernel. It leverages the *extended Berkeley Packet Filter* (eBPF) [6] virtual machine in the kernel to process network traffic right after its arrival in the NIC driver and before entering the kernel network stack. Even though this does not bypass the kernel completely as DPDK can do, it eliminates the need for polling mode execution and dedicated CPU cores, while still being much more efficient compared to earlier methods to redirect packets to custom processing logic without DPDK, such as packet capturing. However, due to safety limitations in eBPF, XDP itself is not suitable for implementing a complete network stack within the kernel. Recent Linux kernel versions have added support for a new socket family, `AF_XDP`, that diverts network traffic to user-level applications using the infrastructure of XDP. It is supported by DPDK as an alternative to its native drivers, since the effect is similar.

Other options for user-level networking include PacketShader [12], which enables not only user processing but offloading of highly parallel networking logic (e.g. forwarding) to a GPU; and PF_RING [30], leveraging virtual functions support in hardware to perform high-speed packet capturing.

2.3 User-Level Network Stacks

Since its release, DPDK has seen wide adoption in the industry to realize user-level networking without building the entire software infrastructure from scratch. F-Stack [44] is a port of the FreeBSD network stack to DPDK, in the hope of achieving performance benefits from user-level networking without relying on and maintaining a customized TCP/IP stack. F-Stack includes the complete source code of the FreeBSD kernel, but replaces all functions pertaining to multi-threading, synchronization, and device drivers with empty stubs. It executes the DPDK polling loop along with the network stack and the application code by emitting packets from the polling loop through the FreeBSD network stack, and invoking an application-defined callback after processing. F-Stack adapts the `kqueue` system call (and a compatibility layer for `epoll`) only for retrieval of events after the application-defined callback is invoked by the network stack. Because this execution model is different from the asynchronous in-kernel model, the existing synchronization stubs cannot be simply ported to user-space threading primitives and F-Stack is effectively limited to single-threaded execution. To fully utilize a multi-core system, an F-Stack application must be designed to run as multiple processes without a default shared address space.

On the other hand, the *Linux Kernel Library* (LKL) [25] is somewhat more compatible with multi-threaded applications. LKL adapts the Linux kernel (including its network stack) into a user-space library, but in order to satisfy in-kernel synchronization constraints, LKL implements a single global lock that allows kernel code to be executed by only one thread at any moment in time. This restriction allows applications to make use of multi-threading, but the network stack is still effectively single-threaded. Consequently, even though LKL's network stack can be used with DPDK without the same set of constraints as F-Stack, its performance is severely limited.

In addition to industry, DPDK and user-level networking enjoy increasing popularity in the research literature. Shenango [32] and its successor Caladan [9] are recent user-level network stacks built on DPDK. Both add significantly to the default execution model of DPDK. Instead of dedicating all cores to one application for polling, in both Shenango and Caladan, only one core is needed to run the DPDK polling loop, which is termed *iokerneld* process. Their work includes a user-level threading runtime along with a fast-path inter-process signal delivery mechanism exposed through a custom kernel module. This allows scheduling to be handled largely by *iokerneld* by waking up application threads only when events of interest are found during a polling iteration. The majority of the network stack, other than the central polling loop, runs in the application, similar to other DPDK-based network stacks. Each application receives data with its own polling loop and, unlike interrupt-driven delivery in the Linux kernel, the arrival of network traffic does not preempt a running application thread. Although the Shenango and Caladan work is mainly focused on scheduling and its evaluation, it is built upon the assumption that user-level network stacks have inherent performance advantages and the papers present extraordinary performance gains over vanilla Linux.

Various other user-level networking approaches exist. Onload [47] is an example based on `AF_XDP` or a kernel-bypass mechanism specific to Xilinx NICs. Seastar [39] is an asynchronous programming framework for C++ and includes a network stack primarily targeting DPDK. mTCP [17] can switch between a number of user-level networking backends, including DPDK and PacketShader. Additionally, there are library protocol stacks decoupled from underlying operating systems or user networking mechanisms, such as lwIP [40]. For specific application scenarios, there are attempts to completely replace the established TCP/IP network stack with novel proposals implemented in user space, such as eRPC [18]; or rearchitected operating systems, such as Snap [28] and Demikernel [48], both of which lie outside of the direct scope of this work. We are not aware of any other mature

and openly available user-level TCP/IP network stack that eclipses the ones described above in performance or functionality.

2.4 Applications

User-level network stacks are often targeted at high-performance server applications. F-Stack includes both Nginx [29], a well-known and popular HTTP server, and Redis [35], an in-memory database engine, in its repository as sample use-cases. The F-Stack team also claims to have deployed F-Stack as the network stack for their DNS service. Both Shenango and Caladan include a port of Memcached [8], which is a widely deployed in-memory key-value store that is often used as caching daemon to store the results of dynamic remote queries. Seastar's kernel-bypass networking optionally powers ScyllaDB [38], a real-time database engine developed by the same team. In all of these cases, the application requires a complete Layer 4 protocol stack.

On the other hand, user-level processing of network packets is also a popular technique in software switching and routing. In this scenario, user-level processing frameworks such as DPDK or `AF_XDP` are used only as a way of efficiently capturing packets, but network traffic does not terminate at an application running above the network stack. The functionality includes minimal Layer 4 processing, if any, and most logic operates directly on raw Ethernet frames or IP packets. For example, NetVM [14] leverages the flexibility of DPDK's user-level processing to implement Network Function Virtualization [7]. Rubik [23] seeks to simplify the programming of network middleboxes by designing a new special-purpose programming language that targets DPDK as its packet processing infrastructure. When hardware offloading is involved, with or without application-level networking, DPDK-based solutions are also often used as state-of-the-art "best-case scenario" for software processing [15, 33].

It is important to note that this paper focuses on the server application aspect of user-level and specifically kernel-bypass networking. While the findings presented here may also apply to other scenarios, for example, middleboxes that parse application-level traffic before forwarding, such scenarios are not evaluated as part of this work.

3 NETWORK STACK OVERHEAD

System software directly affects application performance through the overhead of system services, such as I/O. It can indirectly affect application performance by organizing execution components, which makes application software run more or less efficiently, for example through scheduling. Changes in system software design might cause both of these types of effect. Then, in a slight generalization of Amdahl's Law [2], the question arises how the design changes actually lead to the overall performance difference. For example, for an N-fold increase in performance, $\frac{N-1}{N}$ of the overhead must disappear. In other words, assuming the existing and unchanged functionality and implementation of a given application, what portion of its total execution overhead is attributable to system services and how much of this overhead can be eliminated?

3.1 Performance Model

In addition to experimental observations that document application performance before and after design changes, any conjectures about performance behaviour should be corroborated by comparing the breakdown of execution overhead of the base case with the contender. Breaking down execution overhead and attributing performance improvements is not straightforward. The following simple model is proposed to aid in this task. Application throughput performance can be measured as

$$QPT = queries/time \tag{1}$$

The average CPU resource utilization is given by

$$CPT = cycles/time \tag{2}$$

IPC is a well-known metric describing how efficient a superscalar processor pipeline can process a particular workload:

$$IPC = instructions/cycle \tag{3}$$

In addition, IPQ is used here to capture how many instructions are executed for each application-level query. This metric is used as an estimate for functionality or algorithmic efficiency, i.e., the amount of processing that is done for each application query on average:

$$IPQ = instructions/query \tag{4}$$

All these metrics can easily be measured using hardware performance counters during an experiment. Furthermore, with tooling like the Linux Perf subsystem [42] it is straightforward to distinguish between user and kernel execution, as well as between main application and system libraries. Most importantly, these metrics are related to each other. It is quite trivial to see that

$$QPT = \frac{CPT \times IPC}{IPQ} \Leftrightarrow \frac{QPT}{CPT} = \frac{IPC}{IPQ}. \tag{5}$$

Basically, this model and the right-hand side of Equation 5 illustrates that an increase in performance (QPT) or efficiency (QPT/CPT) must be accompanied by an increase in IPC or a decrease in IPQ or both. A decrease in IPQ could be caused by an algorithmic improvement or by reducing functionality. An IPC increase typically means fewer stalled cycles due to improved efficiency of the processor pipeline, which could be caused by increased cache hit rates (including page translation and branch prediction) or similar effects.

Note that time is not strictly needed and the model could also use total queries and total cycles. However, application performance is typically described as a throughput rate. Without reference to time, a low-performing system at low utilization would be indistinguishable from a high-performing system at high utilization.

The attainable throughput does not capture the latency behaviour of an application. Normally, latency behaviour is described by cross-referencing high-order latency percentiles with throughput performance. The resulting graph typically shows an inflection point where a system transitions from underloaded to overloaded mode with queues building up. An experiment configuration near this inflection point can thus be used to attribute execution overhead using the given model.

3.2 Performance Assessment

The performance model is used for basic observations pertaining to the cost breakdown of I/O-heavy server applications. The particulars of the hardware and software setup for all experiments are given at the beginning of Section 5. The experiments reported here are set up with closed-loop clients that saturate the server at effectively 100% CPU utilization on all relevant server cores. All CPU cores are configured to operate at their maximum fixed frequency, but without turbo-boost. This eliminates the need to report CPT. Note that the results presented in this section show only individual representative data points, because they are primarily used as illustration and motivation for the proposals in Section 4, which in turn are evaluated in Section 5.

3.2.1 Memcached / F-Stack. The first scenario compares Memcached in a default Linux setup with a Memcached version that uses F-Stack/DPDK for user-level networking. Due to the nature of Memcached and F-Stack's limitations (cf. Section 2.3), this experiment is run using only a single worker core. The results are shown in Table 1 with 'Vanilla' denoting the default Linux setup. The overall performance is measured as QPT in queries/second. IPQ and CPU-cycle measurements are

Table 1. Memcached: Vanilla vs. F-Stack (1 core)

	QPT	IPQ			Cycles		
	(T=1s)	App	Sys	Total	App	Sys	IPC
Vanilla	84124	1905	18512	20417	10.6%	89.4%	0.64
F-Stack	106468	1930	20579	22509	10.2%	89.8%	0.89

Table 2. Nginx: Vanilla vs. F-Stack (8 cores)

	QPT	IPQ			Cycles		
	(T=1s)	App	Sys	Total	App	Sys	IPC
Vanilla	508828	5749	19245	24994	24.3%	75.7%	0.59
F-Stack	647441	6330	18037	24367	32.9%	67.1%	0.73

divided into an *App* and *Sys* part denoting overhead in the application vs. the rest of the system (libraries and kernel). While it is difficult to determine a further breakdown of the system part, it can be safely assumed that the vast majority of the system overhead arises in the network stack. It is worth pointing out that only about 10% of Memcached's overhead is actually attributable to Memcached code itself.

Replacing kernel networking with F-Stack results in an overall performance increase of 27%. The IPQ of F-Stack is slightly higher than the vanilla Linux kernel by around 10%, which is likely caused by extra shim functionality added during the porting of Memcached. However, a dramatic IPC increase by 39% compensates for the added overhead and leads to the substantial overall performance improvement.

3.2.2 Nginx / F-Stack. Another scenario studied here uses Nginx, because it operates in multi-process mode and can thus utilize F-Stack in a multi-core setting. The results in Table 2 show a similar overall performance improvement of 27%. It is worth noting that more overhead is attributed to the actual application when F-Stack is in use. The integration of Nginx and F-Stack is by dynamically redirecting I/O system calls, which causes the application's IPQ to slightly increase. Using the Linux Perf subsystem to attribute overhead to different components of a user-level application is neither trivial nor completely precise, because it can only rely on symbols exposed by the compiled binary. However, the overall IPQ is very similar between kernel and F-Stack, and the performance improvement is primarily caused by the IPC increase.

3.2.3 Memcached / Caladan. The Caladan research proposal [9] suggests an approximately 11-fold performance increase for Memcached resulting from Caladan's user-level network stack compared to vanilla Linux[1], even when effectively not using the actual scheduling proposal central to Caladan. The next scenario attempts to reproduce and break down these phenomenal performance observations. In addition to using DPDK for general user-level access to TX/RX rings, Caladan implements a *directpath* feature, by which the NIC is programmed to directly place incoming frames in per-flow-specific buffers, which are consumed by worker threads. If this hardware-specific feature is not available, all network notifications are routed through a central component (IOKernel). While this does not add much to the processing path in terms of latency, it eventually presents a bottleneck. To eliminate this bottleneck, the experiment is restricted to 6 cores in total – for Caladan this means 1 scheduler and 5 worker cores. This is compared to the same vanilla Memcached setup as in the previous two subsections – this time using 6 cores. The results are shown in Table 3. In

[1]Figure 4 on Page 290 in [9]: inflection points of solid green vs. solid blue line

Table 3. Memcached: Vanilla vs. Caladan (6 cores)

	QPT	IPQ			Cycles		
	(T=1s)	App	Sys	Total	App	Sys	IPC
Vanilla	577653	1783	17549	19332	9.89%	90.11%	0.69
Caladan	2108154	2103	5282	7385	28.5%	71.5%	0.97

Table 4. Memcached/Vanilla: 8 cores vs. 4+4 cores/NUMA

Cores	QPT	IPQ			Cycles		
	(T=1s)	App	Sys	Total	App	Sys	IPC
8	724077	1832	17570	19402	9.6%	90.4%	0.65
4 + 4	601494	1851	17672	19522	8.6%	91.4%	0.55

this scenario Caladan achieves a 3.65-fold performance improvement over the default Linux setup, which is a combination of an IPQ reduction by almost 2.6, while the IPC is increased by 41%. The IPC increase is comparable to the previous two experiments, but the IPQ difference is not.

Similar to F-Stack, Caladan is tightly integrated with the application during compile and link time, even more so with its user-level threading runtime. Therefore, it is difficult to cleanly factor out application processing, which results in an increased application IPQ reported here. Overall, though, IPQ is drastically reduced compared to kernel stack processing. A code inspection of the Caladan network stack shows that it implements only the bare minimum functionality for TCP/IP processing that is needed to run these experiments. For example, the stack's TCP component does not implement round-trip time (RTT) estimation or maximum segment size (MSS) adjustments, but instead runs entirely on constant values. Most importantly, it does not implement any congestion control. As outlined in Section 1, the re-implementation of network protocol processing for user space execution can be seen as both a customization strength and/or a maintenance weakness. In trying to compare the observations here with the 11-fold increase previously reported for Caladan, two further aspects need pointing out: 1) The exact configuration setup of the vanilla Linux system is not provided in the original paper. 2) The original experiments utilize 24 cores (or 48 hyperthreads) across two CPUs (12 cores each). While it is stated that NUMA is not considered, it is highly likely that NUMA effects have a detrimental impact on the interrupt-driven default setup, but do not affect a polling-based system like Caladan as much. Both these aspects are investigated further in the remainder of this paper.

3.2.4 Memcached / NUMA. This preliminary experiment investigates the effect of locality in general and non-uniform memory access (NUMA) in particular on network stack and application performance. It compares Memcached on 8 cores on a single socket with an equivalent dual-socket setup using 4 cores on each socket. The results are shown in Table 4. While IPQ numbers are largely unchanged between both setups, it is clear that reduced locality comes with a performance penalty of about 17%, which is caused by a corresponding reduction in IPC.

3.3 Discussion

The observations point to IPC as a key contributor to performance improvements due to user-level networking. IPC describes how efficiently the processor pipeline is utilized by avoiding stalls. Assuming that functionality and basic algorithms and data structures stay the same, low-level pipeline metrics need to be investigated. While we do not report all those findings in detail, we have determined that branch prediction has no discernible impact in this case. Dramatic effects of

system calls are not expected on modern processors, even when security mitigations are used [20]. Similarly, L1 and L2 cache utilization as well as translation lookaside buffer (TLB) miss rates do not show significant enough differences to explain the IPC gap.

For locality in general, we notice a limited impact that is further studied in Section 5. For example, the first experiment in Section 3.2.1 runs on a single core, so there are no locality issues by definition, yet the IPC increases by almost 40% with user-level networking. We conjecture that a side effect of user-level networking is responsible for a substantial fraction of the performance improvement: Continuously polling the NIC queues eliminates the need for asynchronous hardware IRQs, which otherwise distort the processor pipeline quite substantially. Even with standard interrupt moderation techniques implemented in the Linux kernel, we observe that most network packets are still accompanied by IRQs. Furthermore, a synchronous processing path typically has better locality than asynchronous execution. While locality does not matter much within the same NUMA domain, the observations in Section 3.2.4 show that NUMA communication overheads across domains do have an impact on IPC and performance.

4 NETWORK STACK ALIGNMENT

The observations presented in the previous section point to temporal and spatial alignment, namely synchronous processing and core locality, as drivers for performance improvements in the network stack. While the performance effects of IRQs are difficult to measure directly, they can be verified indirectly by reducing IRQs without changing other parts of the system. To illustrate and corroborate the performance potential of alignment, we present a number of proposals, with increasing practicality, to reorganize IRQ handling for the Linux kernel network stack. The final proposal is both practical and highly performant. User-level networking typically leads to a strongly aligned execution model by having to employ continuous polling. The ultimate objective of this work is determining how much of the corresponding efficiency gains can be obtained while using the comprehensive and mature kernel stack and without requiring the explicit and dedicated resource allocation necessary for kernel-bypass approaches. The presentation of these schemes is focused on RX interrupt handling, because it has a much larger effect than TX interrupts (cf. Section 5.2.3).

4.1 IRQ Routing

Most modern platforms provide programmable interrupt controllers that can be configured through operating system mechanisms. In the Linux kernel, each IRQ number has a property called *affinity*, defining which CPU cores receive and potentially handle the corresponding interrupt. The respective CPU core then also typically executes the deferred portion of interrupt handling and protocol processing (termed *softirq* in Linux). Opinions differ among practitioners on the optimal strategy under various scenarios, with some advocating for *balancing* IRQs between CPU cores [19, 31, 43], while others believe that they should be *packed* onto a small number of dedicated cores [34], especially for latency-sensitive workloads.

4.1.1 IRQ Balancing. It is often recommended to balance total IRQ workload across CPU cores in order to achieve higher performance. The *irqbalance* [46] daemon automates this process by observing traffic generated by each interrupt source and directing the highest volume interrupts to a single unique CPU core each. However, there are two caveats: 1) The IRQ arrangement does not necessarily take into account the placement and scheduling of network-intensive applications, which results in less than optimal alignment. 2) Consequently, the very nature of dynamic interrupt assignment can lead to performance variations that make reproducibility difficult. As others have observed previously, disabling irqbalance and controlling interrupt routing explicitly is thus the

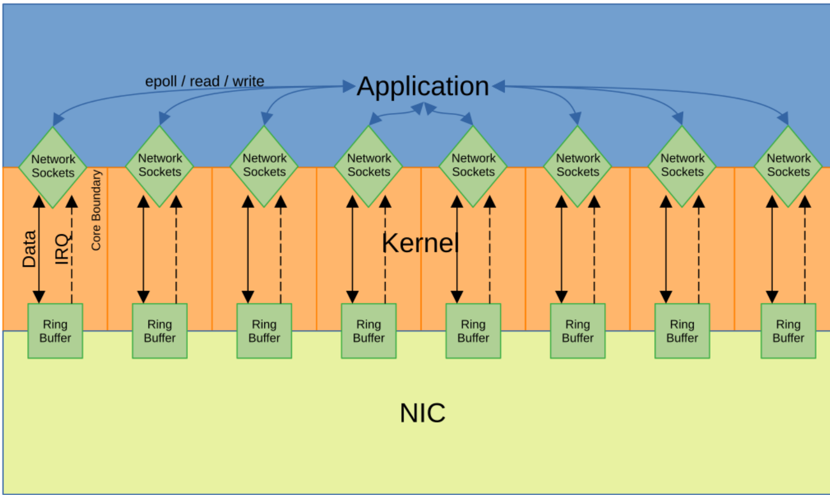

Fig. 1. IRQ Balancing

preferred approach for reproducible high-performance networking experiments, even though it is not always straightforward to accomplish in practice.

For an application deployment on N cores, a typical approach is configuring N RX and N TX queues and assigning one RX and one TX queue (via their respective interrupt) to each core. This setup forms the baseline for the experiments reported in this paper and is conceptually illustrated in Figure 1. The NIC asynchronously notifies the kernel about incoming traffic. After network protocol processing in a softirq kernel thread, another asynchronous notification informs the application that data is ready. The softirq and application threads do not necessarily execute on the same core. We have experimented with Linux mechanisms that attempt to improve locality on the data path between NIC and application threads, such as receive flow steering (RFS) as well as thread pinning in combination with the `SO_INCOMING_NAPI_ID` or `SO_INCOMING_CPU` socket options. However, none of these mechanisms significantly shifts the baseline, at least not when all cores are in the same NUMA domain. Figure 1 shows the most optimistic case of perfect spatial alignment.

4.1.2 IRQ Packing. Linux employs an interrupt mitigation technique termed *New API* or NAPI [37]. After an interrupt is delivered, this interrupt is temporarily masked. The kernel enters polling mode and retrieves all available network packets from the corresponding RX ring. The specifics can be configured to some extent by kernel parameters. However, after completing a polling episode, the corresponding interrupt is re-enabled. Most importantly, interrupts can arrive while the application is still processing data received previously. To achieve better temporal alignment, an *IRQ packing* scheme is proposed, instead of distributing the interrupt load. The objective is forcing the kernel into almost perpetual polling mode and thereby eliminating a large fraction of interrupts. If nothing else is modified, then the resulting performance changes illustrate the cost of handling interrupts traded off against the potential impact of reduced locality.

Interrupts are assigned to a minimal set of dedicated cores, while application threads are restricted to a set of different cores. The number of NIC queues is also set to the number of dedicated IRQ handling cores. The IRQ packing scheme is illustrated in Figure 2. Ideally, it uses just enough cores to handle all network traffic while saturating those cores. Although this does not spatially align the application with the network stack, IRQ packing makes interrupt mitigation extremely effective

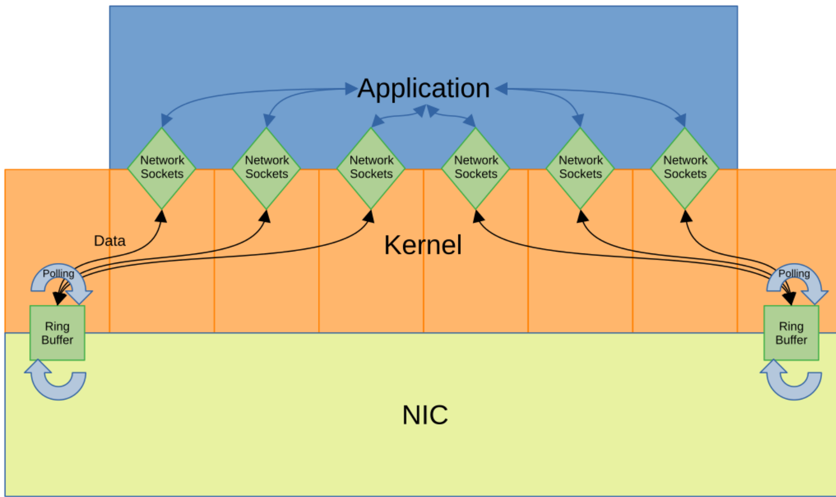

Fig. 2. IRQ Packing

and suppresses most interrupts. The resulting performance improvements, shown in Section 5, indirectly confirm the conjecture that IRQ handling has a significant performance impact. However, IRQ packing is often difficult to configure in practice, since it relies on kernel settings that are hard to adapt dynamically. Most importantly, CPU cores can only be allocated in whole integers and need to be fully saturated by network traffic to effectively suppress hardware interrupts. This creates a bottleneck. Furthermore, proper resource management would require some form of global adaptive core allocation in response to workload dynamics, which is invariably slow and brittle.

4.2 Network Polling

While IRQ packing illustrates the overhead of IRQ handling, it cannot be regarded as a general-purpose scheme, as explained in the previous section. In the vanilla IRQ balancing scheme, on the other hand, IRQ arrivals compete asynchronously with network and application processing for the same set of cores. In an ideal scenario, while an application is busy processing existing requests, no further data (and thus no interrupts) should be delivered until the application is idle again. However, since packet arrivals are fundamentally asynchronous and network traffic is not exclusive to a particular application, there is typically no coordination between application execution and IRQ handling in the kernel.

In contrast, kernel-bypass and user-level network stacks put the application in charge of the entire network stack processing (cf. Section 2). Interrupts are disabled and the application coordinates execution by alternating between processing existing requests and polling the RX queues for new data. Modern programmable NICs address the exclusivity problem by allowing fine-grained control over which network traffic arrives in which RX queue. We present two proposals for emulating this execution pattern with the Linux kernel stack. The principle is shown in Figure 3.

The Linux kernel already contains mechanisms to promote polling-based packet reception. One such feature, though disabled by default, is `sysctl net.core.busy_poll`, sometimes used in combination with the `SO_BUSY_POLL` socket option. When this parameter is set, the kernel's NAPI component enters a short busy polling period, as defined by the value of the parameter, when an application uses any of the I/O multiplexing calls `select()`, `poll()`, or `epoll_wait()` and no events are immediately available. If network packets are received during polling, network protocol

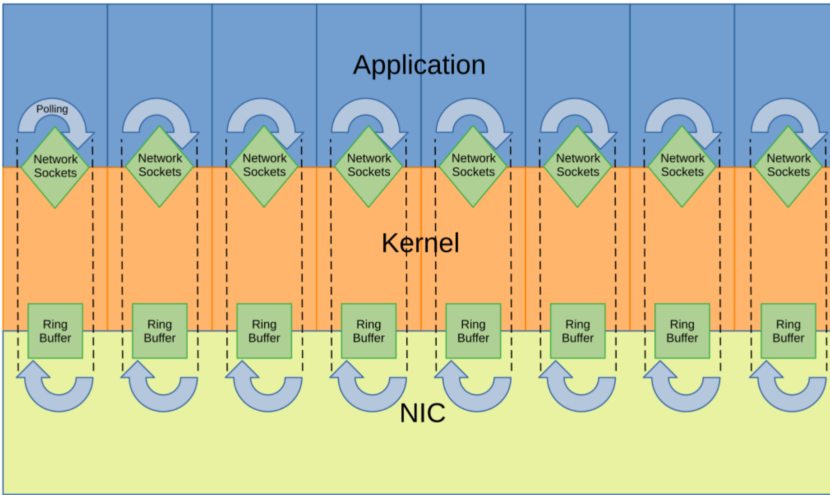

Fig. 3. Network Polling

processing is performed in the same synchronous execution path, resulting in the desired cooperation between the application and the network stack similar to user-level networking. However, this mechanism by itself does not eliminate asynchronous interrupt handling sufficiently. While the kernel suppresses interrupts during the busy-polling episode, interrupts are still immediately re-enabled before control returns to the application. Thus, while the application is processing the previously received data, IRQs continue to arrive and distort the application's execution.

4.2.1 IRQ Suppression. The first approach to network polling is suppressing most interrupts via NIC-based interrupt coalescing. Most NICs provide timer- and/or counter-based configuration parameters to request delayed interrupt generation. These parameters are accessible through the `ethtool` program on Linux and can be set to high values to effectively suppress interrupt generation. In combination with the existing kernel busy polling mechanism, this results in the majority of packets being received via polling, while interrupt generation is strictly limited. However, this approach is not very robust and requires meticulous tuning for each application and in fact, each workload situation. Otherwise, the lack of interrupts during a low-rate arrival phase causes delays before packets are retrieved and the resulting wait times lead to reduced utilization and higher service latencies. However, this scheme does show that there is at least a possibility to realize a high level of coordination between application and network processing without abandoning or modifying the kernel network stack.

4.2.2 Kernel Polling. The missing piece for improved coordination between application and network stack is gaining control over the masking of IRQs. A minor kernel modification is proposed to maintain IRQ masking while an application is processing previously received data. This modification works in tandem with the kernel's busy polling mechanism described above. Instead of re-enabling the respective interrupt(s) as soon as `epoll_wait()` returns from its NAPI busy loop, the relevant IRQs stay masked until a subsequent `epoll_wait()` call comes up empty, i.e., no events of interest are found and the application thread is about to be blocked. A new IRQ inhibition flag is added to NAPI instances that correspond to RX queues. This kernel flag is set in the return path of `epoll_wait()` as long as the application has data to process. When the flag is set, NAPI advises the NIC driver against re-enabling hardware interrupts. With this change, no network interrupt

```
 1: procedure EP_POLL(ep)                                   ▷ epoll_wait() implementation
 2:     avail ← are_events_available(ep)
 3:     loop
 4:         if avail then                                   ▷ Return path
 5:             events ← get_events(ep)
 6:             new_napi_id ← events[0].napi_id
 7:             if new_napi_id ≠ ep.last_napi_id then
 8:                 unmask_interrupts(ep.last_napi_id)
 9:             end if
10:             ep.last_napi_id ← new_napi_id
11:             mask_interrupts(ep.last_napi_id)
12:             return events                               ▷ Copy to user
13:         end if
14:         avail ← do_busy_poll(ep)
15:         if avail then
16:             continue
17:         end if
18:         unmask_interrupts(ep.last_napi_id)
19:         sleep_until_notified(ep)
20:         avail ← are_events_available(ep)
21:     end loop
22: end procedure
```

Fig. 4. Kernel Polling (Pseudo-code)

is delivered while the application is busy receiving and processing data, and interrupts are only used as a fallback when the application is idle. The modification is illustrated with pseudo-code in Figure 4 and implemented by about 30 lines of kernel modifications at the boundary between generic event polling and NAPI code.

The resulting execution model mimics the execution model of typical user-level network stacks and does not add any requirements compared to user-level networking. In fact, it is slightly better, because it can resort to blocking and interrupt delivery, instead of having to continuously busy-loop during idle times. In order to maximize efficiency, an ideal setup has a 1:1 mapping between application threads and RX queues (NAPI instances), as illustrated in Figure 3, so that no ambiguity exists on which queue should be polled during the busy loop. Such a mapping can be achieved by suitable application design purely in user space. All that is needed is grouping and dispatching newly accepted connections to threads according to the `SO_INCOMING_NAPI_ID` flag. Thread affinity to cores is not required.

4.3 Generality and Possible Adoption

As pointed out previously, neither IRQ packing nor IRQ suppression are suitable for potential adoption and deployment in production environments. However, kernel polling seems to be general and nimble enough to pass muster. Similar to user-level networking, kernel polling places RX queues under the control of an application. If such an RX queue receives cross-traffic for other applications, such traffic might be slightly delayed while IRQs are masked. Also, if a controlling application would not call `epoll_wait()` after each round of processing, IRQs may not be re-enabled and no new traffic can be delivered via a particular RX queue. However, in contrast to user-level networking,

the actual processing path of cross-traffic is not changed and any cross-traffic is processed by the kernel and directly consumed by its respective application process. Consequently, kernel polling with interrupt fallback functionally dominates the user-level polling scenario typically embodied by user-level networking – by being more flexible and less intrusive.

While the kernel modification proposed here is currently a proof-of-concept and not yet complete for production use, there is a clear path towards a production-grade kernel polling scheme and possible adoption. As mentioned above, modern programmable NICs can alleviate cross-traffic concerns due to fine-grained flow classification and routing to specific RX queues. A fallback technical approach would use a kernel timeout set on the return path from `epoll_wait()`. If necessary, the timeout re-enables interrupts regardless of the application's (mis)behaviour. An administrative approach towards production-level security and robustness would encode the interrupt masking request in a privileged socket option or `epoll_wait()` flag, only available to threads with a suitable capability [11]. Aside from open-sourcing all code and experiments described here[2], we intend to submit a corresponding patch for possible adoption in the Linux kernel.

5 EVALUATION

5.1 Experimental Setup

5.1.1 Hardware. The evaluation is performed on a server with dual-socket octa-core Intel Xeon E5-2680 CPUs (NUMA setup, 16 cores / 32 hyperthreads total). The server is equipped with 64 GiB of RAM (32 GiB per NUMA node), enough for all experiments reported in this paper, and a Mellanox ConnectX-3 10 GbE network controller. In all experiments, Turbo Boost is manually disabled to rule out any unpredictable effects, and both CPUs run at their maximum non-Turbo Boost frequency of 2.7 GHz. Hyperthreading is avoided by scheduling threads only on the respective first hardware thread of each core. An additional 7 identical machines are used as clients to generate load.

5.1.2 System Software. All machines in the experiments are set up with Ubuntu 20.04, with updates up to Q4 2022. User-level network stacks, such as Caladan, require an older version of the Linux kernel. As a result, these experiments (primarily in Section 3) are performed on kernel version 5.4, provided by the official Ubuntu 20.04 repositories. Since these stacks bypass the kernel and require dedicated CPU cores and/or implement their own scheduling, the older kernel version is not expected to cause any distortion of performance observations - advantageous or disadvantageous. All other experiments are performed on kernel version 5.15, enhanced by the kernel polling patch described in Section 4.2.2. The kernel is booted with the boot setting `mitigations=off`, which disables various mitigations for older CPUs' security vulnerabilities. In addition, automatic NUMA balancing is turned off by setting `sysctl kernel.numa_balancing=0` to avoid interfering with the intended thread placement during experiments.

Resource usage is tightly controlled through IRQ routing and thread affinity, and monitored via the Linux Perf subsystem and other kernel reporting. In particular, all softirq and other network processing is done on those cores that handle the hardware interrupts and/or those that are designated as application cores. For experiments labelled 'Vanilla', a static balanced IRQ assignment is performed with each core mapped to one dedicated RX and TX queue on the NIC, in order to rule out inconsistencies in the default assignment or interference from dynamic IRQ assignment schemes such as *irqbalance*, which is disabled. Normally, this change results in a slight performance increase for the vanilla kernel due to better locality compared to a true default setup. For fairness with user-level network stacks, only those cores that are specified for a particular scenario are assigned IRQ workload during kernel-based experiments. For example, in an experiment labelled as

[2]https://cs.uwaterloo.ca/~mkarsten/netstack/

8 cores, IRQ handling is only allowed on those 8 cores, although specific work assignment within the set of allowed cores can be different depending on the scheme being tested.

5.1.3 Benchmark Software. Memcached is an attractive target application for benchmarking network stacks and other systems software. It is a production-grade and widely used tool, but ultimately a lightweight application that exposes the performance and efficiency of the underlying runtime system stack. Memcached 1.6.9 is used for all experiments using the Linux kernel's network stack. This is the earliest version with support for NAPI locality based on the `SO_INCOMING_NAPI_ID` socket option, which is required for kernel polling (Section 4.2.2).

The benchmarks reported for Caladan and F-Stack in Section 3 use Memcached 1.5.9, which is part of the Caladan software repository. We have ported Memcached to F-Stack and during the process have found that it is necessary to revert most of the changes in 1.6 in order to support user-level networking. For example, the `SO_INCOMING_NAPI_ID` socket option is irrelevant and must not be used when Memcached is running on a user-level network stack. For this reason and to avoid mixing too many versions, Memcached 1.5.9 is also used for the F-Stack experiments. Any porting effort to Caladan or F-Stack consists of extensive code modification and refactoring that, at the very least, fully rewrites the main event loop of Memcached. This results in much greater differences than those between Memcached 1.5.9 and 1.6.9.

Load is generated with *Mutilate* [22], a well-established benchmark client for Memcached, using 8 threads (cores) on each of the 7 client machines, and creating 20 connections per client thread for a total of 1120 connections.[3] The experiments use Mutilate's synthetic recreation of the Facebook "ETC" workload described in the literature [3] with 1,000,000 records.

Section 3 also presents experiments based on Nginx 1.16.1, since it is the latest version supported by F-Stack, and part of the F-Stack software repository. *Wrk* [10] is used for load generation. It is used with 1000 concurrent connections and repeatedly requests a small file located in RAM[4].

5.2 Alignment

The first line of inquiry studies and documents the performance of the various alignment proposals presented in Section 4. An initial overview and breakdown is provided in Table 5, which shows the sustained throughput performance and IPQ/IPC breakdown, similar to the presentation in Section 3.2, for a representative closed-loop experiment for each of the proposals. The result for the vanilla setup from Table 4 is repeated in Table 5 for reference. For the IRQ packing scheme, a configuration of 2 interrupt-processing cores and 6 application cores is set up for this experiment, as 2 is the maximum number of cores that can be fully loaded by interrupt handling generated by this particular workload. IRQ suppression parameters are also manually tuned for maximum throughput in this particular experiment.

It is obvious that all alignment proposals result in a substantial performance increase over the vanilla configuration. Moreover, it can be observed that, similar to the observations for F-Stack reported in Section 3.2, most of the performance improvement can be attributed to an increase in IPC, which closely mirrors the difference in throughput achieved by the respective scheme.

However, maximum throughput is not sufficient to characterize the performance of I/O-heavy server applications. To fully assess the overall performance of each of the proposed schemes, a second experiment is used to assess the resulting tail latency behaviour in relation to throughput. Clients generate a fixed rate of service requests in open-loop mode and the experiment measures the 99th percentile latency achieved for the resulting throughput. Figure 5 shows this tail latency on the Y-axis (logarithmic scale) for varying throughput rates for all alignment proposals. Each data

[3]The number of connections is chosen to avoid excessive cache misses on our hardware with a small LLC; see Appendix A
[4]As a static part of Nginx's configuration, loaded into RAM on start.

Table 5. Memcached: Alignment Proposals, 8 cores

	QPT (T=1s)	IPQ			Cycles			pkts/ irq
		App	Sys	Total	App	Sys	IPC	
Vanilla	724077	1832	17570	19402	9.6%	90.4%	0.65	1.05
IRQ Packing	847669	1981	17549	19530	10.7%	89.3%	0.77	22.1
IRQ Suppression	967675	1842	17123	18965	11.7%	88.3%	0.85	262
Kernel Polling	947021	1853	16716	18569	11.9%	88.1%	0.82	15.6

point shows the average result of 20 independent repetitions of the same experiment. The resulting standard deviation is shown with error bars. It is again very apparent that all alignment proposals result in better performance compared to the vanilla kernel. In particular, they are able to maintain a lower tail latency up to higher rates of throughput. However, the curves differ significantly for the different schemes. This and other details are discussed next for each alignment scheme.

5.2.1 IRQ Packing. IRQ packing maintains a very competitive tail latency, but its throughput capacity is a fair bit lower than IRQ suppression or kernel polling. While IRQ packing promotes polling-based network processing, this processing is still performed in softirq kernel contexts, and is opportunistic in nature. At high load, IRQ packing increases the number of packets that are received per interrupt to about 22. Furthermore, IRQs do not distort application processing, but are delivered to dedicated cores that do not perform much other work. IRQ packing does not have any spatial alignment between network processing and application, yet performance increases significantly. Overall, this corroborates the conjecture that IRQ handling is a significant source of network processing overhead, irrespective of locality.

IRQ packing shows good potential to improve network processing performance without any kernel modification. Unfortunately, its requirement of fully loading an integer number of cores severely limits the possibilities of adopting it as a general-purpose mechanism. By definition, network processing must be a bottleneck in IRQ packing, which most likely also contributes to the limited throughput performance shown in Table 5 and visible in Figure 5.

5.2.2 IRQ Suppression. IRQ suppression results in an impressive 33% throughput increase over the vanilla version, with a corresponding improvement in IPC, which confirms our basic conjecture about IRQ handling. Compared to vanilla and IRQ packing, there is a slight decrease in IPQ. In polling mode, a part of the asynchronous network processing logic is eliminated from the code path, which results in the slight IPQ decrease.

However, as pointed out before, IRQ suppression requires fine-grained manual tuning and even then, remains a fundamentally fragile mechanism. Its tail latency, while better than the vanilla case, is not competitive compared to the other two alignment schemes. In fact, the tail latency starts to grow much earlier than the saturation point where the system becomes overloaded. Furthermore, Figure 5 shows very high variations in the measured 99th percentile latency, even at relatively low load, which points to difficulties in coordinating between application and network stack. Given the current implementation of interrupt coalescing in NICs and the kernel, suppression parameters have to be chosen somewhat statically without taking into account application dynamics. The experiments here have used very aggressive suppression parameters as supported by `ethtool` and the NIC: `rx-usecs 65534 rx-frames 65534 tx-usecs 1024 tx-frames 256`. These parameters would have to be changed rapidly to accommodate dynamic traffic patterns. However, any chosen configuration implies an inherent trade-off between throughput capacity and tail latency. Based

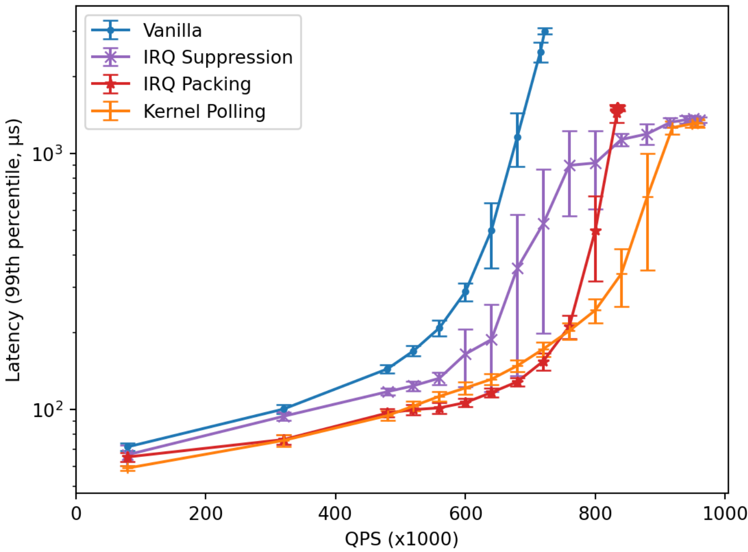

Fig. 5. Memcached: Latency vs. Throughput, 8 cores

on these observations, it is questionable whether IRQ suppression could be deployed in dynamic workload scenarios, especially when a consistent tail latency is as important as throughput capacity.

5.2.3 Kernel Polling. Kernel polling does not suffer from the issues that IRQ packing and IRQ suppression face, because the decision whether to poll or enable interrupts is made automatically based on the application's workload. Its performance is strong in both maximum throughput and tail latency, as evident by Table 5 and Figure 5. While it is difficult to compare throughput numbers for specific tail latencies with this methodology, it is clear that kernel polling outperforms the vanilla configuration by at least 30%.

Concerning tail latency, kernel polling is far superior to IRQ suppression. It does exhibit some tail latency variation shortly before reaching capacity, but this is most likely a normal queueing effect for systems close to capacity and is observed for all different schemes. However, the IRQ suppression scheme achieves a slightly better IPC and maximum throughput than kernel polling. The difference likely results from moderating both TX and RX interrupts when tuning for IRQ suppression, while kernel polling only disables RX interrupts whenever possible. Including TX interrupts in the scheme would require substantially more refactoring than the current RX-only approach, where code modifications are simple and non-intrusive. The performance difference caused by TX interrupts is only around 2% and only affects maximum throughput. This observation also demonstrates that TX interrupts only have a limited impact on performance, at least in the Linux kernel with default settings. TX path processing does not happen in response to TX interrupts, but typically in the sender thread's context or on the RX path (e.g., in response to received TCP ACKs). TX interrupts are primarily used to periodically recycle ring buffer entries and any sensible default configuration does not result in many TX interrupts. The interrupts reported for kernel polling in Table 5 are almost exclusively TX interrupts.

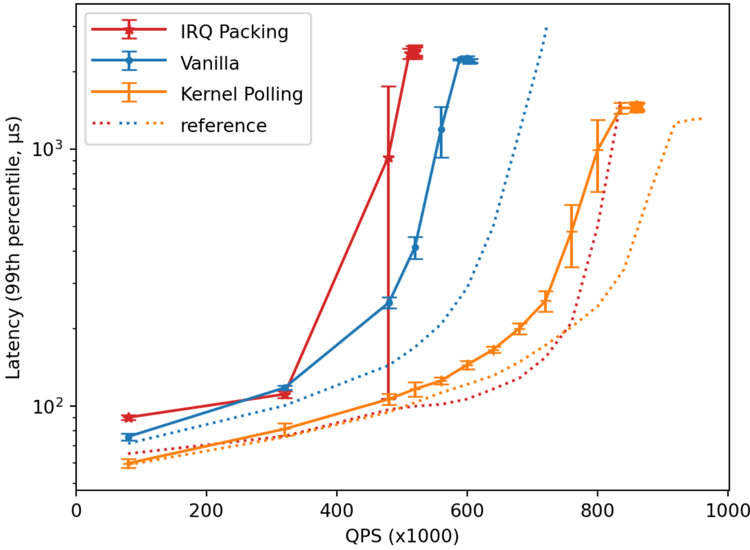

Fig. 6. Memcached: Latency vs. Throughput, 4 + 4 cores

5.3 Locality

A side effect of kernel polling is that all network processing code is guaranteed to execute on the same core, and in fact, the same task context as the application thread initiating the poll. This should result in better locality, which in principle should also contribute to performance positively. However, as examined in Section 3 and Section 5.2, locality effects seem to have limited impact, compared to the effects of reducing interrupts, as long as cores are in the same NUMA domain. On the other hand, the results in Table 4 indicate that NUMA does have an effect, so the single-domain observations are complemented by experiments across a NUMA domain boundary.

The latency-vs-throughput experiment from the previous section is repeated for 3 NUMA-based scenarios: For both vanilla and kernel polling, the application is spread across 4 cores each across two NUMA domains (4+4). For vanilla, as before, each core receives interrupts from one dedicated RX and TX queue on the NIC. For IRQ packing, the 2 cores serving IRQs are configured to be on the second NUMA node, while the 6 application cores are kept on the first NUMA node. Application threads are not pinned to individual cores, and the scheduler is allowed to decide the placement of each thread within the specified groupings. These configurations should result in maximum communication across the NUMA boundary for all schemes examined.

The results are shown in Figure 6 along with dotted lines showing the respective single-domain reference results from Figure 5. It turns out that IRQ packing suffers the worst NUMA penalty, bringing down its performance to less than the vanilla NUMA configuration - along with high latency variation near its saturation point. On the other hand, kernel polling retains very good performance due to its automatic locality and appears to incur a relatively smaller NUMA penalty than the vanilla configuration. To further evaluate the effect of NUMA, Figure 7 presents the closed-loop throughput of the vanilla kernel versus kernel polling for an increasing number of cores in each NUMA domain. One can observe that both vanilla and kernel polling exhibit near-linear

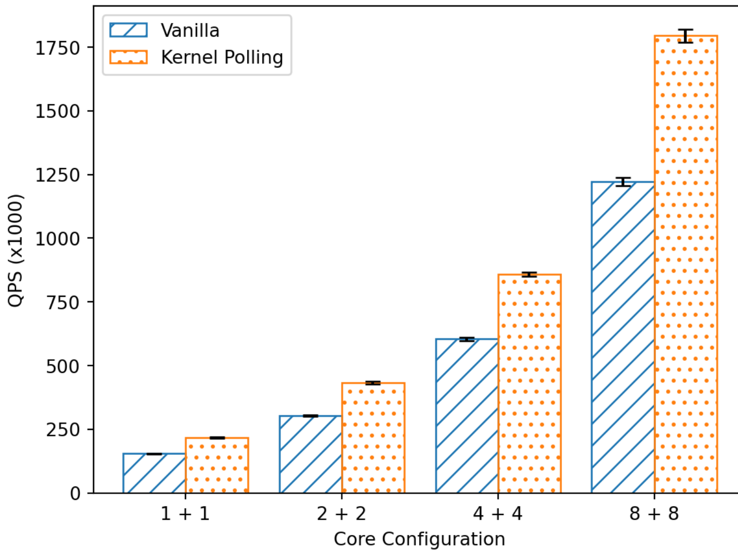

Fig. 7. Memcached: Closed-loop Throughput (NUMA)

scaling with more cores even under NUMA, with very consistent throughput among experiment runs. The relative gap between vanilla and kernel polling is largely constant at 43-46% for these experiments, compared to 30-31% in the single-domain case. These results are consistent with Figure 8 showing relatively constant IPQ and IPC for these configurations. Note that IPC still closely mirrors the performance difference, with a gap of 43% at 8 + 8 cores between vanilla and kernel polling. IPQ in general remains the same, with a similar slight decrease due to the elimination of the asynchronous processing path as described in the previous section.

When taken together, the results reported here demonstrate that locality does not play an important role within the same NUMA domain. However, NUMA overheads incurred by cross-domain communication can be substantial, as shown for IRQ packing. The vanilla Linux kernel with its internal packet routing and thread placement logic is largely successful in keeping NUMA overheads limited. However, kernel polling, as proposed in this paper, shows superior performance, and its automatic locality (cf. Section 4.2.2) further reduces NUMA overheads in the network stack.

6 CONCLUSION

The direct and indirect costs of asynchronous interrupt processing are identified as as a major source of overhead in the kernel network stack. As a secondary concern, locality matters, but it appears only significant when crossing NUMA domains. Several schemes are proposed and experimentally evaluated to improve the alignment between network stack and application, both temporally and spatially. While IRQ packing and IRQ suppression are not practical schemes and not meant as genuine proposals, they are useful analytical vehicles to corroborate the claim about IRQ handling, because directly measuring IRQ overhead is not possible at the system software level. Interrupt reduction is shown to be a key driver for increasing IPC and improving performance. The best-performing scheme, kernel polling, is a practical proposal and can be implemented with a small (~30 lines) and non-intrusive kernel change. Kernel polling increases throughput by up to

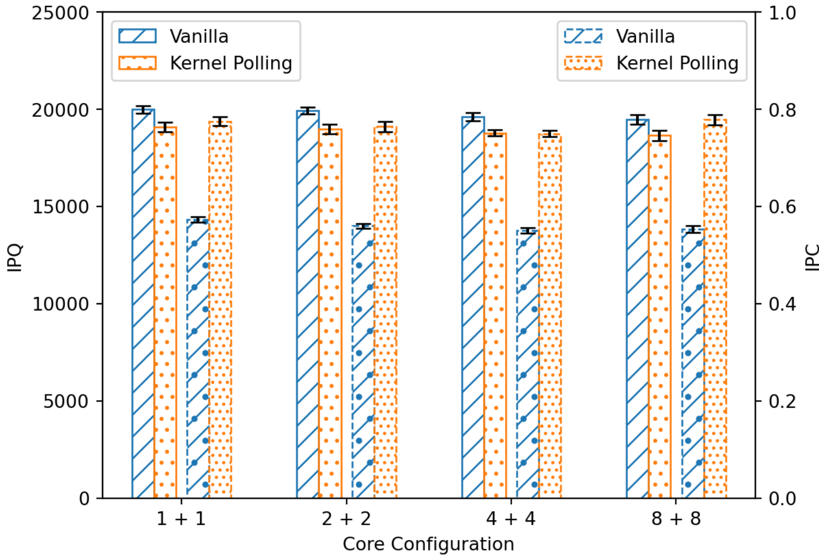

Fig. 8. Memcached: IPQ vs IPC (NUMA)

30% in an UMA and 45% in a NUMA configuration without compromising tail latency. It shows comparable performance to a comprehensive user-level stack, such as F-Stack. Since F-Stack's kernel bypass processing does not need traditional system calls for network I/O, this also indirectly verifies that system calls are not a significant source of overhead.

For kernel polling, the only requirements on the NIC are that its driver must support receive side scaling and dynamic interrupt moderation. Perfect alignment is guaranteed by executing the polling loop in the same application thread context with interrupts masked unless idling. Workload dynamics are handled automatically, with control handed over to the application itself, without the need for manually tuning configuration parameters. Compared to user-level networking, this method does not require reserving dedicated cores, nor pinning threads to cores. There is a clear path to adoption in production through capability-based permissions and/or a kernel timer to guard against misbehaving applications.

Aside from the particular proposals presented here, this work has important implications for research methodology. Since IRQ handling has a significant impact on network processing performance, it is important to properly document IRQ routing in experimental setups. By the same token, to truly understand the performance impact of design changes in system-level software, novel proposals must be compared to a competitive baseline setup. Last not least, for sanity checking, it is important properly attribute all relevant overheads to confirm that the results are plausible.

There are several avenues for future work arising from the findings reported here. First, it would be beneficial to subject our findings and the kernel polling improvements to a wider test vector in terms of hardware (server types, core count, NIC types) and application software. While this paper is focused on server-side network processing, it would be interesting to investigate whether kernel polling can lead to similar benefits in other application domains, such as software switches or middleboxes. This type of investigation would also verify the generality of kernel polling by considering different network protocols and different deployment scenarios, such as containers

and virtual machines. Memory alignment, especially in NUMA scenarios, can possibly be further improved by coordinating the scheduling of application threads in polling mode with ring buffer allocation in NIC drivers. In the long run, it is possible that kernel- and user-level networking converge in several ways. The kernel network stack offers many options for customization, either through XDP or eBPF. New transport protocols, such as QUIC [16], are increasingly implemented as user-level libraries. For user-level networking, it may be possible to deliver interrupts directly to applications to avoid continuous polling.

Acknowledgements

We would like to acknowledge support for this work from Huawei Canada and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Advanced Micro Devices, Inc. 3rd Gen AMD EPYC Processors with AMD 3D V-Cache Technology Deliver Outstanding Leadership Performance in Technical Computing Workloads. https://www.amd.com/en/press-releases/2022-03-21-3rd-gen-amd-epyc-processors-amd-3d-v-cache-technology-deliver-outstanding, 2022. [Online; accessed 2023-07-17].

[2] Amdahl, G. M. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (New York, NY, USA, 1967), AFIPS '67 (Spring), Association for Computing Machinery, pp. 483–485.

[3] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of SIGMETRICS* (2012), pp. 53–64.

[4] Corbet, J. Ringing in a new asynchronous i/o api. https://lwn.net/Articles/776703/, 2019. [Online; accessed 2023-07-17].

[5] Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux device drivers.* " O'Reilly Media, Inc.", 2005, pp. 528–531.

[6] eBPF Foundation. eBPF. https://ebpf.io. [Online; accessed 2023-07-17].

[7] European Telecommunications Standards Institute (ETSI). Network Functions Virtualisation. http://portal.etsi.org/NFV/NFV_White_Paper.pdf, Oct. 2012. [Online; accessed 2023-07-17].

[8] Fitzpatrick, B. Memcached. https://memcached.org/. [Online; accessed 2023-07-17].

[9] Fried, J., Ruan, Z., Ousterhout, A., and Belay, A. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 281–297.

[10] Glozer, W. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. [Online; accessed 2023-07-17].

[11] Hallyn, S. E., and Morgan, A. G. Linux capabilities: making them work.

[12] Han, S., Jang, K., Park, K., and Moon, S. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010), 195–206.

[13] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2018), CoNEXT '18, Association for Computing Machinery, p. 54–66.

[14] Hwang, J., Ramakrishnan, K. K., and Wood, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 445–458.

[15] Ibanez, S., Mallery, A., Arslan, S., Jepsen, T., Shahbaz, M., Kim, C., and McKeown, N. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), USENIX Association, pp. 239–256.

[16] Iyengar, J., and Thomson, M. RFC 9000 - QUIC: A UDP-Based Multiplexed and Secure Transport. Internet RFC, Internet Engineering Task Force (IETF), May 2021.

[17] Jeong, E., Wood, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 489–502.

[18] Kalia, A., Kaminsky, M., and Andersen, D. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 1–16.

[19] Khan, I. irqbalance: design and internals. https://blogs.oracle.com/linux/post/irqbalance-design-and-internals, 2023. [Online; accessed 2023-07-17].

[20] Larabel, M. Disabling Spectre V2 Mitigations Is What Can Impair AMD Ryzen 7000 Series Performance. https://www.phoronix.com/review/amd-zen4-spectrev2. [Online; accessed 2023-07-17].

[21] Lemon, J. Kqueue - A Generic and Scalable Event Notification Facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (USA, 2001), USENIX Association, pp. 141–153.
[22] Leverich, J. Mutilate. https://github.com/leverich/mutilate. [Online; accessed 2023-07-17].
[23] Li, H., Wu, C., Sun, G., Zhang, P., Shan, D., Pan, T., and Hu, C. Programming Network Stack for Middleboxes with Rubik. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 551–570.
[24] Linux Foundation. Data Plane Development Kit (DPDK). http://www.dpdk.org. [Online; accessed 2023-07-17].
[25] Linux Kernel Library Project. Linux Kernel Library. https://github.com/lkl/linux. [Online; accessed 2023-07-17].
[26] Love, R. *Linux system programming: talking directly to the kernel and C library.* " O'Reilly Media, Inc.", 2013, pp. 97–98.
[27] Majkowski, M. Kernel bypass. https://blog.cloudflare.com/kernel-bypass/, 2015. [Online; accessed 2023-07-17].
[28] Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 399–413.
[29] Nginx, Inc. Nginx. https://www.nginx.com/. [Online; accessed 2023-07-17].
[30] Ntop. PF_RING. https://github.com/ntop/PF_RING. [Online; accessed 2023-07-17].
[31] Oracle Corporation. Performance Tuning - Administering Oracle Coherence. https://docs.oracle.com/en/middleware/standalone/coherence/14.1.1.0/administer/performance-tuning.html. [Online; accessed 2023-07-17].
[32] Ousterhout, A., Fried, J., Behrens, J., Belay, A., and Balakrishnan, H. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 361–378.
[33] Phothilimthana, P. M., Liu, M., Kaufmann, A., Peter, S., Bodik, R., and Anderson, T. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 663–679.
[34] Red Hat, Inc. Minimizing system latency by isolating interrupts and user processes. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/8/html/optimizing_rhel_8_for_real_time_for_low_latency_operation/assembly_binding-interrupts-and-processes_optimizing-rhel8-for-real-time-for-low-latency-operation. [Online; accessed 2023-07-17].
[35] Redis Ltd. Redis. https://redis.io/. [Online; accessed 2023-07-17].
[36] Richards, M. Linux kernel vs DPDK: HTTP performance showdown. https://talawah.io/blog/linux-kernel-vs-dpdk-http-performance-showdown/, 2022. [Online; accessed 2023-07-17].
[37] Salim, J. H., Olsson, R., and Kuznetsov, A. Beyond Softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5* (USA, 2001), ALS '01, USENIX Association, p. 18.
[38] ScyllaDB, Inc. Scylladb. https://www.scylladb.com/. [Online; accessed 2023-07-17].
[39] ScyllaDB, Inc. Seastar. https://github.com/scylladb/seastar. [Online; accessed 2023-07-17].
[40] Swedish Institute of Computer Science. lwIP. https://savannah.nongnu.org/projects/lwip/. [Online; accessed 2023-07-17].
[41] Tahhan, M., and Hunter, D. The hybrid networking stack. https://next.redhat.com/2022/12/07/the-hybrid-networking-stack/, 2022. [Online; accessed 2023-07-17].
[42] The Linux Foundation. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. [Online; accessed 2023-07-17].
[43] The Linux Foundation. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt. [Online; accessed 2023-07-17].
[44] THL A29 Limited. F-Stack. https://github.com/F-Stack/f-stack. [Online; accessed 2023-07-17.
[45] Toonk, A. Kernel bypass networking with FD.io and VPP. https://blog.apnic.net/2020/04/17/kernel-bypass-networking-with-fd-io-and-vpp/, 2020. [Online; accessed 2023-07-17].
[46] Ven, A. V. D., and Horman, N. irqbalance. http://irqbalance.github.io/irqbalance. [Online; accessed 2023-07-17].
[47] Xilinx, Inc. OpenOnload. https://github.com/Xilinx-CNS/onload. [Online; accessed 2023-07-17].
[48] Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O. S. N., Martinez, A., Liu, J., Simpson, A. K., Jayakar, S., et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 195–211.

A CACHE CAPACITY

While experimenting with Memcached, we have observed a phenomenon where an increased number of connections, even at moderate values, results in a noticeable throughput decrease. We have tracked this observation to *Last-Level Cache* (LLC) misses and show in Figure 9 a set of closed-loop experiments with an increasing number of connections per client (cf. Section 5.1). The figure

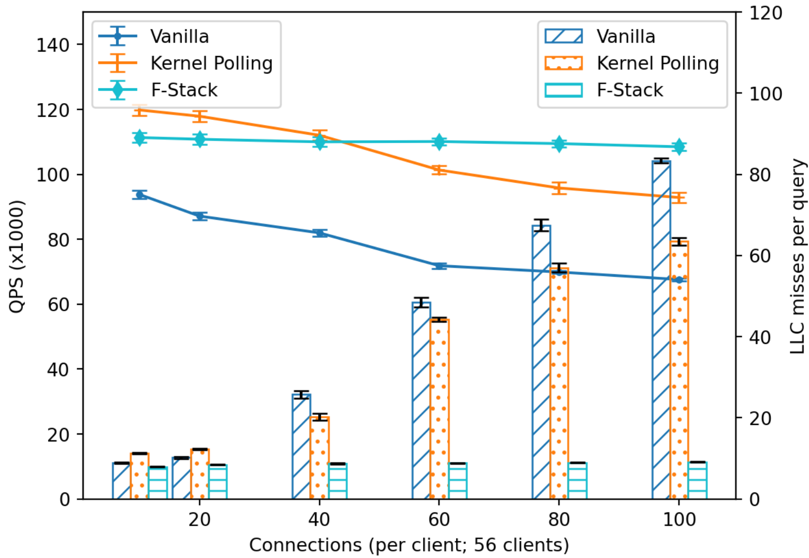

Fig. 9. Memcached: Throughput & LLC misses, 1 core

shows both the throughput performance and the number of LLC misses per query, i.e., normalized by throughput, and include F-Stack for reference. The results for the kernel network stack, for both vanilla or polling, demonstrate an inverse correlation between throughput and the number of connections, while F-Stack appears to be unaffected. At 10 connections per client (560 total), kernel polling achieves a throughput about 8% higher than that of F-Stack, but it eventually falls below as the number of connections increases. The kernel suffers from an increasing number of LLC misses per request, while for F-Stack, this number remains constant regardless of the number of connections.

The only meaningful explanation for this observation is that the effective cache footprint of the Linux network stack exceeds the LLC capacity of our particular server when handling a certain number of TCP connections. F-Stack appears to have a smaller memory footprint per connection, which removes LLC as a limiting factor on this specific hardware platform for these experiments. We remark that this is not a fundamental performance difference between kernel and user-level processing. It might be a difference between Linux and FreeBSD networking, or can be considered a consequence of network stack customization. Then again, we have also observed fairness issues at higher connection counts in F-Stack, so a further comprehensive investigation would be necessary to fully understand the issue at hand and possible mitigations. Finally, as trends in hardware show an increasing amount of LLC, with some recent processors [1] approaching gigabyte-sized LLCs, the real-world relevance of this observation might be limited.

Received August 2023; revised October 2023; accepted October 2023