# HyperPlane: A Scalable Low-Latency Notification Accelerator for Software Data Planes

Amirhossein Mirhosseini*
*University of Michigan*
miramir@umich.edu

Hossein Golestani*
*University of Michigan*
hosseing@umich.edu

Thomas F. Wenisch
*University of Michigan*
twenisch@umich.edu

***Abstract*—I/O software stacks have evolved rapidly due to the growing speed of I/O devices—including network adapters, storage devices, and accelerators—and the emergence of microservice-based programming models. Datacenters rely on fast, efficient Software Data Planes (SDPs), which orchestrate data transfer between applications and I/O devices. Modern data planes are user-level software stacks, wherein cores spin-poll a large number of queues to avoid the attendant overheads of kernel-based I/O. Cores often poll empty queues before finding work in non-empty ones. Interrogating empty queues hurts peak throughput, tail latency, and energy efficiency as it often entails fruitless cache misses. In this work, we propose *HyperPlane*, an efficient accelerator for the notification mechanism of SDPs. The key features of HyperPlane are (1) avoiding iteration over empty I/O queues, unlike software-only designs, resulting in queue scalability, (2) halting execution when I/O queues are idle, leading to work proportionality and energy efficiency, and (3) efficiently sharing queues across cores to enjoy strong theoretical properties of scale-up queuing. HyperPlane is realized through a hardware subsystem associated with a familiar programming model. HyperPlane's microarchitecture consists of a *monitoring set* that watches for work arrival from I/O, and a *ready set*, which tracks ready queues and distributes work to cores based on various service policies and priority levels. We show that HyperPlane improves peak throughput by** 4.1× **and tail latency by** 16.4× **compared to a state-of-the-art SDP.**

***Index Terms*—software data plane, acceleration, spin-polling, notification**

## I. Introduction

Computer system designers are on the hunt to address "Killer Microseconds" [16], [26], [69]. The latency to access modern I/O devices—such as emerging storage-class and disaggregated memories [13], [14], [63], [77], 100+ gigabit networking devices [21], and high-throughput accelerators [24], [82]—is as low as single-digit microseconds. At such low latencies and high throughputs, the I/O software stack becomes a critical factor in end-to-end communication performance. Moreover, modern cloud applications are shifting away from ms-scale single-binary monoliths to loosely-coupled µs-scale microservices, to achieve better scalability, reliability, and programmablity [27], [42], [91]. With µs-scale service times, the I/O software stack's latency becomes comparable to computation time and must be aggressively optimized [90], [92].

In conventional systems, sharing an I/O device among multiple applications is orchestrated through the kernel—depicted in Figure 1(a). When a user process signals the kernel that it wishes to perform I/O via a system call, the protocol and transport processing software stack is carried out by kernel threads, either by directly borrowing the user process's CPU, or by using interrupt mechanisms and kernel scheduling to place work on another core. Nonetheless, the mechanisms involved in such kernel-based approaches—including synchronization, scheduling, inter-processor interrupts, switching address spaces, and copying data across address spaces—impose significant performance overheads when dealing with I/O devices that exhibit µs-scale latencies and gigabits- to terabits-per-second throughputs [50].

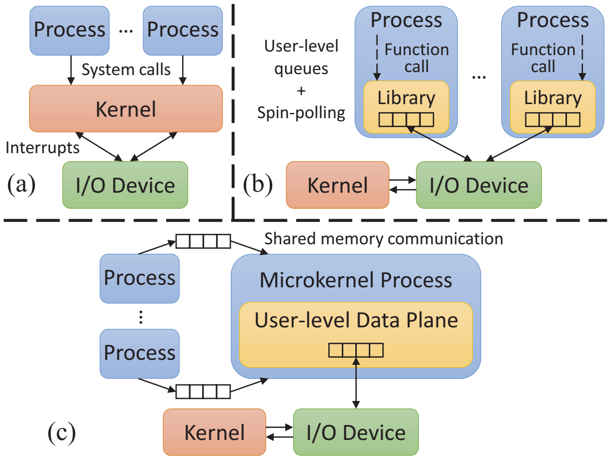


Fig. 1. I/O communication approaches: (a) conventional kernel-based, (b) user-level library OS, (c) microkernel-based "software data planes".

In contrast, as illustrated in Figure 1(b), modern I/O devices provide virtual user-level queue pairs for user processes to communicate directly with them, bypassing the kernel software stack. As such, since traditional interrupts cannot be delivered to user code without kernel assistance, user processes often perform spin-polling on the queues to be notified of new data/task arrivals. In these schemes, I/O software stacks are often implemented as a library operating system loaded as part of the user process, and invoked through function calls [19], [47], [50], [81]. Zero-copy data transfer mechanisms further improve CPU efficiency by placing I/O buffer pools in the user process address space, enabling incoming data items to directly land where the receiving user process can access them. The transport software in zero-copy I/O stacks arranges for data to

[*] Both authors contributed equally to this research.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00074

flow directly between buffer pools in user processes and I/O devices, without any intermediate copies.

Despite all the performance benefits of kernel-bypass I/O stacks, the key shortcoming of these systems is the lack of centralized coordination that conventional kernel-based approaches enable. By providing a central view of all resources and the tenant processes/VMs that communicate with each resource, the kernel is able to deliver more efficient scheduling, fairness, and resource accounting. As a result, a popular alternative approach used by systems like Google's Snap [67] is deploying the I/O stack within a user-level software data plane (SDP) microkernel module—illustrated in Figure 1(c)—communicating with the application processes through user-level shared memory queues. This way, the user-level SDP manages all the I/O queues and exploits its centralized view to provide better resource management and scheduling while retaining most of the performance benefits of kernel-bypass I/O stacks. Furthermore, by decoupling and isolating the I/O stack from both the application and the kernel, this approach provides better locality and also enhances development and release velocity for the I/O software stack. Academic projects like Shenango [79] have advocated similar approaches.

Nonetheless, even microkernel-based SDPs typically rely on spinning cores and user-level queues in their transport software. Although spin-polling is an easy-to-use, fast-reacting approach for communication and signaling, it involves a number of inevitable drawbacks, especially when dealing with hundreds (or more) queues. First, the traffic that passes through these queues is often unbalanced, and consequently, a large subset of queues are empty at any given point in time. Traffic is unbalanced because (1) some I/O devices are inherently more frequently accessed than others, and (2) tenant applications/VMs typically experience bursty activity patterns at different times. As such, a large fraction of time in SDPs is wasted interrogating empty queues, especially when reading empty queue heads incurs cache misses, which is quite likely. This useless work substantially hurts the tail latency and peak throughput of SDPs and limits their queue scalability [44]. Furthermore, SDPs may even exhibit "work disproportionality"; that is, they perform more fruitless spinning work in terms of Instructions Per Cycle (IPC) at lower transport load, leading to energy inefficiency and fast core aging. Finally, SDPs can benefit substantially from sharing queues across multiple cores to achieve better queuing properties. However, the coherence and synchronization costs of spinning on shared queues make such sharing impractical [30].

In this paper, we propose *HyperPlane*, a hardware notification accelerator that facilitates fast (user-level) SDPs, which unlike software-only spinning alternatives, exhibits queue scalability and work proportionality and enables efficient queue sharing. HyperPlane comprises a programming model front-end and a hardware microarchitecture back-end that together enable efficient operation. The core of its programming model is the `QWAIT` instruction, which has similar semantics to the *select-case* construct in the Go programming language [4]. `QWAIT` waits on a set of doorbell locations associated with queues and blocks execution until a work item arrives to a queue. Once one or more queues are ready, `QWAIT` returns the next Queue ID (*QID*) that must be serviced according to the specified service policy—round-robin, weighted round robin, or strict priority. `QWAIT` is inspired by the x86 `MWAIT` and ARM `WFE` instructions, which halt execution until the contents of a single memory address or address range change.

The two key components of HyperPlane's microarchitecture back-end are a *monitoring set* and a *ready set*. The monitoring set comprises the locations of doorbells associated with each queue. These locations are monitored in hardware for cache coherence write transactions that indicate new work item arrivals. Once a write transaction for a doorbell is matched to a QID in the monitoring set, the QID is moved to the ready set. The ready set, which is HyperPlane's key departure from prior monitoring schemes, tracks ready QIDs and determines the next queue to service according to a service policy. `QWAIT` returns the QID of the next ready queue from the ready set. The ready set effectively functions as a task scheduler at non-trivial loads, sorting the order of ready queues to be serviced.

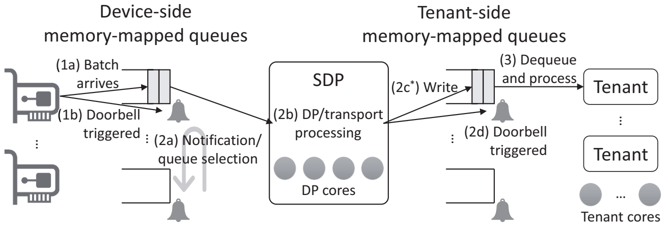
To the best of our knowledge, HyperPlane is the first hardware accelerator proposal that enhances the performance and energy efficiency of SDPs. HyperPlane achieves queue scalability as `QWAIT` always returns the next QID to be processed without the need for checking many empty queues. It avoids work disproportionality of fruitless spinning because it halts execution when there is no work item in any queue, avoiding wasted energy/execution resources or harming the execution of another hyperthread on the same core. Finally, since the monitoring and the ready set units are shared across all cores, HyperPlane enables efficient cross-core queue sharing and enjoys the strong properties of scale-up queuing models, providing higher performance and better support for queue priorities. Our results show that HyperPlane improves peak throughput by $4.1\times$ and tail latency by $16.4\times$, on average, in comparison to a state-of-the-art spin-polling–based SDP, across a varying number of I/O queues (up to 1000).

We first provide background on SDPs and motivate our work by illustrating challenges of SDPs (Section II). Next, we describe the design and detailed microarchitecture of HyperPlane (Sections III and IV). We then evaluate HyperPlane (Section V). Finally, we discuss related work (Section VI) and conclude (Section VII).

## II. Background and Motivation

### A. Software Data Planes

SDPs manage data communication of tenants (i.e., host applications or client applications/VMs) with I/O devices, such as Network Interface Controllers (NICs) [67], Solid State Drives (SSDs) [59], persistent memory devices [7], [9], and accelerators [11]. SDPs have two key functionalities: First, they manage I/O queues and direct traffic to the corresponding tenant and vice versa. Each tenant uses one or more queue pairs to communicate with particular I/O devices. Thus, SDPs must efficiently handle a large number of queues (i.e., O(1k)) corresponding to many tenants and several I/O devices.



* Not needed in case of in-place Data Plane (DP) / transport processing

Fig. 2. Software Data Plane (SDP) operations.

Second, SDPs provide low-level I/O operation services to tenants. Software defined networks and virtual network functions demand fast packet processing, which is enabled by SDPs through services like address translation, firewall, software switching, and deep packet inspection [33]. SDPs also provide services to virtualize storage systems [98], administer shared I/O bandwidth [56], [95], enable Remote Direct Memory Access (RDMA) [12], [56], facilitate high-performance computing [1], perform erasure coding [6], and encrypt/decrypt data [3], [6].

Figure 2 illustrates the receive-side interactions in an SDP (the transmit-side diagram looks similar): (1a) a batch of one or more packets/work items arrives to one of the device-side memory-mapped queues; (1b) the device triggers the corresponding doorbell; (2a) the SDP module is informed of the arrival (through the doorbell); (2b) depending on the format/semantics of the work items in the queue, the SDP either performs transport processing in-place or (2c) writes/copies the transformed packets/work items to the corresponding tenant-side queue; (2d) the SDP triggers the tenant-side doorbell; and finally, (3) the tenant is informed of the packet/work item arrivals to process. Each tenant has a single or a few queues per (virtual) core. Therefore, it can easily monitor the queue via spin-polling or different variants of the `MWAIT` instruction. However, the SDP has to monitor all queues simultaneously and service them based on the predefined system policy, hence it cannot use `MWAIT` variants.

### B. SDP Challenges and Goals

Even though SDPs rely on spin-polling to deliver high throughput and low latency notification, they suffer from several inherent inefficiencies. First, SDPs lack queue scalability [44]. Spinning cores iterate over all of the input queues at full tilt even when there is no work item in any of them. Increasing the number of queues puts excessive pressure on processor caches, which can hurt peak throughput and tail latency. This effect is exacerbated when traffic lacks balance, i.e., when a subset of queues contain no work items most of the time. Empty queues cost a spinning core time as it searches for the next ready work item in a non-empty queue. This cost is particularly high when interrogating empty queues may incur cache misses, slowing the polling loop. Since the time required to process a work item is usually short (i.e, a few microseconds), missing on multiple empty queue heads might take even longer than processing a ready queue.

Second, SDPs are not necessarily work-proportional. Modern cores can spin with high IPC. Therefore, spin-polling may require a core to perform more work when there are, in fact, fewer work items in the queues. Work disproportionality translates to energy disproportionality and faster processor aging [17], [44], [78]. It also has an adverse effect on workloads co-running on Simultaneously Multi-Threaded (SMT) cores. Useless spinning consumes execution resources and L1 cache bandwidth that could otherwise be effectively used by a co-runner hyperthread. Whereas spin-locks also exhibit the same drawback, the collateral damage of a spin-lock is lower because they spin only on a single memory location. Modern cores can easily detect such spin-loops and slow the spinning process [62]. Moreover, as shown by prior work, variants of the `MWAIT` instruction may be used to put the core in halt/sleep state until a write is performed to the lock location to prevent useless spinning and save energy [36].

Finally, scale-up queuing is impractical in SDPs [44], [79]. The scale-up queuing organization, wherein multiple cores fetch work items from a shared set of queues, has strong theoretical advantages compared to scale-out queuing, wherein each core is associated with a different set of queues [70]. First, scale-out designs may suffer from load imbalance as the traffic is usually unbalanced and only a subset of queues have work items—these queues are often non-uniformly distributed among cores [30]. In contrast, scale-up organizations provide an inherent load balancing property as all queues are visible to all cores in a work conserving setting. Second, scale-out organizations are prone to Head-of-Line (HoL) blocking [71], [72]—if the work item at the head of a queue takes longer than average to process, all work items behind it experience long queuing delays, yielding a high tail latency. Scale-up designs, however, are not susceptible to HoL blocking as all queues are visible to all cores—if an item takes long to process, the items queued behind it are drained by other cores. Finally, scale-up organizations provide better support for queue priorities. With scale-out organizations, each core can only prioritize over its own subset of queues. Despite these theoretical merits, SDPs that leverage scale-up queuing suffer from excessive synchronization and coherence performance overheads in practice, as the cores must frequently synchronize to dequeue items and the corresponding cache lines ping-pong among the cores' L1 caches.

Our goal is to design a hardware accelerator subsystem that can efficiently address these shortcomings and provide a scalable, low-latency, and work-proportional notification mechanism to enable high-performance SDPs. We will quantify these inefficiencies and how our proposed design, HyperPlane, can address them in Section V. In the next subsection, we present a case study of DPDK's queue scalability.

### C. Case Study: DPDK Queue Scalability

The Data Plane Development Kit (DPDK) [3] is a representative software infrastructure for building spin-polling–based user-level data planes. DPDK provides highly optimized poll mode drivers for numerous modern I/O devices—such as NICs

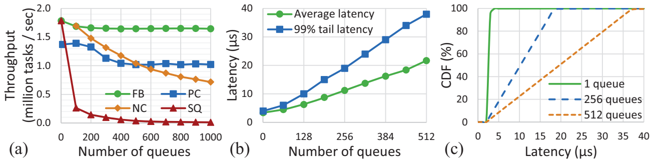


Fig. 3. DPDK: (a) Throughput of packet encapsulation, (b) Round-trip latency of packet forwarding under light traffic (~0.01 MPPS), (c) Distribution of round-trip latency.

and crypto devices—which enable cores to spin on user-level queues to communicate with the devices. Using DPDK, we illustrate the inherent queue scalability challenge of SDPs on a real server with a 24-core Xeon Skylake processor and a 100GbE Mellanox ConnectX-5 NIC.

We first consider the effect of increasing the number of queues on the maximum achievable throughput of a core performing network packet encapsulation tasks with various traffic shapes: *Fully Balanced* (**FB**), where traffic passes through all the queues; *Proportionally Concentrated* (**PC**), where traffic passes through 20% of the queues all the time and through the rest with a probability of 5%; *Non-proportionally Concentrated* (**NC**), where traffic passes through 100 queues all the time and through the rest with a probability of 5%; *Single Queue* (**SQ**), where traffic passes through only one queue. Figure 3(a) shows task execution throughput at different numbers of queues for the mentioned traffic shapes. We observe a drastic drop in throughput with *SQ* traffic. This drop is caused by useless spinning on empty queues, which is exacerbated by cache misses incurred for fetching queue heads. The throughput drop with the *NC* traffic is milder since the ratio of non-empty to empty queues grows at a smaller rate by increasing the number of queues, compared to *SQ*. With *FB* and *PC*, the ratio of non-empty queues to empty queues is constant (i.e., zero and four, respectively). Therefore, the throughput stabilizes as the number of cache misses per task becomes constant. In summary, the throughput of SDPs is adversely affected when traffic is concentrated in a small number of queues, and the rest are usually empty, which is the common case.

Next, we show how latency is affected by increasing the number of queues. Figure 3(b) reports the round-trip latency of a core forwarding packets received from the machine's NIC at different numbers of queues. Latency is measured at a packet generator, which sends/receives packets to/from the machine under test. To avoid queuing delays, we offer minimal load in this test (~0.01 Million Packets Per Second (MPPS)). Therefore, the reported latency is composed of service time (packet forwarding by the core) plus round-trip time. As shown in the Figure, both average and 99th percentile tail latencies grow almost linearly with the queue count because of more cache misses due to reading empty queue heads. Furthermore, tail latency grows with a higher slope—in the tail case, the data plane has to poll over far more empty queues before finding work in a ready queue, compared to the average case. This finding is further illustrated in Figure 3(c), which shows the Cumulative Distribution Function (CDF) of latency at three different queue counts. With more queues, the latency distribution spans a wider range, resulting in a larger difference between average and tail latencies.

## III. HyperPlane Design

In this section, we explore the design of the HyperPlane notification system. HyperPlane seeks to enable efficient SDPs that, unlike spinning-based variants, (1) do not need to iterate over empty queues to find work in ready ones, (2) block/halt when all queues are empty rather than spinning fruitlessly, and (3) allow multiple cores to efficiently monitor a shared set of queues to provide higher performance with strong support for queue priorities and different service policies.

HyperPlane seeks to facilitate the notification/queue selection operation of SDPs (step (2a) in Figure 2) in both directions (transmit and receive). It comprises a programming model and a hardware notification subsystem. At a high level, the programming model centers around the `QWAIT` instruction, which waits on a set of queue head doorbell locations and returns the QID for the next ready queue. The hardware subsystem relies on two key components, the *monitoring set* and the *ready set*. The monitoring set tracks the doorbell locations associated with each queue and observes cache coherence write transactions to these locations, which indicate that a work item has been enqueued. The ready set tracks, orders, and prioritizes queues that are ready to be processed.

### A. Programming Model

The key component of the HyperPlane programming model is the `QWAIT` instruction. `QWAIT` is inspired by x86 `MWAIT` and ARM `WFE` instructions, which monitor a single memory address or a contiguous address range. `MWAIT` halts the execution of a hardware thread and waits until the contents of a specified address range change. Whereas `MWAIT` is a privileged instruction that cannot be used in user applications, Intel has recently introduced a user-mode variant of this instruction, called `UMWAIT`, which can also run in unprivileged code [2]. Nonetheless, the `MWAIT` variants can at best only partially address the work disproportionality of spin-based data planes by blocking execution when all queues are empty and waiting for a work item to arrive in some queue. However, they cannot indicate in which queue the work item is located, requiring the code to iterate across many (likely empty) queues, hurting latency and throughput.

In contrast, `QWAIT` monitors a set of queue doorbell locations and returns the QID of the next ready queue—similar semantics to the *select-case* construct in the Go programming language [4]. Each queue is associated with a doorbell in memory, which is usually a word composed of multiple fields that specify various properties of the I/O queue. We assume a doorbell implementation wherein a field represents an atomic counter, indicating the number of elements in the queue, with similar semantics to a semaphore [87]—producers atomically increment the counter after enqueuing each element and consumers decrement the counter before dequeuing each element.

**Algorithm 1:** HyperPlane Programming Model

```
QWAIT_init(doorbell_addr_range, service_policy)      // Control Plane

for all QIDs do
    do
        doorbell = allocate_address(doorbell_addr_range)
    while (QWAIT-ADD(QID, doorbell) == FAIL)
    doorbell_map[QID] = doorbell
end

while true do                                        // Data Plane
    QID = QWAIT()
    doorbell = doorbell_map[QID]
    if QWAIT-VERIFY(doorbell) == False then
        continue
    end

    work_item = dequeue(QID)
    QWAIT-RECONSIDER(QID, doorbell)
    process(work_item)
end
---------------------------------------------------------------------
QWAIT-VERIFY(doorbell):                              // Atomic Instruction
    if is_empty(doorbell) then
        arm_in_monitoring_set(doorbell)
    end

QWAIT-RECONSIDER(QID, doorbell):                     // Atomic Instruction
    if is_empty(doorbell) then
        arm_in_monitoring_set(doorbell)
    else
        activate_in_ready_set(QID)
    end
```

A write from the producer to the doorbell location indicates that an item has been enqueued. These writes typically either trigger interrupts (e.g., PCIe MSI-X mechanism) or are polled by the SDPs. By watching all doorbell locations, HyperPlane is able to determine the next ready queue without iterating across them and without the overheads of an interrupt. Algorithm 1 presents the high-level programming model of the HyperPlane architecture, centered around the QWAIT instruction. Each HyperPlane thread runs the code presented in Algorithm 1 and is pinned to a physical core to prevent it from being context-switched.

**Control plane primitives.** These primitives are required to setup and configure the HyperPlane hardware and modify the list of queue doorbells. They are privileged instructions as they need access to physical or kernel memory. Therefore, they are only used in the kernel driver code. QWAIT_init is used to initiate the HyperPlane hardware and specify the address range from which doorbells can be allocated, as well as the service policy—round-robin, weighted round-robin, or strict priority. We will discuss service policies and their implementations in more detail in Section IV-B. QWAIT-ADD associates a doorbell address with a QID, adds it to the HyperPlane's monitoring set, and *arms* the address to be watched for work arrivals. It is used when a new tenant connects to the data plane. Conversely, when a tenant process terminates, either the tenant itself or the kernel driver must disconnect it from the data plane by removing its QIDs and releasing their space from the monitoring set via QWAIT-REMOVE.

**Data plane primitives.** In the body of the data plane thread, the QWAIT instruction is executed in a loop. Similar to MWAIT, the QWAIT instruction halts a hardware thread's execution if all queues are empty and waits for a work item to arrive in some queue. By halting, QWAIT prevents useless spinning and the consequent work disproportionality. A core may also enter a power-optimized mode to save more energy if all hardware threads are halted. When work arrives, QWAIT returns the QID of the ready queue. If multiple queues already have ready work items when QWAIT is executed, it returns the QID of the queue that should be serviced first according to the selected service policy, specified via QWAIT_init. The returned QID can then be used to service the corresponding queue. Hence, using the QWAIT instruction, HyperPlane does not waste time interrogating empty queues to find work, and immediately moves on to the next ready queue to be serviced.

The two yellow highlighted parts of the code in Algorithm 1 are required for the correctness of the hardware implementation and do not impact the high-level semantics of the code. Prior to servicing the returned QID, a QWAIT-VERIFY instruction is called to check whether the returned QID is in fact ready. QWAIT-VERIFY atomically performs two functions: (1) it indicates whether the queue is empty (i.e., by checking the value of the doorbell's atomic counter), and if it is, (2) re-arms it in the monitoring set to detect the arrival of subsequent work items. This instruction is needed to detect potential spurious wake-ups or QID returns—i.e., a returned QID might not necessarily correspond to a ready queue with available work items (e.g., due to false sharing). After the work item has been dequeued, the QWAIT-RECONSIDER instruction is called, which either re-arms a QID in the monitoring set or re-activates it in the ready set (we will discuss these structures later) based on whether additional work items are queued, and is atomic with respect to new work item arrivals. QWAIT-VERIFY and QWAIT-RECONSIDER are both atomic instructions with memory barrier semantics, to prevent the execution from advancing before their operation is complete. We will explain these instructions in more detail in the next subsection.

Whereas QWAIT provides work proportionality by halting execution and avoiding fruitless spinning when all queues are empty, it might be desirable to execute a latency-insensitive task on the core when it is waiting for work items to arrive. This can be achieved in two different ways: (1) QWAIT can provide a non-blocking variant, which returns a reserved QID immediately even if there is no ready QID in the ready set. This way, the code performing a background task might poll the entire ready set with a single QWAIT instruction to see if any work item has arrived. (2) A background task may run on the second hyperthread of the core, which can efficiently use the core resources while the QWAIT thread is halted. To ensure the background task does not hurt data plane performance, the core may prioritize its SMT threads—using mechanisms proposed by prior work [34]—and only execute instructions from the low-priority background thread when the high-priority foreground thread is halted. QWAIT may be used as the signaling mechanism to detect when the data plane thread is halted, waiting for work to arrive.

Finally, we also envision two additional primitives, QWAIT-ENABLE and QWAIT-DISABLE, which may be used

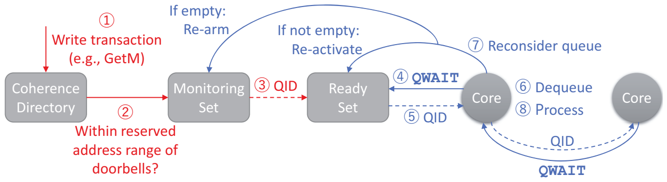


Fig. 4. High-level hardware block diagram of HyperPlane.

by the service procedure of a queue to temporarily inhibit a queue being serviced despite having ready work items. If a queue is disabled via QWAIT-DISABLE, its QID will not be returned until its service procedure is re-enabled via QWAIT-ENABLE (e.g., by timer). An example use case of these primitives is to limit the processing rate of a queue for a period for, e.g., congestion control in networking applications [67].

## B. Hardware Components

Figure 4 depicts the HyperPlane hardware block diagram. HyperPlane's operation is orchestrated by two hardware components: the *monitoring set* and *ready set*. At a high level, the monitoring set snoops the write transactions to a reserved address range dedicated to doorbells (steps 1 and 2). If a write transaction matches a QID in the monitoring set, it disarms the entry and activates the QID in the ready set (step 3). At this point, when a data plane core executes the QWAIT instruction (or has been blocked on it) (step 4), it will be able to return the corresponding QID, according to the service policy. When a QID is returned (step 5), the data plane core dequeues/locates a single or a batch of work items (step 6), and signals HyperPlane to reconsider the queue by either re-arming it in the monitoring set, if the queue is empty, or re-activating it in the ready set, if it is non-empty (step 7). Finally, the data plane core performs transport processing for the work item (step 8) and signals the tenant, prior to re-executing the QWAIT instruction (step 4). In the rest of this section, we explain the detailed functionality and interactions of the monitoring and the ready sets. We will later explore the detailed microarchitectural implementation of these components.

The monitoring set observes the doorbell memory addresses and detects work arrival by snooping the cache coherence write transactions to a specific pinned address range, reserved by the kernel driver for I/O doorbells. Any coherence transaction that grants exclusive ownership of a cache line to the requester will cause the monitoring set to indicate a wake-up/arrival on the corresponding queue (e.g., GetM transactions in the generic coherence protocols described in [89]). The monitoring set is independent of the coherence organization and is able to snoop messages either at a bus or directory. At a high level, the internal structure of the monitoring set is similar to a large associative memory that maps cache line tags to QIDs.

A monitoring set entry is composed of the following fields: tag, QID, monitoring bit, valid bit. The monitoring bit indicates that a cache line is "armed", being watched for write transactions. The monitoring set snoops all incoming write transactions, and if their tag matches an entry, it disarms the entry (i.e., sets the monitoring bit to 0, to indicate the line is no longer being watched), and activates the associated QID in the ready set. The QWAIT-ADD instruction is used to add a new entry to the monitoring set, e.g., when a new tenant connects to the data plane. The entries may later be removed via the QWAIT-REMOVE instruction. QWAIT-VERIFY and QWAIT-RECONSIDER instructions are used to re-arm an entry in the monitoring set. When an entry is re-armed, a coherence read transaction (i.e., GetS) is issued to ensure the line has no owner and the writes cannot be performed locally.

Although the QWAIT-VERIFY instruction filters out spurious writes, it is desirable that only doorbell writes performed by a producer (not the data plane thread itself) signal a QID in the monitoring set. Due to the memory barrier semantics of the QWAIT-RECONSIDER instruction, it is not issued before the dequeue operation (line 17 in Algorithm 1) is completed. Therefore, potential write transactions issued by the dequeue operation (i.e., decrementing the doorbell counter) do not trigger any QID in the monitoring set, since the corresponding entry is not armed during the dequeue operation. Note that once an item arrives to an armed queue, its entry in the monitoring set is disarmed, and further arrivals have no effect in the monitoring set until the queue is armed again (via QWAIT-RECONSIDER). When a QID is returned by the QWAIT instruction, the dequeue operation can retrieve a batch of items provided it correspondingly decrements the doorbell counter. Furthermore, note that Algorithm 1 seeks to deliver maximum intra-queue concurrency in multicore data planes, to eliminate potential HoL blocking scenarios and improve tail latency (see Section II). However, in various flow-based stateful networking applications—such as TCP/IP processing—packets or work items have to be processed in order [15], and intra-queue concurrency is not allowed. In such cases, lines 18 and 19 should be swapped to ensure a queue may only be serviced again when its previous work item has been processed.

The ready set is responsible for returning the QID of the next ready queue upon QWAIT, according to the selected service policy. Conceptually, it is composed of a list of the QIDs with available work items and an iterator that searches over the list and finds the next ready QID according to the service policy. For example, in the case of round-robin policy, the iterator searches over an unsorted list of QIDs to find the first one after the last serviced QID in a circular order. The ready set and its iterator may in principle be implemented either in hardware or in software. In case of a software implementation, the iterator code would be embedded into the QWAIT function (QWAIT would no longer be a single atomic instruction). However, in this case, in addition to the complications of providing atomicity, with fully- or semi-balanced traffics, the iterator code may need to iterate over potentially O(1k) QIDs in the list, adding a significant runtime overhead to the data plane performance, which may even be longer than the time required for processing an individual work item after locating it.

The QWAIT-VERIFY instruction in Algorithm 1 ensures that the returned QID indeed corresponds to a queue with ready work items. In case the queue is empty, its QID is atomically re-armed in the monitoring set. This instruction filters spurious wake-ups/activations—due to exclusive reads, false sharing, or doorbell writes that do not correspond to work item arrivals—while ensuring there is no window of opportunity for actual work arrivals to be missed. The QWAIT-RECONSIDER instruction in Algorithm 1 arranges for a QID to be considered again for service in a future iteration if it has ready work items. It atomically checks whether the queue is empty or already has work items available (e.g., more work items have arrived while the QID was waiting in the ready set). If the queue is empty, its entry is re-armed in the monitoring set. If it already has work items, the QID is directly activated in the ready set, so the iterator will select it again for service according to the service policy. The entire QWAIT-RECONSIDER operation must be implemented atomically to prevent various possible data races, including a scenario wherein the queue tests empty but a work item arrives before the QID is re-armed in the monitoring set, leading to a missed write transaction and consequent missed wake-up/activation.

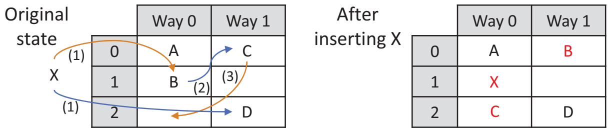
When running the data plane software on multiple cores to distribute the load, two options might be considered, as described in Section II: scale-up and scale-out. The scale-up organization is theoretically preferred as it achieves better throughput and latency, since it is not susceptible to load imbalance and HoL blocking. It also provides better support for queue priorities and weights, as it makes all queues visible to all cores. However, despite the theoretical advantages of scale-up queuing, scale-out organizations are often implemented in practice due to the synchronization overheads of scale-up organizations in spinning data planes. In contrast, HyperPlane enables efficient scale-up queuing organizations as the monitoring and ready sets are shared across the chip, accessible by all of the cores running the data plane software. As we will show in Section V, HyperPlane enjoys optimal latency and throughput characteristics, as it relies on scale-up queuing.

In case of NUMA systems, we envision a multi-socket HyperPlane deployment to employ separate data plane cores on each socket, with NUMA affinity between tenants and the data plane cores (e.g., doorbells mapped to the local memory channel address space) to avoid costly inter-socket communication. In this case, the deployment would exhibit scale-out queuing properties across sockets, which may result in suboptimal performance if there is heavy load imbalance across sockets. To mitigate this issue, a work-stealing approach may be used, wherein the data plane cores fetch ready QIDs from remote ready sets if the local ready set is empty. We defer exploration of such mechanisms to future work.

## IV. Detailed Microarchitecture

### A. Monitoring Set

Conceptually, the monitoring set provides a fully associative key-value lookup functionality, wherein the cache line tags for



| | Way 0 | Way 1 |
|---|---|---|
| 0 | A | C |
| 1 | B | |
| 2 | | D |

| | Way 0 | Way 1 |
|---|---|---|
| 0 | A | B |
| 1 | X | |
| 2 | C | D |

Fig. 5. An example 2-way Cuckoo hash table insertion.

the doorbell locations of all QIDs are watched simultaneously. However, large fully associative hardware structures are costly in terms of area and especially energy, due to performing many comparisons in parallel. Alternatively, serializing the comparisons significantly increases lookup latency [74]. Set-associative structures can suffer from high conflict rates, unless associativity is high (e.g., 64 or higher), which makes them almost as costly as fully associative structures. We propose to use a ZCache-like structure [85] to overcome these challenges.

The key idea behind ZCache operation is Cuckoo hashing [80], which decouples ways and associativity, enabling a highly associative structure with low lookup latency and energy. Figure 5 depicts a simple Cuckoo hash table with two ways. Each key is hashed using two hash functions, H0 and H1. When key X is looked up, it is hashed into row 1 in way 0 and row 2 in way 1, but both of those rows are occupied by other keys (i.e., a miss). To insert key X, way 0 is picked, and key B that is currently occupying the location where X must be inserted is rehashed using H1 and moved to way 1. By moving B to row 0 of way 1, C will be rehashed by H0 and moved to row 2 of way 0. The "table walk" process terminates at this point as C is placed in an empty location. If no empty location is found, despite the table containing empty entries, a *conflict* is said to have occurred.

As illustrated in the example, to look up a key, Cuckoo hashing need only check as many locations as the number of ways. However, by walking the table and performing a chain of replacements, the scheme provides a high effective associativity, and thus low conflict rate. Even though lookup latency and energy of this scheme are low, insertions can take substantially more time and energy than a conventional set-associative cache. However, insertions are only performed by QWAIT-ADD instructions, which are only executed when a new tenant is connected to the data plane (i.e., second or minute time scales). When an entry is disarmed and then re-armed in the monitoring set, it is not evicted and then re-inserted, but instead only its monitoring bit is set/reset. As a result, the costly hash table walks for insertion are only performed once for each QWAIT-ADD instruction. Re-arming a QID in the monitoring set via QWAIT-VERIFY or QWAIT-RECONSIDER instructions only involves a single tag lookup, similar to snooping the incoming transactions.

When a new tenant wishes to connect to the data plane, it executes the QWAIT-ADD instruction for every <*QID, doorbell address*> pair. Whereas Cuckoo hash tables exhibit much lower conflict rates than typical set-associative structures, conflicts are still possible. If QWAIT-ADD fails to insert a QID into the monitoring set due to a conflict, it returns an

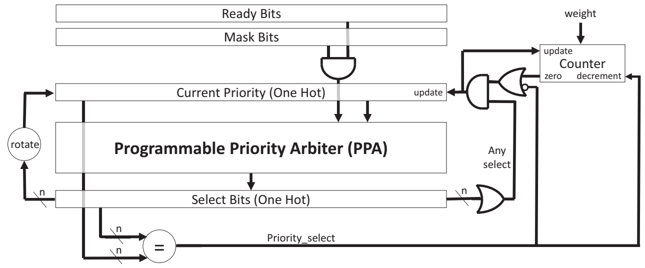


Fig. 6. High-level block diagram of the ready set hardware.

error code, and invokes driver code to reallocate a different doorbell address to the QID. Nonetheless, to minimize the conflict rate and ensure doorbell address reallocations are rare, the monitoring set (Cuckoo hash table) may be over-provisioned with respect to the maximum number of supported doorbells. Prior work has shown that over-provisioning the size of a Cuckoo hash table by 5%-10% reduces the conflict rate down to 0.1% [85], which is negligible. Note that after a new QID is added to the monitoring set, it stays there conflict-free, and is only removed by an explicit QWAIT-REMOVE.

Because the monitoring set snoops all coherence transactions to the doorbell memory address range, it is not subject to the conflict replacement behavior of a directory-based coherence scheme. In most directory-based schemes, when an entry is evicted from the directory, it sends invalidations to all the sharers of the line. However, the monitoring set is not an explicit sharer, but rather snoops all relevant coherence transactions (i.e., conceptually implemented as part of the directory). Therefore, it retains all the monitored doorbell tags, even in case of evictions in the directory. When an external write is about to be performed on the doorbell, the directory has be informed via a write transaction (i.e., GetM), which also informs the monitoring set. Since doorbells are allocated from a restricted address range (managed by the HyperPlane kernel driver), the monitoring set only need snoop addresses in this range and the snooping bandwidth is tractable. In the case of distributed directories, the monitoring set must also be banked, attached to individual directory banks. In such cases, the driver must spread doorbell addresses across banks.

### *B. Ready Set*

When the monitoring set matches a coherence transaction to a monitored doorbell, it disarms the entry and activates the QID in the ready set. The main responsibility of the ready set is to determine the next QID to be returned by QWAIT, according to the service policy. Whereas the monitoring set must be implemented in hardware (since coherence transactions are not visible to software), the ready set may in principle be implemented in software or hardware. In a software implementation, an iterator traverses a list of ready QIDs to find the next QID to be processed based on the service policy. However, in fully- or semi-balanced traffic scenarios where most queues are non-empty, the code must iterate over a large number of QIDs, imposing a substantial runtime overhead.

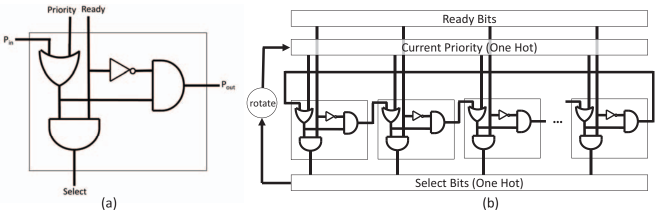


Fig. 7. (a) A bit-slice Programmable Priority Arbiter (PPA) cell, and (b) a multi-bit ripple-priority PPA design.

Instead, we propose a hardware implementation for the ready set, presented in Figure 6. Our hardware implementation takes as input a bit vector representing "ready bits" that correspond to different QIDs. That is, when a QID is returned by the monitoring set, the corresponding ready bit is set. As shown in the Figure, there is also a "mask bits" vector, which filters ready bits that should not be returned. These mask bits are manually set/reset via QWAIT-ENABLE and QWAIT-DISABLE to temporarily disable queues. The ready set hardware produces the "select bits" vector as its output, which is encoded in a "one-hot" fashion—that is, at most one of the bits can be set, indicating the selected QID to be returned by QWAIT. When the QWAIT instruction is executed, it is the ready set's responsibility to compute the "select bits", based on the the ready bits and the service policy.

The core of our ready set hardware implementation is a Programmable Priority Arbiter (PPA)—a widely used building block in on-chip networks and switching devices to grant access to one of the many requesters of a shared resource [31]. Besides "ready bits", the PPA module also takes a "current priority" one-hot bit vector as an input. The only bit position set to one in the current-priority bit vector indicates the QID with the highest priority. If that QID is ready, it is selected. Otherwise, its priority is propagated to the next bit position, wrapping around, until a ready QID is found.

To explain the operation of PPA, Figure 7 presents the ripple-priority bit-slice implementation of the PPA module, which is one of its simplest implementations. Its operation is similar to a ripple-carry adder. As shown in the Figure, at each bit position, the hardware checks (1) whether the ready bit is set to one and (2) whether priority is given to that bit position (via the one-hot $Priority$ input or from a previous bit position via $P_{in}$). If both conditions are met, the corresponding select bit is set. Otherwise, if $Priority$ or $P_{in}$ is asserted, but the ready bit is not set, priority is propagated to the next bit position via $P_{out}$. Ripple-priority implementation of PPA results in linear delay and hardware complexity. Furthermore, as shown in Figure 7, it requires a "wrap-around" connection that results in a combinational loop, making it difficult for EDA tools to synthesize and analyze the hardware.

In contrast, modern PPA implementations use thermometer coding [45] to eliminate the wrap-around connection and Parallel Prefix Network (PPN)-based designs to reduce the delay complexity of priority propagation to logarithmic [39]. PPNs are enhanced variants of look-ahead designs—such as carry look-ahead adders—and are used in almost all state-of-the-art high-speed adders [18]. PPNs provide a better hardware complexity vs. latency trade-off, compared to naïve carry look-ahead designs, making them scalable to thousands of bits [84]. In our implementation, we employ a Brent-Kung PPN [23], which is optimized for hardware complexity to be scalable to high bit counts—our RTL analysis shows the latency and hardware costs of the ready set to be small. We will provide a detailed analysis in the next subsection.

Our proposed ready set hardware design can efficiently implement the three most common service policies. With a *round-robin* policy, the selected QID in each round must exhibit the lowest priority in the next round. Thus, as shown in Figure 6, if the "Any Select" signal is set to one, indicating there was a QID selected at this round, the current-priority bit vector will be the rotated version of the select bit vector to give the highest priority to the bit position next to the one corresponding to the currently selected QID. The *weighted round-robin* policy is a generalization of round-robin, which allows each queue to be serviced for multiple consecutive rounds once it is selected. By giving different weights to different queues, weighted round-robin accommodates the differentiated arrival rates and QoS requirements of various tenants. In this case, when the current-priority bit vector is reloaded, the corresponding weight for the current-priority QID is loaded into a counter. Every time the queue is serviced, the counter is decremented. When the counter reaches zero or the current queue runs out of work items, the priority is passed to a different QID by reloading the current-priority and the weight register. Finally, by fixing the value of the current-priority bit vector to "10...0", the hardware implements a strict priority policy, wherein lower-numbered QIDs are always prioritized over higher-numbered ones. However, this policy is usually not used in real applications as it would result in the starvation of low-priority queues; instead, a weighted round-robin policy is often used, which differentiates queue priorities through weights and avoids starvation.

### C. Hardware Costs

We have considered a 1024-entry banked monitoring and unified ready set, shared across 16 cores. We modeled the hardware costs of the ready set via an RTL implementation in 32nm technology, and derived the area, power, and timing estimates for the core and the monitoring set via CACTI [74] and McPAT [61] models. Note that, during normal data plane operation, the monitoring set is similar to the tag array of a 2-way associative cache in terms of latency and energy since arming/disarming QIDs only involves 2-way lookups—the table walk process is performed only once for each `QWAIT-ADD` instruction. Synthesis of our RTL design reports the area of a 1024-entry ready set to be 0.13 $mm^2$. We estimate the area of the monitoring set to be 0.21 $mm^2$, while our baseline core occupies 8.4 $mm^2$ of area. Hence, the overall area overhead of the HyperPlane hardware components is within 0.26% of the total core area, for a 16-core chip. Similarly, we estimate the power costs of HyperPlane to be within 0.4% the total core power (within 6.2% of a single core; 2.1% for the ready set and 4.1% for the monitoring set). Note that our analysis considers 16 cores but does not include the uncore area/power. Thus, full-chip overheads are even smaller.

TABLE I
MICROARCHITECTURE DETAILS.

| | |
|---|---|
| Core | 8-wide issue OoO, 192/32-entry ROB/LSQ |
| L1 I/D | Private, 32 KB, 64B lines, 4-way SA |
| LLC | 1 MB per core, 64B lines, 16-way SA |
| CMP | 16 cores, directory-based MESI coherence |
| HyperPlane | 1024-entry monitoring and ready set |

From a timing perspective, our RTL model reports the latency of the ready set to be 12.25 ns. Since processing each work item takes a few microseconds, the ready set can easily serve `QWAIT` requests from O(100) cores—the number of cores needed for SDPs is usually small (1-4 [67], [79]). We have considered the lookup latency of the monitoring set to be within 5 CPU cycles but this latency is not on the critical path of the `QWAIT` instruction, unless it is halted, waiting for arrivals. To simplify the complexities of non-uniform access latencies of different cores to the single unified ready set, we have conservatively considered the `QWAIT` instruction latency to be 50 cycles in our experiments, which is higher than sum of all the latencies involved.

## V. EVALUATION

### A. Methodology

We use the gem5 simulator [20] and augment it to model the hardware components of HyperPlane. We model a 16-core x86-64 CMP system of which our data plane software runs on 1-4 cores [67], [79]. We model the core power consumption using McPAT [61]. Experimental microarchitecture details are described in Table I. We use an in-house SDP system based upon DPDK [3] that is able to be run in the simulator. Our SDP infrastructure closely tracks the performance characteristics of DPDK. Producer and consumer cores communicate through lock-free task queues. Emulated I/O sources running on "producer" cores generate traffic with different shapes and loads, which is passed through the data plane. Traffic shapes are the same as those used in Section II. We only report results for the round-robin service policy, as we found service policy to have minimal impact on the performance trends. We evaluate our SDP framework using the following tasks:

- **Packet encapsulation**: Network tunneling protocols leverage packet encapsulation to enable data movement of a network over another network. We use the GRE protocol [37] to encapsulate IPv4 packets within IPv6 packets.
- **Crypto forwarding**: Network traffic is often encrypted for secure communication. In this task, network packets are encrypted through AES-CBC-256 [41].
- **Packet steering**: Work distribution mechanisms aim to avoid datacenter network traffic congestion and scale performance [10], [15]. We employ a packet steerer that redirects the traffic by obtaining a session affinity from a hash table.
- **Erasure coding**: Erasure codes are commonly used in storage applications to detect and/or correct errors in stored

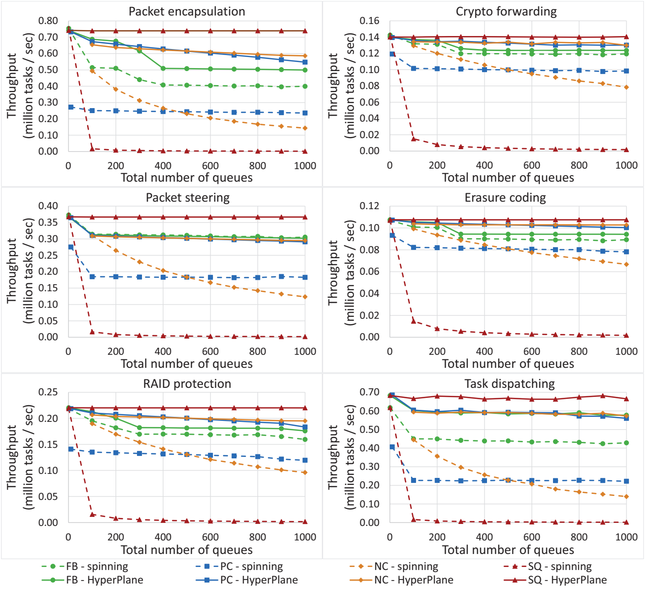


Fig. 8. Peak throughput of a spinning data plane and HyperPlane.

data [60]. We use Reed-Solomon erasure coding to encode data blocks/fragments using a Cauchy matrix.

- **RAID protection**: RAID is another mechanism for storage fault tolerance. In this task, RAID with P+Q redundancy is used to calculate parity bytes of input data blocks [25].
- **Request dispatching**: Online data-intensive applications dispatch microservices between servers at different tiers [92]. Our dispatcher task identifies request types and prepares the remote procedure calls to be dispatched.

### B. Queue Scalability

**Peak throughput.** We first characterize the peak achievable throughput for different numbers of queues to evaluate the queue scalability of HyperPlane and alternative spinning data planes. The peak throughput of the spinning data plane at different numbers of queues on a single core is reported in Figure 8 for the various workloads. Consistent with Section II, the throughput drop is the most drastic with the *SQ* traffic and is milder with the *NC* traffic since the core needs to spin-poll a larger number of empty queues before finding work in ready ones. With more queues, executed tasks use more data buffers in total, and as a result, we also observe a throughput decrease with the *PC* and *FB* traffics when the total size of task data and queue metadata exceeds the LLC size. However, since a task is executed for every $n$ queue head polls ($n \approx 5$ for *PC* and $n = 1$ for *FB*)—each incurring a queue head cache miss at larger queue counts—throughput converges to a constant value with these two traffic shapes.

Figure 8 also reports the peak throughput achieved by HyperPlane. HyperPlane avoids the useless work of interrogating empty queues and the corresponding cache misses. Thus, it recovers the lost throughput of the spinning data plane caused by the empty queues. In the *SQ* and *NC* traffics, where the number of active queues is constant (1 and 100, respectively), HyperPlane maintains its peak throughput when the total number of queues is increased. In the case of packet encapsulation, however, we observe a slight decrease in throughput for the *NC* traffic, due to an increase in the total data size of tasks and queues when the queue count increases. HyperPlane also exhibits a slightly larger throughput decrease for the *PC* and *FB* traffics, again due to the larger total data size of tasks and queues with more queues. Under the *FB* traffic, HyperPlane achieves better peak throughput in comparison to the spinning data plane in the case of packet encapsulation and task dispatching. Whereas the offered load fully saturates the processing capacity of the data plane core, empty queues are still occasionally observed, as our arrivals follow a Poisson process (memoryless inter-arrival times), which exhibits transient load variability. Thus, HyperPlane improves the peak throughput in the *FB* traffic particularly for shorter workloads, where the processing time of work items is more comparable to missing on empty queue heads in the spinning data plane. Overall with different traffic shapes and queue counts, HyperPlane improves the peak throughput by 4.1×, on average, compared to the spinning data plane.

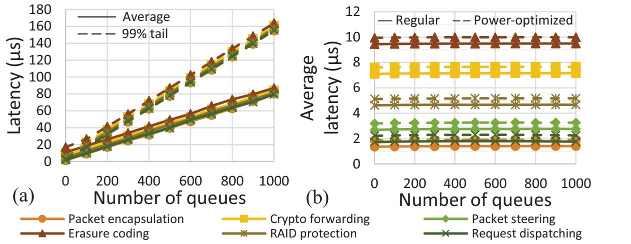


Fig. 9. Latency under light traffic (<1% load): (a) Average and tail latency of a spinning data plane, (b) Average latency of HyperPlane in regular and power-optimized modes.

**Zero-load latency.** Figure 9 reports the zero-load latency across workloads as the queue count is increased. Traffic is set to be very light (<1% load) to avoid queuing delays. With the spinning data plane (Figure 9(a)), both average and tail latencies grow linearly as the number of queues is increased, because the core has to check more empty queues (and possibly incur cache misses) before finding work in the ready queue. Consistent with Section II, the difference between tail and average latency grows with the queue count as the latency variation is higher with more queues—tail latency represents a worst case, wherein the iterator code has to traverse almost all queues before it reaches the ready one. Using HyperPlane, in contrast, the core avoids additional latency of checking empty queues. As a result, HyperPlane is perfectly queue-scalable, and neither average nor tail latency is affected with more queues, as depicted in Figure 9(b) (tail latency is not shown for HyperPlane as it does not differ significantly from the average latency at zero load). Whereas tail latency can be more than 100 µs for large queue counts in the spinning data plane, HyperPlane keeps both average and tail latencies below 10 µs even at 1000 queues. HyperPlane improves average/tail latency by 9.1× / 16.4×, on average, at different queue counts. Note that with one queue, the core in the spinning data plane quickly finds a task in the queue upon its arrival. But, due to the latency of the monitoring and ready sets (Section IV-C), HyperPlane underperforms the spinning data plane by at most 3% for a single queue. However, as the latency of the spinning data plane grows with queue count, HyperPlane outperforms the spinning data plane with as few as two queues.

HyperPlane may enter a power-optimized mode when it is idle and all queues are empty. Power saving in the idle state introduces an additional wake-up latency, which we will discuss in Section V-D. The spinning data plane may outperform HyperPlane because of such a wake-up latency for small numbers of queues. Figure 9(b) reports the average latency of HyperPlane at zero load with a wake-up latency of ~0.5 µs (transitioning from C1 to C0 state). Our experiments show that because of this additional latency, the spinning data plane reacts faster to task arrival in comparison to HyperPlane for up to 6 queues on average (nine queues in the worst case). With more than six queues, even the power-optimized HyperPlane outperforms the spinning data plane.

### C. Multicore Performance

In this section, we compare the performance of HyperPlane and spinning data planes under multicore scenarios. To provide a comprehensive analysis under the entire load spectrum, we report only results for the packet encapsulation workload. Other workloads follow the same performance trends. Figure 10 reports the 99th percentile tail latency under (a) *FB* and (b) *PC* traffics with four cores and 400 total queues. We report latency under the following configurations: scale-out, where each core is statically assigned 100 queues to serve; scale-up-2, where each 2-core cluster is assigned 200 queues; and scale-up-4, where all four cores share all 400 queues. Note that in HyperPlane, there is a single ready set shared among all serving cores. To fairly compare HyperPlane with the spinning data plane, we assume the ready set is partitioned in the scale-out and scale-up-2 configurations and only returns QIDs that belong to a core's subset of queues when the core executes `QWAIT`. In practice, any core can serve any ready queue in HyperPlane, as in the scale-up-4 configuration.

We make two key observations in Figure 10(a) for *FB* traffic: First, whereas a scale-out HyperPlane system does not considerably increase the saturation throughput compared to the spinning alternative, it significantly reduces the tail latency under pre-saturation loads (e.g., by 3.2× under 50% load). At lower loads, the expected number of empty queues the spinning data plane interrogates is higher, but this number reduces to zero at 100% load, and hence the performance of both designs converge. Second, whereas scale-up designs improve HyperPlane latency, especially at high loads, due to their queuing model advantages (see Sections II), spinning alternatives experience significant performance drops due to (1) synchronization and cache coherence (ping-ponging of queue heads) costs, and (2) the expected number of empty queues traversed increases with the total number of queues—that is, each core in the scale-up-4 design iterates over 400 queues (compared to 100 in scale-out) and is likely to interrogate 4× more queues every time it looks for work.

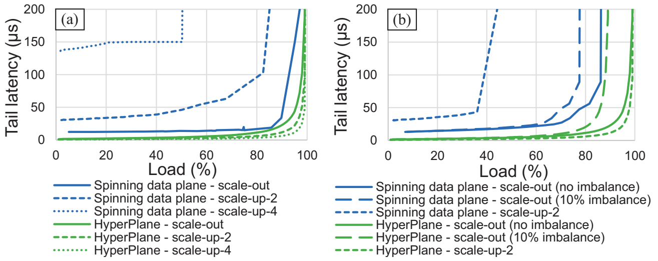


Fig. 10. Multicore 99% tail latency: (a) Fully balanced traffic, (b) Proportionally concentrated traffic.

For the *PC* traffic, Figure 10(b) compares the tail latency of the scale-out and scale-up-2 organizations as well as a variant of scale-out with 10% static load imbalance. As explained in Section II, the scale-out organization is susceptible to load imbalance. Whereas for *FB* traffic, load imbalance might occur only dynamically, depending on the instantaneous availability of work items in queues, non–fully-balanced traffics, such as *PC*, are also subject to static load imbalance, wherein active queues are not assigned to cores in a balanced manner. Even though the runtime system may detect such load imbalance scenarios and reassign queues to cores, it can only react to traffic and workload changes at coarse-grain time scales—in practice, load imbalance of at least 10% is inevitable.

We make two observations in Figure 10(b): First, unlike *FB* traffic, HyperPlane increases the saturation throughput by 23% compared to the spinning data plane, in addition to improving tail latency under pre-saturation loads by at least 73% for *PC* traffic. Moreover, load imbalance is inevitable in real scenarios. The scale-up HyperPlane is not subject to load imbalance and improves the saturation throughput by 11% and 37% compared to scale-out HyperPlane and scale-out spinning data plane both with 10% load imbalance, respectively. However, the scale-up spinning data plane exhibits 54% lower saturation throughput, even compared to its scale-out alternative with 10% load imbalance, due to synchronization overheads.

### D. Work Proportionality

HyperPlane is designed to avoid the useless spinning of SDPs and only execute when there is work in the system—that is, it halts execution when there is no work item in any queue. We quantify work proportionality of HyperPlane with respect to the data plane load. Figure 11(a) reports the Instructions Per Cycle (IPC) of a core running a packet encapsulation data plane. In HyperPlane, IPC—which is a measure of core activity—grows linearly with load. In contrast, when using a spinning data plane, the IPC is disproportionate to the amount of load and decreases as the load increases. The IPC of the spinning core is the highest at 0% load, meaning that the core spins full-tilt, desperately looking for work. Figure 11(a) divides the IPC based on performing useful work or useless spinning for the spinning data plane. At zero load, all the committed instructions are useless, and useful instructions gradually grow by increasing the load. Whereas the IPC of the spinning data plane generally decreases at higher loads, we observe an anomaly at loads above 50%. This anomaly

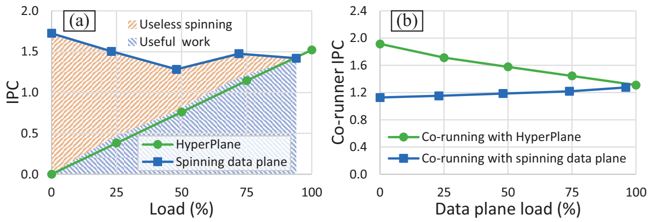


Fig. 11. (a) IPC breakdown of a software data plane, (b) IPC of an application co-running with the software data plane.

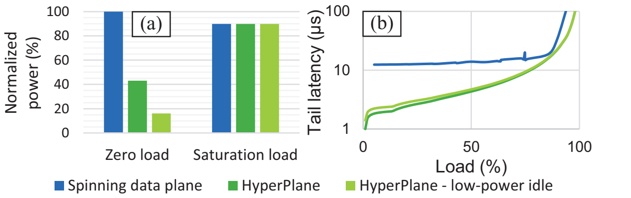


Fig. 12. (a) Power consumption of a spinning data plane and HyperPlane with/without power optimization, (b) The effect of wake-up latency of power-optimized HyperPlane.

arises because queue heads start to fall out of the L1 cache at higher loads, slowing the IPC of spinning.

The high IPC of useless spinning can harm system efficiency and restrict the performance of other applications. In particular, it has an implication on the applications co-located with the data plane through SMT. Scheduling execution resources among competing hyperthreads is typically performed based on thread activity or IPC (e.g., the ICOUNT policy [96]), which is counterproductive for idle poll loops. We quantify interference of the spinning data plane as well as HyperPlane with an SMT co-runner, which is a regular application performing matrix multiplication, on a core with two hardware threads. Figure 11(b) reports the IPC of the co-runner at different loads of the SDP. Interestingly, when the spinning data plane is used, the co-runner IPC increases with the data plane load—spinning is a more severe antagonist than performing actual work. With HyperPlane, however, the co-runner IPC decreases when data plane load increases. This again implies work proportionality of HyperPlane. HyperPlane does not interfere with a co-runner when there is no work.

Work disproportionality in spinning data planes also results in energy disproportionality of the core. We use McPAT [61] to model the core power consumption running the SDP. Figure 12(a) reports the normalized power consumption of the core at zero and saturation loads. Perhaps surprisingly, the spinning data plane consumes more power at zero load compared to saturation. This is consistent with the previous observation of the disproportional IPC at zero load due to full-tilt useless spinning (See Figure 11). HyperPlane, however, exhibits higher energy proportionality. Whereas HyperPlane already consumes much less power at zero load by halting the execution, it can also enjoy a power-optimized mode, wherein the core enters a deeper "C state" to save power. We only consider transitioning from C0 to C1 state, as long latencies of deeper C states may hurt data plane performance. As shown in Figure 12(a), by using HyperPlane in the power-optimized mode (i.e., core transitions to C1 when halted), core power consumption reduces down to only 16.2% at zero load.

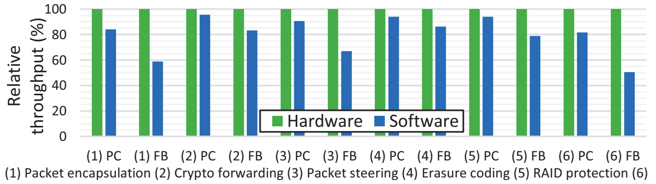


Fig. 13. Throughput of a software-based vs. hardware-based ready set with two different traffic shapes.

Using HyperPlane in the power-optimized mode may cost additional wake-up latency. We consider the wake-up latency of the power-optimized HyperPlane to be ~0.5 µs to be consistent with performance characterizations of `MWAIT` [36] and C1-to-C0 transitions [86]. We report the tail latency of the experimental scenario of Figure 10(a) for the power-optimized HyperPlane in Figure 12(b). Results are reported in log-scale, so the differences can be visible. As the Figure shows, at zero load, power-optimized HyperPlane yields 38% higher tail latency, compared to regular HyperPlane. However, its achieved tail latency is still 8.9× lower than the one achieved by the spinning data plane. As the load increases, HyperPlane enters the power-optimized mode less often, and hence the gap shrinks rapidly—only 8% higher latency at 50% load.

### E. *Ready Set Implementation*

As discussed in Section III-B, the ready set may be implemented either in hardware or software. We now consider performance implications of a software-based ready set. In the software implementation, `QWAIT` either waits or iterates over the list of ready QIDs in a piece of code and returns one of them based on the system policy. We measure the peak throughput of a single core in HyperPlane monitoring 1000 queues. Figure 13 reports the normalized throughput of the software-based implementation over the hardware-based implementation for different workloads with the *PC* and *FB* traffic shapes. For both shapes, throughput of the software-based implementation is considerably lower than its hardware-based counterpart. Throughput drop is more severe with the *FB* traffic (down to 50%) as the iterator code of the ready set must choose a QID from a larger set of ready QIDs.

## VI. Related Work

**Memory monitoring**. There are various hardware-assisted memory monitoring proposals for reliability and security applications [32], [75], [94], [97], [101], none of which is readily usable for SDPs. We consider one of the most general-purpose designs for a more detailed comparison: ECMon [75] is able to monitor various cache events (e.g., invalidation) for different ranges of addresses, specified in multiple entries of a *per-core* event descriptor table. Each entry corresponds to a handler function. However, ECMon does not provide any mechanism to keep certain cache lines (i.e., queue doorbells) in the caches. Even if cache lines are assumed to be locked in the cache, the event descriptor table is a small associative structure, which cannot efficiently support O(1k) events for different doorbells. Furthermore, almost all of these proposals only provide a scheme for monitoring memory locations but no efficient mechanism to provide priority among ready events. In other words, the prior mechanisms at best replace only HyperPlane's monitoring set functionality. If multiple events are ready, handlers are called in the order the events are received (i.e., FIFO), or a bit-vector representing the ready events is passed to software. Similarly, HypePlane differentiates from list/queue-based locking schemes (such as MCS [68], CLH [29], [65], and QOLB [53]) in that they avoid spinning on a single lock location by forming a FIFO queue of the requesting processors, whereas HyperPlane operates on multiple I/O queues, servicing them based on a wide range of defined policies, rather than the FIFO order of work item arrivals in the queues. In SDPs, work items arrive at a high rate and the system must perform task scheduling for non-trivial loads, prioritizing the service order among queues.

**I/O software stacks**. Several works enhance interrupts by reducing corresponding overheads [48], [88], combining them with spin-polling as a hybrid notification mechanism [35], or bringing them to user level [22], [28], [73], [76], [93]. HyperPlane, on the other hand, avoids the overheads of interrupts and spin-polling altogether. Kernel-bypass software stacks enable user processes to directly communicate with I/O. In such systems, application and transport software are integrated via a library OS. IX [19], Arrakis [81], ZygOS [83], and Andromeda [33] are specialized networking data planes with different features—such as task stealing [83], task preemption [52], virtualization [33], [81]—while ReFlex [57] and PASTE [49] target storage devices. Demikernel [100] specifies I/O abstractions that a library OS should provide in general. Other systems—such as Snap [67] and Shenango [79]—deploy centralized microkernel-like software, which orchestrates data communication of applications and I/O. HyperPlane, as a notification accelerator, can benefit transport software implementations, especially in case of microkernel-based SDPs like Snap [67] and Shenango [79].

**Data plane optimizations**. Prior works have proposed solutions to improve performance and efficiency of SDPs. DDIO [5], CacheDirector [38], and FlexNIC [55] optimize data transfer between I/O and CPU. Halo [99] proposes a near-cache accelerator for network packet flow classification. Compute-capable I/O devices, such as smart NICs/SSDs [8], [40], [64], and accelerators [43], [46], [58], [59], [82] are used to offload data plane operations from CPU. Particularly, hardware-managed transport protocols by RDMA NICs or SmartNIC-based network flow processing can ease tasks of data plane cores [30], [40], [54], [77]. Memory copy accelerators can also be used in SDPs for faster data movement [51], [66]. While HyperPlane, as a flexible centralized data plane, is compatible with commodity devices and protocols, it can leverage these proposals to further improve data plane performance.

## VII. Conclusion

In this work, we presented and evaluated *HyperPlane*, a hardware notification acceleration subsystem and programming model, which allows SDPs to efficiently monitor many I/O queues for work arrival. HyperPlane brings queue scalability, by avoiding spin-polling empty I/O queues unlike software-only designs, and work proportionality, by halting execution when I/O queues are idle. Furthermore, HyperPlane facilitates efficient sharing of queues across cores, enabling the strong properties of scale-up queuing. HyperPlane's programming model centers on the `QWAIT` instruction, which either returns a ready queue or halts execution. HyperPlane's microarchitecture comprises a *monitoring set*, which watches I/O queues for work arrival, and a *ready set*, the key component in HyperPlane's design that realizes various service policies, prioritization of ready queues, and work distribution among cores. We showed that HyperPlane improves peak throughput and tail latency by 4.1× and 16.4×, respectively, as compared to a modern spinning SDP.

## Acknowledgment

We thank the anonymous reviewers for their valuable feedback. We especially thank Adam Belay and Saman Amarasinghe from MIT, Marc de Kruijf and Shrijeet Mukherjee from Google, and Geoffrey Blake from Amazon Web Services for their insightful suggestions that helped improve this work. This work was supported by the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA. This work was done while Amirhossein Mirhosseini was an ADA center visiting researcher at MIT.

## References

[1] "Reaching the summit with inifiband," 2018. [Online]. Available: https://bit.ly/2Y3vfz3

[2] "Intel® 64 and ia-32 architectures software developer's manual," Oct. 2019, volume 2B: Instruction Set Reference.

[3] "Data plane development kit (dpdk)," 2020. [Online]. Available: https://www.dpdk.org/

[4] "The go programming language," 2020. [Online]. Available: https://golang.org/

[5] "Intel data direct i/o technology," 2020. [Online]. Available: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html/

[6] "Intel intelligent storage acceleration library," 2020. [Online]. Available: https://software.intel.com/en-us/isa-l

[7] "Intel optane technology," 2020. [Online]. Available: http://www.intel.com/optane/

[8] "Samsung smartssd computational storage drive," 2020. [Online]. Available: https://samsungatfirst.com/smartssd/

[9] "Samsung z-ssd," 2020. [Online]. Available: https://www.samsung.com/semiconductor/ssd/z-ssd/

[10] "Scaling in the linux networking stack," 2020. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt

[11] "T6 crypto offload," 2020. [Online]. Available: https://www.chelsio.com/crypto-offload/

[12] "urdma: User-space software rdma," 2020. [Online]. Available: https://github.com/zrlio/urdma/

[13] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 631–644.

[14] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, *Remote Memory in the Age of Fast Networks*. New York, NY, USA: Association for Computing Machinery, 2017, p. 121–127.

[15] T. Barbette, G. P. Katsikas, G. Q. Maguire, and D. Kostić, "Rss++: Load and state-aware receive side scaling," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, ser. CoNEXT '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 318–333.

[16] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, p. 48–54, Mar. 2017.

[17] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, Dec. 2007.

[18] A. Beaumont-Smith and C.-C. Lim, "Parallel prefix adder design," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, ser. ARITH '01. USA: IEEE Computer Society, 2001, p. 218.

[19] 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 *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 49–65.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011.

[21] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian, "The end of slow networks: It's time for a redesign," *Proc. VLDB Endow.*, vol. 9, no. 7, p. 528–539, Mar. 2016.

[22] A. Bracy, K. Doshi, and Q. Jacobson, "Disintermediated active communication," *IEEE Computer Architecture Letters*, vol. 5, no. 2, pp. 15–15, 2006.

[23] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE transactions on Computers*, no. 3, pp. 260–264, 1982.

[24] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.

[25] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "Raid: High-performance, reliable secondary storage," *ACM Computing Surveys (CSUR)*, vol. 26, no. 2, pp. 145–185, 1994.

[26] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the killer microsecond," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 627–640.

[27] S. R. Chowdhury, M. A. Salahuddin, N. Limam, and R. Boutaba, "Rearchitecting nfv ecosystem with microservices: State of the art and research challenges," *IEEE Network*, vol. 33, no. 3, pp. 168–176, 2019.

[28] J. Chung and K. Strauss, "User-level interrupt mechanism for multicore architectures," Aug. 28 2012, uS Patent 8,255,603.

[29] T. Craig, "Building fifo and priority-queuing spin locks from atomic swap," Tech. Rep., 1993.

[30] A. Daglis, M. Sutherland, and B. Falsafi, "Rpcvalet: Ni-driven tail-aware balancing of $\mu$s-scale rpcs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 35–48.

[31] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[32] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A flexible information flow architecture for software security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 482–493.

[33] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat, "Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 373–387.

[34] G. K. Dorai and D. Yeung, "Transparent threads: Resource sharing in smt processors for high single-thread performance," in *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '02. USA: IEEE Computer Society, 2002, p. 30.

[35] C. Dovrolis, B. Thayer, and P. Ramanathan, "Hip: Hybrid interrupt-polling for the network interface," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 4, p. 50–60, Oct. 2001.

[36] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis, "Unlocking energy," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 393–406.

[37] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, "Generic routing encapsulation (gre)," RFC 2784, March, Tech. Rep., 2000.

[38] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić, "Make the most out of last level cache in intel processors," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019.

[39] H. Fatih Ugurdag and O. Baskirt, "Fast parallel prefix logic circuits for n2n round-robin arbitration," *Microelectron. J.*, vol. 43, no. 8, p. 573–581, Aug. 2012.

[40] 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 *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66.

[41] S. Frankel, R. Glenn, and S. Kelly, "The aes-cbc cipher algorithm and its use with ipsec," 2003.

[42] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud  edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–18.

[43] Y. Go, M. Jamshed, Y. Moon, C. Hwang, and K. Park, "Apunet: Revitalizing gpu as packet processing accelerator," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. USA: USENIX Association, 2017, p. 83–96.

[44] H. Golestani, A. Mirhosseini, and T. F. Wenisch, "Software data planes: You can't always spin to win," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 337–350.

[45] P. Gupta and N. McKeown, "Designing and implementing a fast crossbar scheduler," *IEEE Micro*, vol. 19, no. 1, p. 20–28, Jan. 1999.

[46] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: A gpu-accelerated software router," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 195–206.

[47] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 489–504.

[48] W. Hofer, D. Lohmann, and W. Schroder-Preikschat, "Sleepy sloth: Threads as interrupts as threads," in *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*, ser. RTSS '11. USA: IEEE Computer Society, 2011, p. 67–77.

[49] M. Honda, G. Lettieri, L. Eggert, and D. Santry, "PASTE: A network programming interface for non-volatile main memory," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 17–33.

[50] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "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: USENIX Association, 2014, pp. 489–502.

[51] X. Jiang, Y. Solihin, L. Zhao, and R. Iyer, "Architecture support for improving bulk memory copying and initialization performance," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 169–180.

[52] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for $\mu$second-scale tail latency," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 345–360.

[53] A. Kägi, D. Burger, and J. R. Goodman, "Efficient synchronization: Let them eat qolb," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 170–180.

[54] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram rpcs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 185–201.

[55] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High performance packet processing with flexnic," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 67–81.

[56] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "Freeflow: Software-based virtual RDMA networking for containerized clouds," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 113–126.

[57] A. Klimovic, H. Litz, and C. Kozyrakis, "Reflex: Remote flash $\approx$ local flash," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, pp. 345–359.

[58] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1–14.

[59] H. Li, M. Hao, S. Novakovic, V. Gogte, S. Govindan, D. R. K. Ports, I. Zhang, R. Bianchini, H. S. Gunawi, and A. Badam, "Leapio: Efficient and portable virtual nvme storage on arm socs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 591–605.

[60] J. Li and B. Li, "Erasure coding for cloud storage systems: A survey," *Tsinghua Science and Technology*, vol. 18, no. 3, pp. 259–272, June 2013.

[61] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: Association for Computing Machinery, 2009, p. 469–480.

[62] T. Li, A. R. Lebeck, and D. J. Sorin, "Spin detection hardware for improved management of multithreaded systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 6, pp. 508–521, 2006.

[63] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 267–278.

[64] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading distributed applications onto smartnics using ipipe," in *Proceedings of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '19. New York, NY, USA: ACM, 2019, pp. 318–333.

[65] P. S. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in *Proceedings of the 8th International Symposium on Parallel Processing*. USA: IEEE Computer Society, 1994, p. 165–171.

[66] H. Mao, "Hardware acceleration for memory to memory copies," Master's thesis, EECS Department, University of California, Berkeley, Jan 2017. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-2.html

[67] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat, "Snap: A microkernel approach to host networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 399–413.

[68] J. M. Mellor-Crummey and M. L. Scott, "Scalable reader-writer synchronization for shared-memory multiprocessors," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 106–113.

[69] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing server efficiency in the face of killer microseconds," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2019, pp. 185–198.

[70] A. Mirhosseini and T. F. Wenisch, "The queuing-first approach for tail management of interactive services," *IEEE Micro*, vol. 39, no. 4, p. 55–64, Jul. 2019.

[71] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch, "Express-lane scheduling and multithreading to minimize the tail latency of microservices," in *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2019, pp. 194–199.

[72] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch, "Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 207–219.

[73] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood, "Coherent network interfaces for fine-grain communication," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ser. ISCA '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 247–258.

[74] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[75] V. Nagarajan and R. Gupta, "Ecmon: Exposing cache events for monitoring," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 349–360.

[76] G. Neiger and R. M. Sankaran, "Delivering interrupts to user-level applications," Mar. 20 2018, uS Patent 9,921,984.

[77] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out numa," ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 3–18.

[78] F. Oboril and M. B. Tahoori, "Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level," in *Proceedings of the 2012 42Nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, ser. DSN '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–12.

[79] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 361–378.

[80] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *European Symposium on Algorithms*. Springer, 2001, pp. 121–133.

[81] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, Nov. 2015.

[82] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch, "Optimus prime: Accelerating data transformation in servers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1203–1216.

[83] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: ACM, 2017, pp. 325–341.

[84] M. Rogawski, E. Homsirikamol, and K. Gaj, “A novel modular adder for one thousand bits and more using fast carry chains of modern fpgas,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–8.

[85] D. Sanchez and C. Kozyrakis, “The zcache: Decoupling ways and associativity,” in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’43. USA: IEEE Computer Society, 2010, p. 187–198.

[86] R. Schöne, D. Molka, and M. Werner, “Wake-up latencies for processor idle states on current x86 processors,” *Computer Science-Research and Development*, vol. 30, no. 2, pp. 219–227, 2015.

[87] M. L. Scott, “Shared-memory synchronization,” *Synthesis Lectures on Computer Architecture*, vol. 8, no. 2, pp. 1–221, 2013.

[88] R. T. Short, J. M. Parchem, and D. N. Cutler, “Method and apparatus for reducing the rate of interrupts by generating a single interrupt for a group of events,” Jan. 13 1998, uS Patent 5,708,814.

[89] D. J. Sorin, M. D. Hill, and D. A. Wood, “A primer on memory consistency and cache coherence,” *Synthesis lectures on computer architecture*, vol. 6, no. 3, pp. 1–212, 2011.

[90] A. Sriraman and A. Dhanotia, “Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 733–750.

[91] A. Sriraman and T. F. Wenisch, “$\mu$ suite: a benchmark suite for microservices,” in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 1–12.

[92] A. Sriraman and T. F. Wenisch, “$\mu$tune: Auto-tuned threading for OLDI microservices,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 177–194.

[93] K. Strauss and J. Chung, “Flexible notification mechanism for user-level interrupts,” Oct. 9 2012, uS Patent 8,285,904.

[94] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, “Secure program execution via dynamic information flow tracking,” in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. New York, NY, USA: Association for Computing Machinery, 2004, p. 85–96.

[95] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, “Resq: Enabling slos in network function virtualization,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 283–297.

[96] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, “Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor,” in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ser. ISCA ’96. New York, NY, USA: ACM, 1996, pp. 191–202.

[97] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, “Memtracker: Efficient and programmable support for memory access monitoring and debugging,” in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 273–284.

[98] Z. Yang, C. Liu, Y. Zhou, X. Liu, and G. Cao, “Spdk vhost-nvme: Accelerating i/os in virtual machines on nvme ssds via user space vhost target,” in *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, Nov 2018, pp. 67–76.

[99] Y. Yuan, Y. Wang, R. Wang, and J. Huang, “Halo: Accelerating flow classification for scalable packet processing in nfv,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 601–614.

[100] I. Zhang, J. Liu, A. Austin, M. L. Roberts, and A. Badam, “I’m not dead yet! the role of the operating system in a kernel-bypass era,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 73–80.

[101] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, “Iwatcher: Efficient architectural support for software debugging,” in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA ’04. USA: IEEE Computer Society, 2004, p. 224.