Hermes: Evaluating Ensō Schedulers for High-Performance Networking

Kaajal Gupta (kaajalg)
Research Advisor: Justine Sherry (sherry)

School of Computer Science
Carnegie Mellon University
United States
April 26, 2024

Hermes: Evaluating Ensō Schedulers for High-Performance Networking

Kaajal Gupta (kaajalg)

Abstract

Kernel bypass has recently gained popularity as a way to achieve high-performance networking. However, it pays for its improvements in performance with CPU efficiency. For applications to respond quickly to packets and have high performance, they must be continually polling the NIC, never yielding to the kernel–leading to low CPU efficiency. Alternatively, CPU efficiency can be kept high by having the process scheduler interpose between the NIC and applications, letting it deschedule idle applications—which leads to low performance due to data movement. In this project, a system of kernel bypass is introduced that reconciles performance and CPU efficiency. The key idea behind this system is to have the process scheduler interpose on the network control plane while letting the applications exchange data directly with the NIC (dataplane). This design is enabled by Ensō, a recent proposal for a new NIC interface that detaches data from notifications of the data arriving. Different designs incorporating Ensō are evaluated on a variety of metrics to determine how to best leverage the control plane information coming into the process scheduler.

1 Introduction

In order to communicate with the network interface card (NIC), applications transfer data to the network through the kernel network stack—this enables the kernel scheduler to directly wake applications when data has been received, swapping out applications that are idle. However, this kernel stack has emerged as a bottleneck when accessing the network: context switching between kernel and user space, copying memory between kernel and user space, and delivering interrupts to applications incur overhead, leading to slower data transmission and increased latency. Thus, it has become common to adopt kernel bypass as a method for applications to access the NIC: circumventing the kernel and having apps directly share packet buffers with the NIC, processing packets with their own network stacks.

Several kernel bypass technologies have been developed, such as DPDK [2] and Solarflare's OpenOnload [12]. However, in these approaches, applications are pinned to cores isolated from the kernel scheduler, and repeatedly poll the NIC's queues for incoming packets–reducing CPU efficiency as idle applications that are busy-polling cannot be replaced.

Thus, network functions and micro services that rely on low-latency RPCs face an inherent tension between performance and CPU efficiency. In order for applications to respond quickly to packets, applications must be continually polling the NIC, acting immediately upon the arrival of new packets. Moreover, packet bursts are difficult to predict and cores may need to busy-poll for an unknown period of time before the next packets arrive. Alternatively, CPU efficiency can be kept high by having the scheduler swap out applications that are waiting on packets, sacrificing performance.

Existing systems of running applications across an array of cores do a poor job at achieving high CPU efficiency while also maintaining microsecond-scale tail latency. For instance, Linux can support low tail latency only when there are idle cores available for allocation to handle incoming requests [6]. This thus leads to low CPU efficiency. ZygOS [10] implements a specialized operating system and uses kernel bypass, while also rebalancing work among cores using shared-memory data structures, inter-processor interrupts (IPIs), and multi-queue NICs. However, it faces low CPU efficiency as it too relies on spin-polling the NIC queues for new packets, while also being unable to reallocate cores among applications quickly.

Shenango [9] was developed as a means of addressing this CPU efficiency and tail latency tension. It similarly uses kernel bypass, but also employs an additional core, called the IOKernel, that handles all incoming and outgoing packets, distributing packets to cores running certain applications. The IOKernel here is able to reallocate cores and detect congestion (which here refers to core contention rather than network congestion) efficiently by polling all of the cores at a 5 μs time interval. However, there is high overhead from copying packets from the IOKernel to applications' packet buffers. Caladan [4] addresses the issue of high overhead from copying packets to applications' packet buffers, and instead has the IOKernel only be responsible for congestion detection and core reallocation. It has the NIC directly steer packets into applications' packet buffers, with the applications updating the NIC flow steering rules on movement across cores. However, the IOKernel can quickly become a bottleneck to scalability in these systems, particularly due to the need to check for congestion among quickly-moving packet queues. Moreover, Caladan faces issues with runtime efficiency due to the need to update NIC flow table rules regularly, as will be explored in Section 2.

Ensō [11] is a new streaming NIC-to-application interface designed to better take advantage of how NICs and applications currently interact with each other. Most NICs provide an interface of packet buffers for delivering requests, as seen in Figure 1.

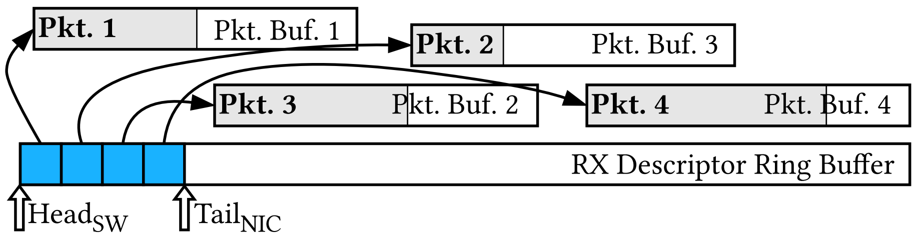


Figure 1: Traditional Packetized Interface: Packet buffers can be placed arbitrarily in memory. Extracted from [11].

The length of these packet buffers is dictated by software–for example, Linux uses packet buffers of length 1537 bytes while DPDK [2] uses 2 kB length packet buffers. Ensō demonstrates that this packetized interface has three key issues:

1. **Splitting and re-joining packet buffers**: Many NIC offloads can take inputs and outputs that can vary in size, thus leading to packet buffers of varying lengths. As the length of packet buffers is dictated by software, this leads to the need for the NIC or software to split messages that are too large into multiple packet buffers, and applications would need to join packets that were split across multiple buffers. This can also lead to overheads with copying data from buffers to streams.

2. **Poor cache interaction**: As the packet buffers can be scattered across memory, the effectiveness of prefetchers and CPU optimizations in lessened: as the CPU will be unable to predict where the next interaction with memory will occur.

3. **Metadata overhead**: In the packetized interface, up to 39% of the available bandwidth is spent just transferring per-packet metadata for small messages. Thus, applications that use small messages are bottlenecked by PCIe, hindering scalability.

Ensō employs a streaming abstraction that can be used to communicate arbitrary-sized chunks, by replacing ring-buffers that are filled with metadata descriptors with ring buffers that are filled with the packet data itself. In order to notify applications and the NIC of where packet data begins and ends in the ring buffer, it also uses a separate ring buffer of packet 'notifications.' This provides a natural control and data plane split: where the control plane is the part of the network responsible for directing and routing packets, while the data plane is the transmission of the packets themselves.

An Ensō-based NIC can saturate a 100 Gbps link with minimum-sized packets using a single core, and is capable of reducing latency up to 43%. For this project, we are interested in utilizing the capabilities of Ensō to implement a fast scheduler, combining the runtime efficiencies of Shenango and the data copy avoidance of Caladan, while also avoiding checking fast-moving packet queues for congestion. The next sections will delve into how Ensō is designed, and how its control-data plane separation can be utilized to make a robust scheduling system.

2 Background

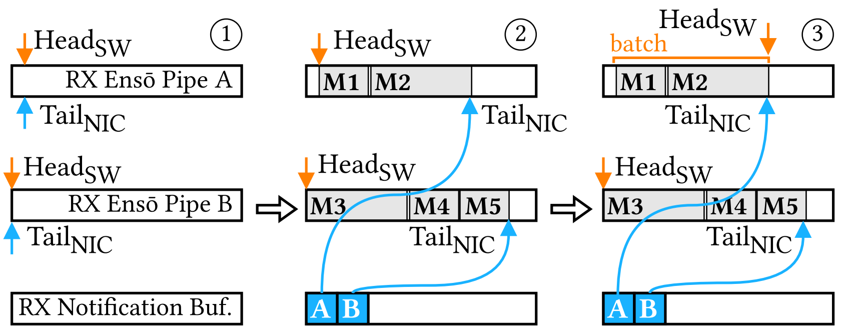


Figure 2: Receiving batches of messages in two Ensō Pipes. Extracted from [11].

Ensō functions with two types of ring buffers: Ensō Pipes and Notification Buffers. Ensō Pipes carry the raw packet data, while Notification Buffers carry the notifications of pointer updates to the Ensō Pipes. When the NIC receives packets, it uses DMA (direct-memory access) to enqueue them in the Ensō Pipes, while also enqueueing a notification in the associated notification buffer. However, there does not necessarily need to be one notification per packet: in fact, this is often not the case except for slow loads. Instead, Ensō employs the concept of reactive notifications: that is, the NIC only sends a notification for queued packets when signaled by software itself. This can be seen in 2, as notifications A and B both deliver information on a batch of packets in Ensō Pipes A and B.

How does the NIC know when software is requesting more packets in Ensō? When software consumes packets, it writes to a Memory-Mapped I/O (MMIO) register, notifying the NIC: which can then check if there has been new packets in that Ensō Pipe and can send a new notification if that is the case. This thus enables the rate of notification arrivals to adapt to the speed at which software is consuming data. However, it should be noted that this reactivity can have its downsides: particularly with latency-sensitive applications. In order for software to receive a new notification upon consuming data, it will have to wait for a PCIe RTT before receiving the next notification. Ensō attempts to mitigate this by introducing a mechanism called notification prefetching: which has software explicitly requesting new notifications from the NIC just before returning data that was consumed to the application. But, this has the disadvantage of potentially receiving a notification with no new information if nothing new has arrived for that Ensō Pipe.

How can one use Ensō to create a system capable of balancing low tail latency with high CPU efficiency? First, we must look at how existing state-of-the-art implementations are designed, and how we can modify them to accommodate Ensō. Shenango runs applications in user-level runtimes, which are high-level programming abstractions that communicate with the IOKernel to facilitate core allocations. Shenango achieves high CPU efficiency by utilizing an additional core called the IOKernel, that is responsible for:

1. receiving all packets for runtimes from the NIC,
2. forwarding these packets to the correct cores running the runtime the packets were meant for,
3. detecting congestion in the runtimes,

4. and reallocating cores based on this congestion detection.

When the IOKernel receives packets from the NIC, it calculates a hash over some packet header fields to determine which core to direct the packet. When the runtimes receive the packets, they copy the payload to their own buffer and release the sent buffer back to the IOKernel. The IOKernel detects congestion by iterating at every 5 μs interval over all packet queues and all thread queues for each core: if there exists a thread or packet that has not moved between the last two iterations, it determines that the runtime is congested. From there, it grants a new core. Moreover, it has the runtimes yield cores when they are idle, and can thus either grant congested runtimes idle cores or preempt the cores of runtimes that have more cores than guaranteed.

However, Shenango brings in the same data movement overheads that kernel bypass aimed to avoid, as it involves copying of packets from the IOKernel buffers to the application buffers. In other words, the dataplane and the control plane travel through the IOKernel. Additionally, Shenango maintains a linear-time congestion detection algorithm, proportional to the number of cores: this draws into question if there is a need for periodic polling, and if it were possible to enable greater reactivity in congestion detection?

Caladan is derived from Shenango, and implements a separation of the dataplane from the IOKernel. The NIC itself is responsible for packet steering: when core reallocations occur, the IOKernel updates the NIC flow tables. However, updating the NIC flow tables can reduce runtime efficiency by up to 5%, as observed when comparing Caladan to Shenango [8]. An alternative strategy to avoid flow table updates would be to keep the mapping between NIC queues and applications fixed. However, this requires applications to check multiple queues from the same thread or to have multiple threads share the same queue—which would add overhead due to cache invalidations and locking. Caladan similarly uses a congestion detection algorithm that involves polling runtime packet and thread queues for movement (while also integrating other useful signals, such as LLC miss rates and memory bandwidth). This again calls into question how well the algorithm would scale. Moreover, the need to periodically poll application buffers can be inefficient when loads are stable and slow when loads are changing.

How can Ensō be used with a framework such as Shenango or Caladan's? By leveraging Enso pipes, the control and data planes can be split naturally—an IOKernel can maintain a notification buffer that the NIC fills in. This core can then forward notifications to the relevant application thread. It can act as a packet steering mechanism: directing notifications for packets with certain queue IDs to certain application threads. Additionally, it can check when application threads yield due to inactivity, and handle the scheduling of new threads on those cores. Moreover, the number of channels to check for updates would be limited to the TX notification queues of each core and the notification buffer from the NIC. This design can avoid the runtime inefficiencies of Caladan by not updating the NIC flow table rules, while also avoiding the overhead of copying packets in Shenango.

In this thesis, we explore different ways of integrating and taking advantage of the control-data plane split that Ensō provides, and considering alternate methods of congestion detection by utilizing the information provided by notifications rather than by polling runtime queues.

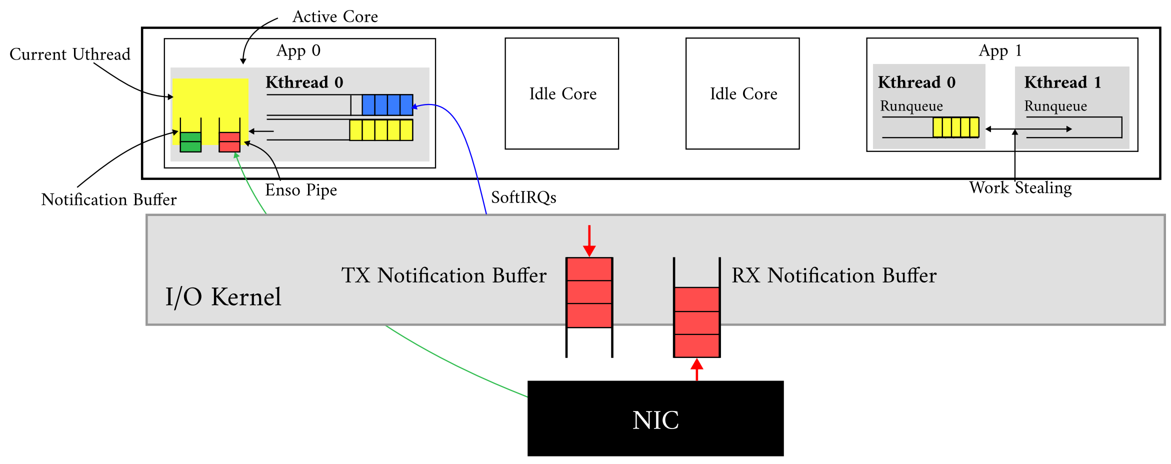


Figure 3: Hermes Baseline Architecture

3 Design

Hermes explores the challenge of integrating Ensō into a congestion detection algorithm by tracking one key signals for each application: the maximum thread queueing delay. It does so by dedicating a busy-spinning core to a centralized software entity called the IOKernel. This IOKernel serves as an semi-intermediary between the NIC and applications, with notifications being forwarded from the NIC to the IOKernel to applications, and data being forwarded directly from the NIC to applications. The following sections explore how the IOKernel can be used to detect congestion by evaluating implementations with different design considerations. Hermes's design shares many similarities with Shenango and Caladan, with the threading, uthread scheduling, and core allocation systems being the same. The key differences lie in how the data travels to the uthreads, and how the IOKernel detects congestion.

Application logic runs in per-application runtimes, which communicate with the IOKernel via shared memory queues. At start-up, each runtime spawns multiple kernel-level threads (i.e., pthreads) up to the maximum number of cores that applications may use. Each kernel-level thread (kthread) has a local runqueue that is populated with lightweight user-level threads (uthreads). Work is balanced in a decentralized manner with work stealing.

3.1 IOKernel

The IOKernel runs on a dedicated core and provides two main functions:

1. It forwards Ensō notifications to runtime uthreads from the NIC, and transmits notifications from these uthreads back to the NIC. The IOKernel continually polls its own notification buffer for notifications from the NIC, and upon reception, immediately places them on uthreads' notification buffers. It also forwards Ensō notifications from uthreads to the NIC by polling queues from all cores.

2. It coordinates the initialization of new Ensō devices.

3. It determines how many cores to allocate to each application, and which cores to allocate to each application based on the rate at which uthreads are run from being placed on a kthread's runqueue.

3.1.1 Runtime Communication

The IOKernel communicates with runtime kthreads via shared memory queues. Every core running an application maintains a single-producer single-consumer queue to communicate with the IOKernel. Runtimes communicate with the IOKernel on a variety of events, including:

1. **Allocation of Notification Buffers**: When creating a new Ensō device, runtimes create the memory regions for notification buffers themselves through calls to the Ensō API. Additionally, they send a message in these shared queues to the IOKernel to receive their notification buffer ID. The IOkernel keeps track of a monotonically increasing count of the notification buffer, thus providing each notification buffer created with a unique ID.

2. **MMIO Reads and Writes**: In plain Ensō, runtimes send updates on the memory regions, heads, and tails of Ensō pipes and notification buffers by writing to MMIO registers, notifying the NIC. In Hermes, runtimes send these updates in the *same way for Ensō pipes*, by directly writing to these registers. However, *notification buffer updates are sent to the IOKernel* via these queues, so it will be notified when new notifications must be transmitted (when the TX Notification Buffer tail advances), and when notifications have been consumed (when the RX Notification Buffer head advances).

3. **Accessing and Setting Configuration Values**: Runtimes communicate through the queues to the IOKernel to access and update global configuration values, such as the number of fallback queues (pipes that receive packets that do not match any explicit bind rules) and the round robin status of packets (determining how to send these packets to fallback queues) arriving in Ensō Pipes.

In certain implementations of Hermes, there are some additional events that are communicated to the IOKernel, such as uthread parks and switches: this will be elaborated upon in section 3.3.2.

3.1.2 SoftIRQ Queues

In addition to these shared memory queues per core, the IOKernel also shares a SoftIRQ queue per kthread (for each runtime). The purpose of this queue is to communicate any new event—a ’softIRQ’—that the IOKernel has received to the kthreads. When the IOKernel receives a new RX notification, for example, it first checks which runtime the notification is for by checking what Ensō Pipe the notification is referencing. Then, it checks the active kthreads for the runtime: if there are no active kthreads, it immediately allocates a core for the runtime. If there are active kthreads, it sends the softIRQ to one of the kthreads, preferring to send notifications designated for a particular uthread to the same kthread. This can help improve the locality of cache references and avoid having uthreads jump between cores, which would occur if the IOKernel sprayed softIRQs to all kthreads in a round-robin fashion. The content of the softIRQs can vary across the Hermes implementations depending on how much information the IOKernel has, as described in Section 3.3.2.

3.1.3 Core Allocation

The IOKernel must make decisions on core allocations quickly, as time that it spends on core allocations is time it could be spending forwarding notifications and receiving messages from runtimes. As with Shenango, it decouples its two decisions: Hermes first decides which application to grant an additional core to, and then decides which core to grant. Note that this system of determining which core to allocate and which application to grant a core to is from Shenango: I simply extended it to use Ensō. We elaborate briefly on the system here for clarity. We use the **ksched** kernel module, implemented in Caladan, to shift kthreads among cores and preempt kthreads efficiently. The IOKernel makes core reallocation decisions at a few points: when a runtime has yielded a core, when a new notification has arrived for a runtime without any running kthreads, and when it detects congestion in the queues of a runtime.

Let's consider the when the IOKernel needs to select an application to grant an additional core to, as seen in the case when a core is newly idle. The IOKernel maintains a group of overloaded apps and a group of bursting apps. The overloaded apps are those in need of a core, while the bursting apps have been granted more than their guaranteed number of cores. Overloaded apps are stored as the congestion detection algorithm detects congested apps that have already been granted their guaranteed number of cores. Thus, the IOKernel grants cores to the oldest overloaded app–and if there are none, it simply leaves the core idle.

How does the IOKernel determine which core to grant a congested application? It can either grant it an idle core, or grant it a core of a bursting application. When granting it the core of a bursting application, it preempts the application by sending it an Inter-Processor Interrupt (IPI), kicking it from the core, and allowing it to yield cleanly before running the congested runtime's kthread on it.

3.2 Runtimes

On launching, the runtimes first inform the IOKernel of their initialization via a control socket: the IOKernel can then allocate a core for it. Following this, the runtimes launch kthreads, as many as the number of cores they would like to be able to run on. From there, the runtimes can provide a main uthread for the kthreads to spawn, and from there more uthreads may be spawned.

3.2.1 Scheduling and Stealing

The kthreads schedule uthreads on their cores using runqueues. When new uthreads are created, they are added to the current kthread's runqueue. The system follows cooperative, rather than preemptive, scheduling within each runtime: the uthreads can run for as long as they wish before yielding the core to other uthreads. The order of operations when a kthread determines what uthread to run next is as follows:

1. If it has been at least 50 μs since the last check of the softIRQ queue, the kthread jumps into the softIRQ handler thread.
2. If not, the kthread checks its own runqueue of uthreads: if there exist any uthreads on it, then it pops the oldest element and runs it.
3. If nothing is available in the runqueue, the kthread then checks its softIRQ queue for any new softIRQs: if they are present, then the kthread adds a softIRQ handler thread to its runqueue and jumps into that. The softIRQ handler thread processes up to 16 softIRQs from the queue at a time.
4. If nothing is available in the softIRQ queue, the kthread iterates over all other kthreads in the runtime, checking their runqueues and softIRQ queues, stealing work when available.

3.2.2 A Hybrid Backend

Plain Ensō uses an FPGA backend: that is, all information and updates on notification buffers and Ensō Pipes are directly communicated with the NIC. Additionally, toward the beginning of this project, another backend was created: the software backend, that has all notification buffer and Ensō pipe updates sent to the IOKernel to handle. For this project, a new backend was created: the hybrid backend. This backend has all notification buffer updates sent to the IOKernel, while all Ensō Pipe updates are communicated to the NIC. The IOKernel uses the FPGA backend. Ensō Pipes are configured in this backend such that their associated notification buffer in the NIC is the IOKernel's notification buffer. Thus, this results in the NIC directly sending packets to these Ensō Pipes while sending their corresponding notifications to the IOKernel's notification buffer, thus allowing a zero-copy system.

3.3 Design Alternatives

In this section, we explore the design choices we considered and implemented.

3.3.1 Congestion Detection

We speculated if there could be a way to leverage the information coming in and out of the IOKernel to detect congestion reactively rather than by iterating over all kthreads periodically. Thus, two methods of congestion detection were implemented:

Priority Queue As the IOKernel is kept as an intermediary between the NIC and applications for notifications, in one method, it was leveraged as keeping track of congestion. During the IOKernel's busy-spinning, it keeps track of the pending notifications and when they have been consumed, as well as pending uthreads and when they have been run: thus enabling itself to calculate the thread and packet queueing delay, orchestrating core reallocations.

How can the IOKernel keep track of uthread queueing delay? In this implementation, when uthreads yield due to a lack of work, they **send a message to the IOKernel** with the position of the last RX and TX heads of their notification buffers, as well as the next uthread they are switching to. Upon reception of this message, the IOKernel checks if there have been new RX notifications received or new TX notifications consumed since they yielded based on the values sent. If so, it immediately places the uthread back on the runqueue of one of its runtime's kthreads by sending a kthread a softIRQ. If not, when it does receive a new RX notification or TX consumption for the uthread, the IOKernel then sends the kthread the softIRQ indicating that the uthread is ready to be run. This architecture enables the uthread to be immediately placed on a kthread's runqueue on reception of any packets.

How would the IOKernel determine when to grant an application a core? In this implementation, it **maintains a min-heap of applications**, with the application at the top of the heap having the most uthread queueing delay. As the IOKernel is notified when a uthread is placed on the runnable queue and when it yields, it can maintain a queue of all pending runnable uthreads for each application separately. Figure 4 demonstrates this queue for a single application.

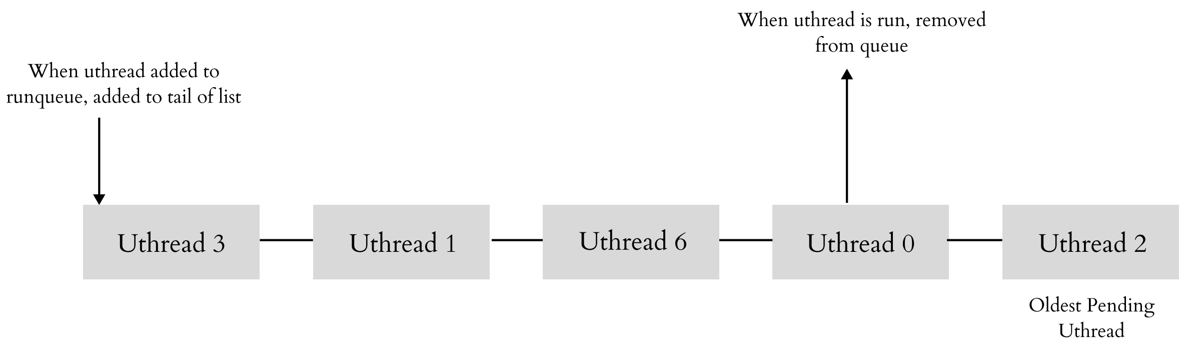

When uthread is run, removed from queue
When uthread added to runqueue, added to tail of list
Uthread 3
Uthread 1
Uthread 6
Uthread 0
Uthread 2
Oldest Pending Uthread

Figure 4: Application Uthread Organization

The priority for each application is kept as maximum of the time when the oldest pending uthread was added to the linked list and the time when the application was last granted a new core. This is to ensure that an application has time to adjust to its new core before detecting congestion again. Thus, the IOKernel can peek the top application in the heap and from its priority, can measure if more than 5 μs have gone by without any movement, granting a new core if so.

Note: In this implementation, the queueing delay we measured encapsulates both the softIRQ and the uthread queueing delay, as it measures the time from when the IOKernel adds the uthread to the softIRQ queue, to when the uthread is run by the kthread.

Periodic Iterations The system of iterating over all kthreads periodically and checking for uthread queueing delay was also implemented. As per Shenango, this comprises of checking heads and tails the softIRQ queue and the runqueue of each kthread of each runtime (including those kthreads that have yielded) periodically: which was kept to $5\mu s$. If there exists a uthread or notification on the queue that has not been consumed since the last check, the runtime is granted an additional core. Note that it is possible that there will be less movement in the softIRQ queues in Hermes in comparison to Shenango or Caladan as notifications can batch packets.

3.3.2 Detection of Pending Uthreads

Moreover, we also implemented two methods of detecting uthreads that have yielded due to waiting for incoming packets.

IOKernel tracking Waiting Uthreads In this implementation, the IOKernel receives notifications of when uthreads yield when waiting for new packets. From this information, when new notifications come in, the IOKernel can check if the uthread associated with the notification is waiting: if it is, then it can simply forward the uthread itself as a softIRQ to a kthread to add back to its runqueue. Thus, in this method, the uthreads themselves are softIRQs. However, this method has a drawback: there can be build up of uthreads yielding and the queue to inform the IOKernel of this can quickly become congested, as the IOKernel only consumes one notification off the queue at each iterations. However, there's also the added benefit of having the IOKernel immediately add notifications to the notification buffers of uthreads: and thus, if uthreads are running, then they could receive updates at a faster rate than if kthreads were to process the notifications and add them to the notification buffers.

Kthreads tracking Waiting Uthreads In this method, the IOKernel no longer receives a notification when uthreads yield: instead, it simply forwards the notifications it receives and the completions it processes to kthreads running a particular runtime. Thus, in this implementation, RX notifications and TX completions are the softIRQs. The kthreads are able to check if a uthread is waiting with a spinlock, and thus add them back to their runqueue to be run in the future.

3.4 Implementations

Thus, with these designs, three different implementations were finished: **Hermes Indirect**, **Hermes Semi-Direct**, and **Hermes Priority**.

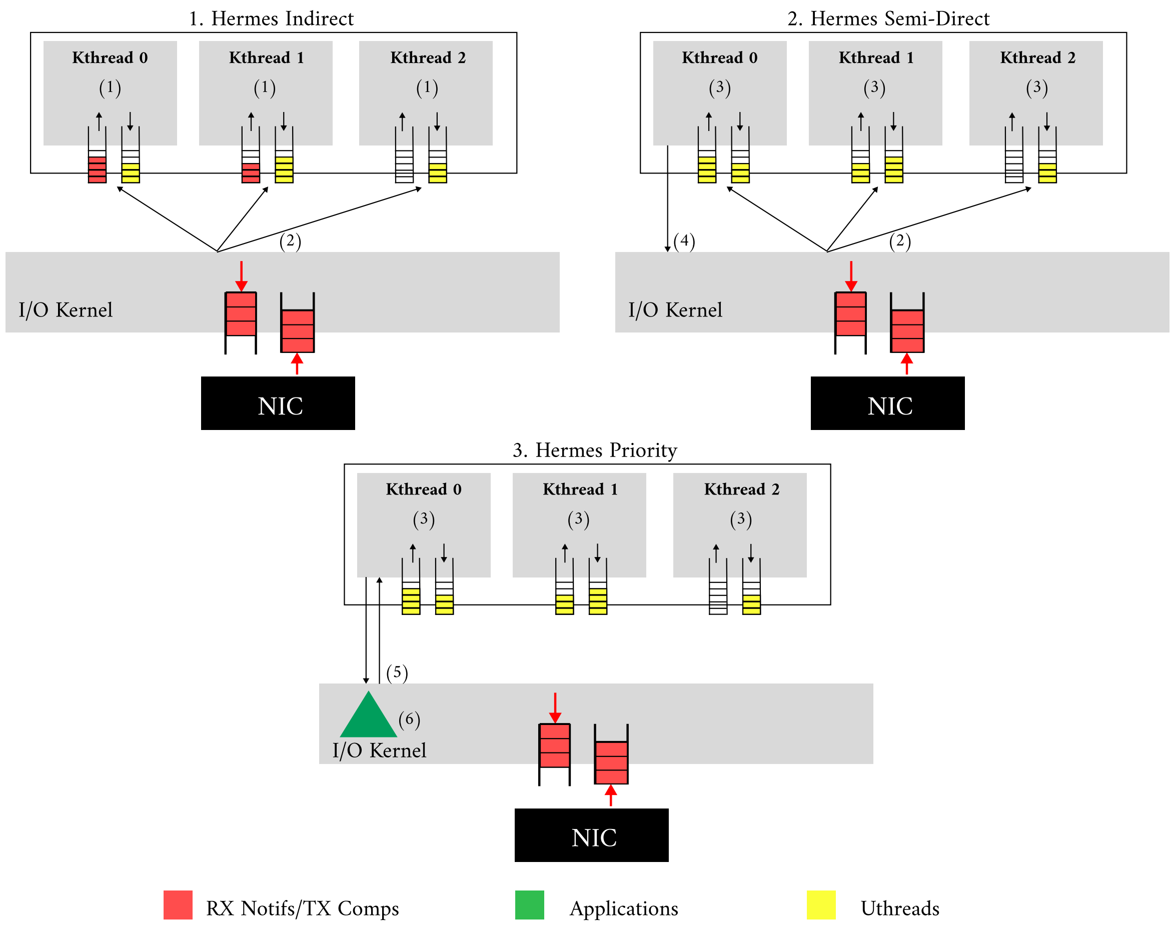


Figure 5: Various Hermes Implementations. (1) Kthreads receiving RX Notifications and TX Completions and adding Uthreads to their Runqueues based on this (2) The IOKernel iterating over all kthread queues to determine congestion (3) Kthreads receiving uthreads and adding them to their Runqueues (4) Kthreads notifying the IOKernel of uthread parks (5) Kthreads notifying the IOKernel of uthread switches and waiting for a response (6) The IOKernel maintaining a priority queue of applications based on the uthread switches.

Hermes Indirect incorporated having kthreads track waiting uthreads and the IOKernel performed periodic iterations to detect congestion in the kthreads' queues. When uthreads switched between each other, no new messages were sent to the IOkernel.

Hermes Semi-Direct has the IOKernel track waiting uthreads and the periodically iterate over the kthreads' queues to detect congestion. Importantly, when the uthreads yield **due to a lack of work**, they send a message to the IOKernel using the shared memory queues. The uthreads do not need to wait for a response from the IOKernel in this implementation.

Hermes Priority has the IOKernel track waiting uthreads and maintain a priority queue to detect congestion. When uthreads switch between each other in this implementation, they must send a message to the IOKernel with their uthread and the next uthread they are switching to: this is to ensure that the IOKernel has up-to-date information on when uthreads were added to runqueues and when they were actually run. When sending this message, kthreads must wait for a response from the IOKernel before continuing on. This is because of ordering: if the kthreads did not wait, messages about the same uthread could show up from various kthreads and thus could disrupt the IOKernel's information about the uthreads.

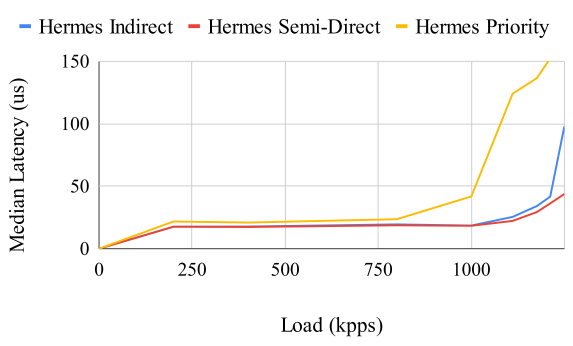
Note that there could not be an implementation with kthreads tracking waiting uthreads and the IOKernel maintaining a priority queue, as that would lead to the IOKernel only being able to detect notification congestion and not threading congestion (since it would not be notified of when uthreads switch anymore, and only of when RX notifications and TX completions are received and consumed).

4 Evaluation

4.1 CPU Efficiency and Latency

Firstly, the ability of Hermes's various implementations to achieve efficiency under different loads and service times was considered, while also being able to load-balance between latency-sensitive and batch throughput applications. This experiment involves using 100 uthreads and 100 connections that packets are being sent to. Moreover, a batch application was also run to keep CPU utilization at 100% and to evaluate how effective the system is at swapping out applications depending on when all cores are not needed to achieve low latency.

First, we ran an application that has a constant service time of $10\mu s$ per request: that is, for each packet, the runtime spins for this length of time before echoing the packet back to the client application that is generating packets.



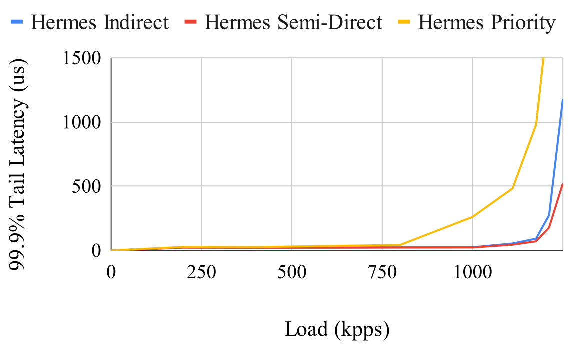


Figure 6: Load vs Median Latency (left) and 99.9% Latency (right)

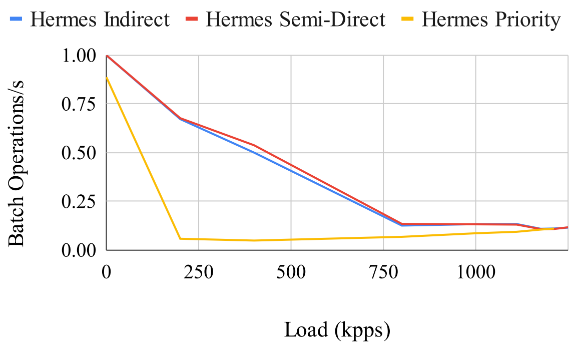
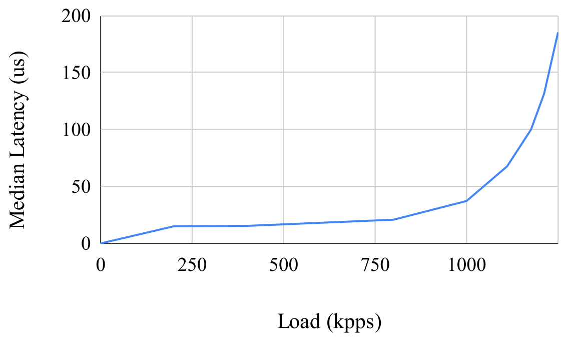


Figure 7: Load vs Batch Operations/s

First, analyzing Hermes Indirect vs. Hermes Semi-Direct: we observe that Hermes Semi-Direct performs better in terms of latency, achieving lower 99.9% and median latencies at higher loads. Hermes Semi-Direct's performance being better is likely due to the service time being present, and thus allowing the IOKernel enough time to process messages (such as uthread parks) from the runtimes. Hermes Priority performs significantly worse than Indirect and Semi-Direct: and this can largely be attributed to the fact that runtimes must wait for a response from the IOKernel on uthread switches, as proven in Section 4.4.3.

Interestingly, the number of batch operations in Hermes Priority is consistently almost none–implying that there almost no swapping of applications due to lower load. Upon further investigation, it was observed that at lower loads, the batch application does get run frequently–but gets preempted very quickly, as the priority queue algorithm very quickly detects congestion in the network application. With a congestion limit of 5 μs, and a variety of signals to process from all cores in order to maintain the priority queue and keep it up to date, it stands to reason that the priority queue would detect congestion in the queues, while they are in fact not congested.

Next, we also analyze the exponential case: in which the service times for each packet is exponentially distributed with a mean of 10 μs. Significantly worse performance was observed in this case, here we analyze the best-performing implementation for this experiment: Hermes Indirect (shown in Figure 9). However, Caladan and Shenango both are able to handle this case well, similar to how their uniform case is handled. Thus, this implies something amiss with the added changes of Ensō and the core reallocation system.



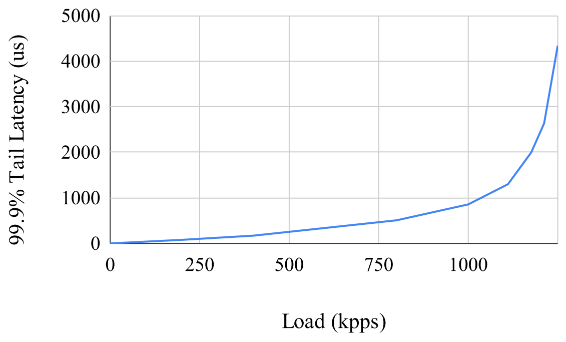


Figure 8: Hermes Indirect: Load vs Median Latency (left) and 99.9% Latency (right)

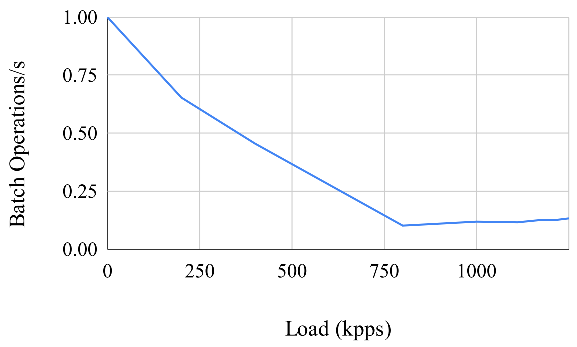


Figure 9: Hermes Indirect: Load vs Batch Ops/s

From further inspection, we saw that the performance with fewer uthreads was better in the exponential case–while in the constant case, performance got better with more uthreads, as the load is spread across threads and thus work stealing is more efficient. Moreover, we noticed that the time for a notification or completion to get to a uthread dramatically increased in the exponential case with more uthreads: and the largest component of that span of time was measured to be the waiting time of uthreads on the runqueues. This led us to believe that the reason why the exponential case was doing significantly worse was due to two key reasons:

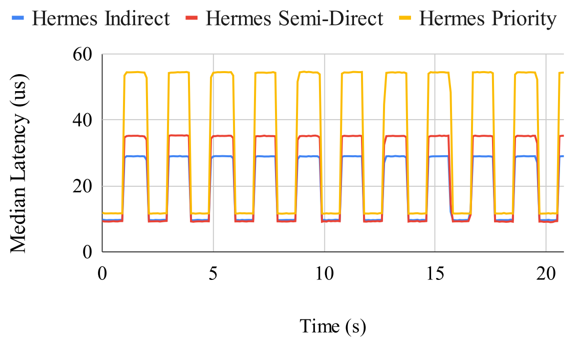
1. **Reactive notifications**: The added latency of a PCIe RTT after receiving a packet, particularly if the the service time for a packet is higher than 10 μs, would result in a higher latency.

2. **Re-adding uthreads to the queue on TX Completions**: On a TX completion, the IOKernel informs a selected active kthread of the occurrence. Then, the kthread would add the uthread for which the TX completion occurred back to the runqueue. However, in Caladan and Shenango, completions do not lead to uthreads being re-added: rather, the kthreads simply free the buffer that has completed transmitting, and move on. Uthreads were re-added to runqueues on completions for them themselves to use their Ensō devices to process the completions, avoiding locks and contention over the devices. This can lead to up to double the uthreads being added to runqueues, inflating the time for a notification to actually reach a uthread.

It is suspected that the higher variability in service times associated with the exponential distribution helped to bring out these issues clearly. To address this, I aim in the future to use Ensō with a lock and thus have the kthreads directly modify the Ensō devices rather than having the need to add uthreads to the notification queue, and evaluate the performance in comparison to Shenango.

4.2 Resilience to Bursts in Load

The ability of the programs to react quickly and efficiently to bursty loads is an important aspect of keeping tail latency as low as possible. In order to test this, an echo server with $1\mu s$ of fake work per request was run and loads varied from 100,000 requests per second to 5 million requests per second at 1 second intervals. Moreover, batch work was run continually to keep CPU utilization at 100%. 50 uthreads with 1 connection each were run on 14 cores, with continually streaming packets.



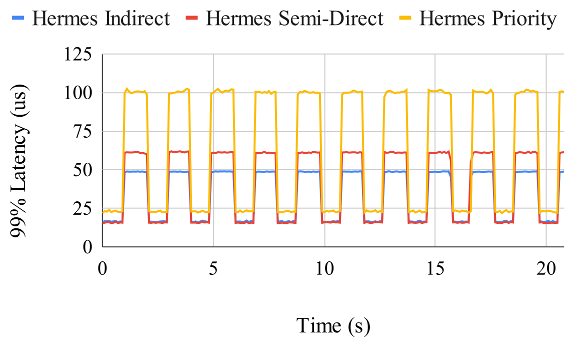
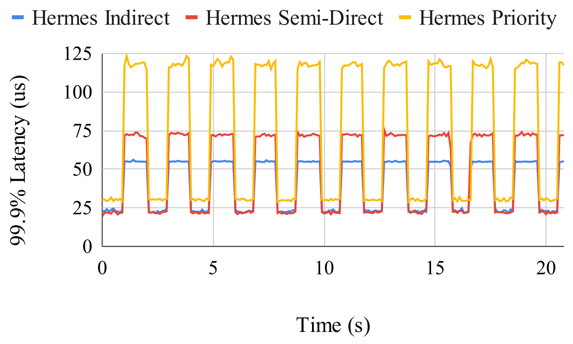


Figure 10: Median Latency (left) and 99% Latency (right) over Time for Implementations



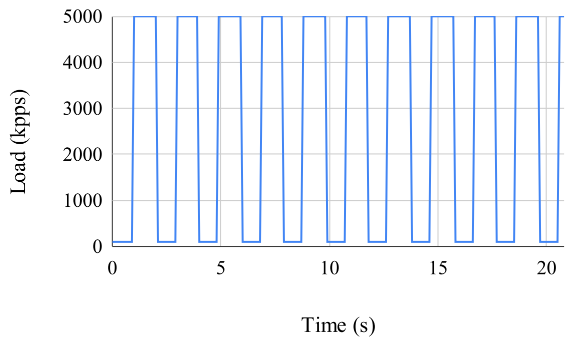


Figure 11: 99.9% Latency (left) and Load (right) over Time for Implementations

As can be observed from the above figures, we do observe an increase in latency as the load changes from 100,000 packets per second to 5 million packets per second at each 1 second interval. We can see from the three graphs provided of the median, 99% tail latency, and 99.9% tail latency, that there is a fairly consistent difference between the three implementations, with the 99.9% tail latency revealing the greatest differences at the points of spike. Hermes Indirect performs the best with this load with 99.9% latency spiking to just 54 μs, followed by Hermes Semi-Direct, and finally by Hermes Priority.

It is likely that Hermes Semi-Direct performed worse due to the short nature of packet service times being 1 μs, leading to an increase in the overall time for SoftIRQs to reach uthreads as the IOKernel is unable to process messages from the cores as quickly (this is proven in Section 4.4.2). In Hermes Priority, the performance is likely worse due to the time for messages from the IOKernel about uthread parkings to reach the uthreads, exacerbated due to the higher latency in packets reaching uthreads, as described in Section 4.4.3.

4.3 How the IOKernel Spends its Time

The IOKernel's many purposes and significance in enabling fast communication between the runtimes and the NIC makes its distribution of work and how it spends its time an important benchmark to analyze. It has a four main purposes: forwarding RX and TX notifications, detecting congestion, adding new cores, and busy-spinning to check for newly idle cores. For this test, we ran 100 uthreads on 14 cores, using an echo server with 0 service time and compared how long the IOKernel spends on the congestion detection and core allocation components. A batch application was also run to keep CPU utilization at 100%. We compare Hermes Indirect with Hermes Priority, as these two have different congestion detection implementations.

	Hermes Indirect	Hermes Priority
Congestion Detection	25.60%	2.87%
Core Reallocation	0.32%	1.40%

Table 1: Percentage of IOKernel's Time spent on Different Tasks

We calculate the percentages in Table 1 by varying the load on the echo server and measuring what is the maximum amount of time spent on each component over all the loads. As can be seen, Hermes Indirect spends significantly more time checking for congestion than Hermes Priority, which comes from the fact that it must check all of the kthreads of all runtimes at every 5 microsecond interval for congestion, and this incurs cache misses as it is checking the head and tail pointers of the queues. On measuring this further, we observe that checking each kthread in Hermes Indirect for congestion, on average, takes up to $17ns$ for the echo workload. Thus, with a $5\mu s$ core reallocation interval, the IOKernel can support up to 294 kthreads across all applications running concurrently in Hermes Indirect. In Hermes Priority, up to 14.9 ns is spent on average checking the priority queue on each iteration of the IOkernel. It is likely that the core reallocation time for Hermes Priority is higher as adding a core would result in the re-insertion of the application into the priority queue, with a lower priority than before.

4.4 Microbenchmarks

We will now evaluate the individual components of Hermes using microbenchmarks.

4.4.1 Uthread Wake-up Time

The number of microseconds for a runtime's uthread to be awakened from being idle was estimated for the implementations. There are a few different circumstances during which a uthread would need to be awoken, and these include:

1. when a kthread is running for the uthread's runtime,
2. when no kthreads for the uthread's runtime are running but idle cores exist,
3. and when no kthreads for the uthread's runtime are running and NO idle cores exist, and an active kthread must be preempted.

Each of these cases can lead to different costs for waking up a uthread and the eventual packet RTT across the different versions.

In Hermes Indirect, when the IOKernel receives a notification, it sends it to an active kthread in the runtime. Following reception of this notification, the kthread then adds it to the uthread's notification buffer, checks if the uthread is waiting, and if it is, then adds the uthread to its own runqueue. Once the uthread gets a chance to run, it will process the notification and the packets, sending the TX notification to the IOKernel. The IOKernel, on reception of this message, will forward it to the NIC in its own TX notification buffer. This encapsulates the low-load RTT of packet for Hermes Indirect.

In Hermes Semi-Direct, when the IOKernel receives a notification, it first adds the notification to the uthread's notification buffer directly, and knowing that the uthread is waiting, also sends a message to any active kthread in the runtime that the uthread must be re-added to the queue. Following reception of this message, the kthread adds the uthread back to its queue, and from there a similar process follows as in Hermes Indirect. Thus, the uthread wake-up time of Hermes Indirect and Semi-Direct to be roughly equivalent.

In Hermes Priority, the path of a packet that encapsulates its RTT is similar to Hermes Semi-Direct: however, there is one difference. When the IOKernel sends the uthread to the kthread to add back to the runqueue, it registers in its priority queue that there's a new pending uthread waiting to be run for the given runtime. If there are no other uthreads on the runtime's queue, then the runtime would need to be removed and re-inserted into the priority queue. Otherwise, the uthread would simply be added to the tail of the runtime's queue. This additional cost is reflected in Figure 12.

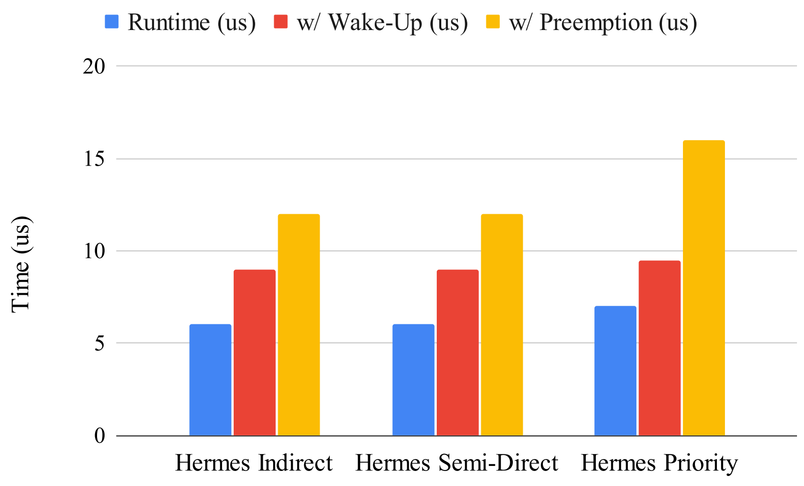


Figure 12: Uthread Wake-Up Times

4.4.2 IOKernel Communication

The time it takes to communicate a message to the IOKernel is also an important benchmark to consider. The IOKernel is a single core handling the incoming RX notifications, outgoing TX notifications, and adjusting core allocations for congestion. The ability of the IOKernel to react quickly to incoming messages from the kthreads within the shared memory queues is essential.

This benchmark became glaringly important when considering the approach of tracking when uthreads are yielding to enable a priority queue of applications to grant cores to, and in order for the IOKernel to notify the kthreads of uthreads that must be re-added to the runqueues. If there is an excess of uthreads that are continually yielding, this could lead to a decrease in performance as the IOKernel would be unable to keep up with the load.

We explored this benchmark for our implementations for an increasing number of cores and uthreads. The echo runtime was used for this (in which each packet has 0 service time), under a load of 10 Gbps. For an increasing number of uthreads, we ran uthreads in range from 2 to 100 on 2 cores, and measured the time it took for the runtime to write a message to the IOKernel and for the IOKernel to read this message.

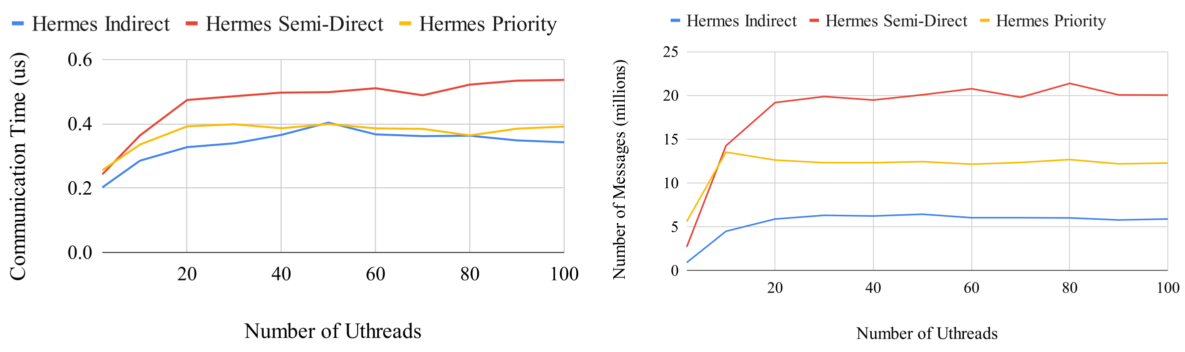


Figure 13: IOKernel Communication as Number of Uthreads Increase

These results are expected: communication time should experience an uptick as the number of messages sent to the IOKernel increase. Following reaching about 20 uthreads, the batch size per notification will approach the same size, resulting in roughly the same number of messages sent and thus resulting in the IOKernel communication time stabilizing. Moreover, in Hermes Semi-Direct, communication time will be higher as the number of messages sent is higher due to uthread parks being sent to inform the IOKernel of a uthread waiting. In Hermes Priority, the communication time is between Indirect and Semi-Direct: this is because it still sends the uthread parks to the IOKernel like Semi-Direct–however, it waits for a response from the IOKernel following each uthread park message. This results in a lower communication time than Semi-Direct, as in order for a uthread to run in the first place on a core, all previous messages from that core must have been acknowledged by the IOKernel.

For an increasing number of cores, we ran 100 uthreads with 100 connections on the core range of 1 to 14. However, it was observed that as the number of cores increased beyond 4, there was a sharp increase in the time it took for the runtime to communicate with the IOKernel. The total number of messages sent was also recorded, and is included in Figure 14. This increase is likely due to a combination of more cores for the IOKernel to iterate through, as well as the higher number of messages sent to the IOKernel resulting in saturated queues. Additionally, in Hermes Priority, the IOKernel communication time was observed to be linear: which reflects how the implementation relies on the IOKernel responding to uthread parks and thus avoiding a build-up of messages to the IOKernel on the queues.

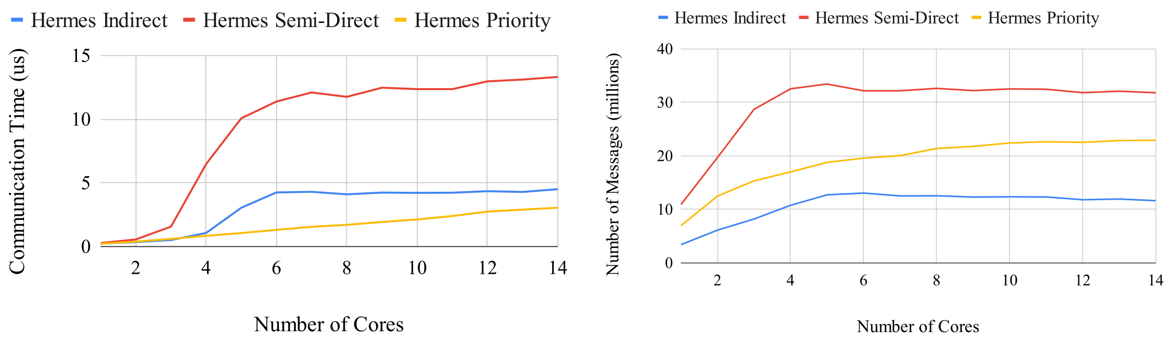


Figure 14: IOKernel Communication as Number of Cores Increase

4.4.3 Time to Reach Uthread

The time it takes for an RX notification or TX completion to reach a uthread was also measured across the implementations. The primary area of concern for this benchmark was the difference between Hermes Indirect and Hermes Semi-Direct: this is because they have two different methods of delivering RX and TX messages to uthreads. Figure 15 hows how these times differ for different numbers of cores. The same parameters as the IOKernel communication microbenchmark were used, with 100 uthreads and a 10 Gbps load.

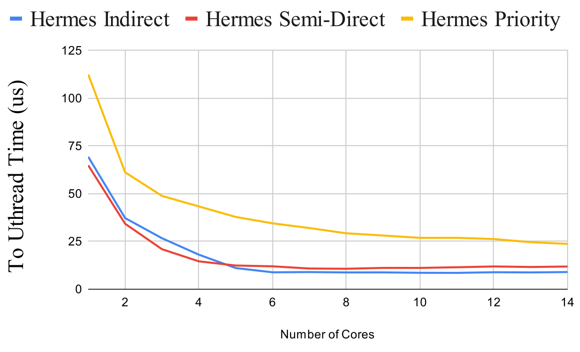


Figure 15: Time for Notification to Reach Uthread from IOKernel

It was expected that the to-uthread time for Hermes Semi-Direct would be less than Hermes Indirect, as Semi-Direct cuts out the middleman when the IOKernel directly adds notifications and completions to the uthreads' notification buffers. However, this was not consistently the case. When the number of cores was low (4 or under), Hermes Semi-Direct did perform better than Hermes Indirect, however, upon reaching 5 cores, Hermes Indirect consistently performed better.

This is due to the overhead of IOKernel communication increasing drastically with Hermes Semi-Direct as the number of cores increase, as seen in Figure 14. In Hermes Semi-Direct, in order for the IOKernel to know to send a uthread to a particular kthread to run, it must know that the uthread is waiting for a notification. With the queueing time for the messages to the IOKernel increasing as the number of cores increase, these messages from uthreads will not be read quickly, resulting in an added latency of $\tilde{3}$ μs. Moreover, it was observed that the waiting time per uthread and the time for a kthread to process a softIRQ were lower in Hermes Semi-Direct, as seen in Figure 16. This further indicates that the delay in the IOKernel reading messages from the uthreads contributes to this latency.

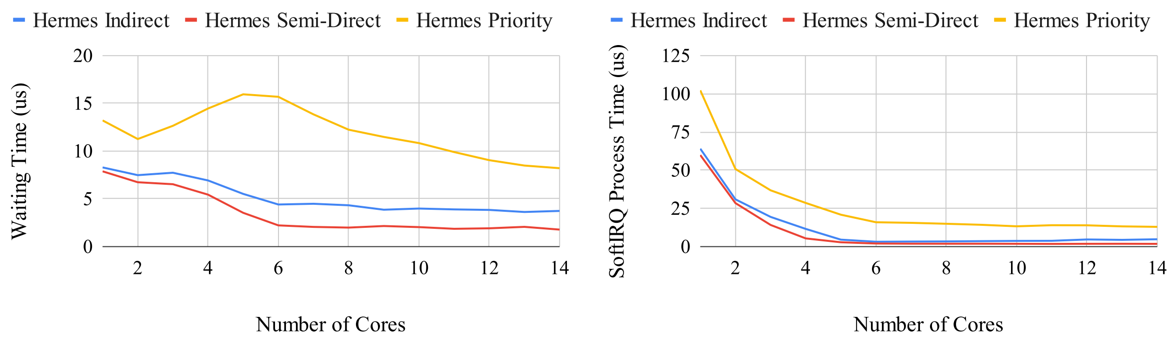


Figure 16: Components of Time to Uthreads: Waiting Time (left) and SoftIRQ Process Time (right)

In Hermes Priority, the time for a notification to reach the uthread is higher than the other implementations. This difference is likely a result of the speed of uthread parks being significantly slower in Hermes Priority, with the IOKernel needing to respond to the sent message each time. The waiting time for Hermes Priority becomes higher when about 5 cores are used, and then decreases. This is likely because the uthread parking time increases as the number of cores increases (as seen later in Figure 17), and the number of cores at 5 is too low to accommodate that speed of uthread parks.

Investigating this further, we have Figure 17 which compares the average time between each softIRQ check (that is, checking if a new softIRQ has arrived) in Hermes Priority to Hermes Semi-Direct, as well as how long it takes for a uthread to park (this includes sending a message to the IOKernel). As can be seen, the time between softIRQ checks for Hermes Priority is significantly higher than Hermes Semi-Direct. While the time between softIRQ checks is lower than the watchdog limit of $50\mu s$, the fact that the time is higher than Hermes Semi-Direct implies that the runqueue for the kthreads does not become empty as quickly in Hermes Priority (as a softIRQ check occurs when the runqueue is empty): likely as each uthread runs longer due to the longer park time.

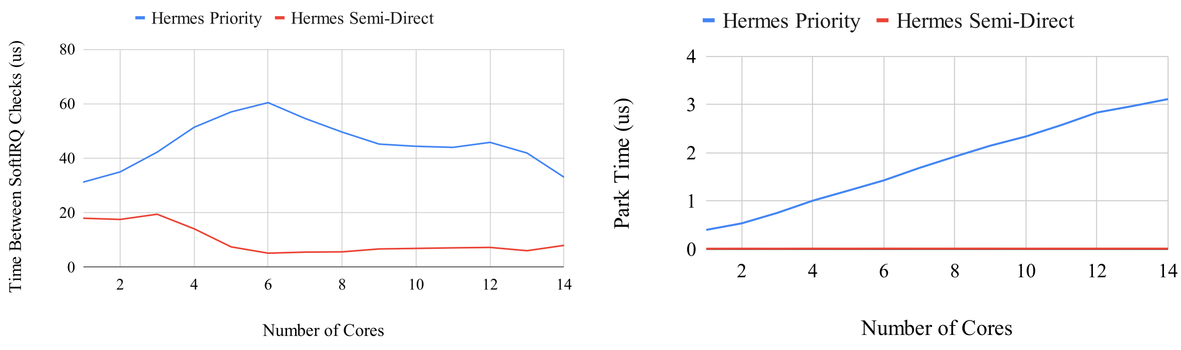


Figure 17: Components of SoftIRQ Process Time: Time between SoftIRQ Checks (us) and Uthread Parking Time (us)

4.4.4 Throughput Scalability

The ability of Hermes to scale throughput as the number of cores increase can be an important metric to take into consideration. To evaluate this, the number of cores available for use by a simple echo server with a service time of $0\mu s$ was increased and the maximum zero-loss throughput was measured: that is, the maximum throughput possible while not dropping any packets. The ability of Ensō to scale well with the number of cores is excellent, and thus throughput scalability performed well, saturating the 100 Gbps line at just 2 cores across all versions.

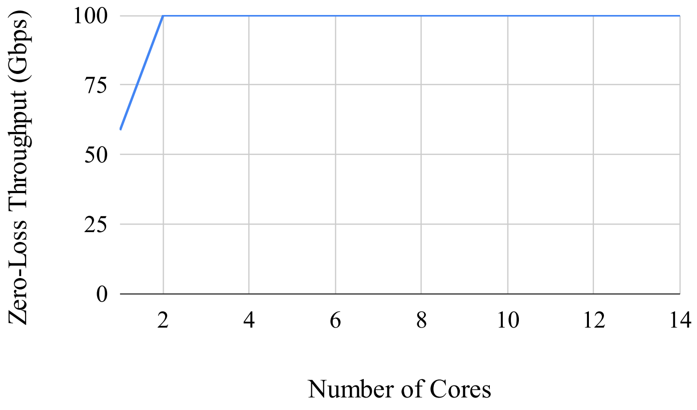


Figure 18: Throughput Scalability: Zero-Loss Throughput vs. Number of Cores

4.4.5 Runtime Efficiency

We can approximate the CPU efficiency of the various implementations by considering the time spent performing useful work compared to the total amount of time during which runtime CPU cores are in use. The total amount of useful time was measured by counting the number of cycles during which uthreads are being run, and the application itself was scheduled to be run for a certain duration of time. The application was a simple echo server, with an constant service-time distribution of 10 μs.

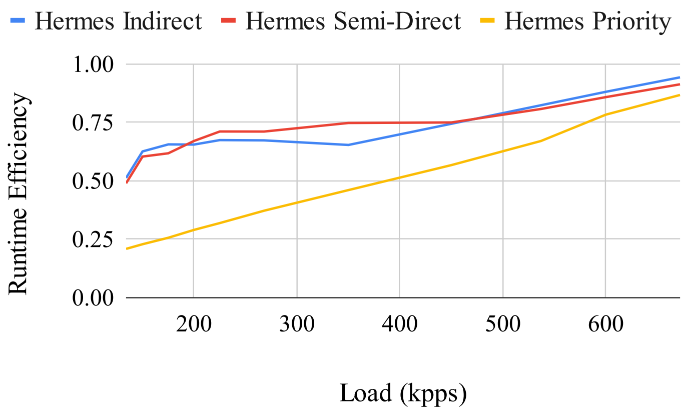


Figure 19: Runtime Efficiency

At lower loads, it can be seen that the runtime efficiency is lower across the board. We can also observe that Hermes Indirect and Semi-Direct face similar runtime efficiencies–it was expected that Semi-Direct would face lower runtime efficiency due to the need to send a message to the IOKernel on each uthread park. At higher loads, this seems to be the case–however, it is a fairly insignificant difference. Hermes Priority has a linear increase in runtime efficiency, and is lower than the other two contenders: which can be attributed, again, to its need to send a message on uthread switching and to wait for a response, hogging up runtime cycles.

5 Related Work

In this section, we compare the various Hermes implementations to prior work. IX [1] bypasses the kernel to achieve low latency as well, and processes batches of packets and thus achieves high throughput for workloads with uniform service times. However, it does not implement work stealing or reallocate cores at 5 μs intervals. Hermes is similarly capable of achieving high throughput due to the nature of Ensō, but can make these reallocation decisions at a smaller granularity.

ZygOS [10] implements distributed first-come-first serve with work stealing, and no core is dedicated to a centralized entity such as the IOKernel. It statically provisions cores to runtimes for peak load–thus resulting in cycles wasted during busy-waiting. Shenango is similar to ZygOS, in that both perform work stealing for load balancing among runtimes. However, Shenango improves on ZygOS by improving CPU efficiency with a microsecond-scale core allocator. Similarly, Hermes also ensures that CPU efficiency is kept higher than ZygOS's with the congestion detection algorithm.

Moreover, Caladan [4] is capable of achieving low latencies for a variety of service times, including constant, exponential, bimodal, etc. While Hermes is currently not achieving similar performance, extending it to use locking behaviors when accessing Ensō devices could serve to yield comparable performance to Shenango [9] and perhaps Caladan, while reaping the benefits of using Ensō's non-packetized abstraction. Additionally, we expect Hermes to achieve better runtime efficiency than Caladan due to the lack of updating flow steering rules.

Ringleader [7] attempts to solve the problem of optimizing CPU efficiency and minimizing tail latency with a hardware-based approach, offloading orchestration entirely on the NIC and supporting NIC-assisted CPU scheduling. It attempts to address the limits faced by software orchestrators such as Caladan and Shinjuku with **scalability**, with them requiring additional IOKernels to handle more applications and cores. Hermes is orthogonal to this work as it too focuses on a software-based approach. It would be interesting to consider the scalability capabilities of Hermes vs. Ringleader, with Ensō being introduced.

6 Future Work

Shinjuku [5] is another kernel-bypass system that uses a single-address space operating system to make preemption practical at the microsecond scale, allowing centralized scheduling policies to be implemented every 5 μs. Additionally, it has the capability of stopping long-running requests within an application and placing them back on a centralized scheduling queue could be interesting to incorporate in the future: currently, Hermes is only capable of preempting applications as a whole, and the applications themselves manage scheduling within their runtimes with the cooperative scheduling policy. Modifying the policy to be more preemptive could be another route to improving tail latency.

Additionally, Junction [3] is the continuation of Caladan, that aims to support thousands of instances of runtimes that use kernel bypass on a set of cores without compromising on CPU efficiency–essentially trying to make Caladan more suitable for the cloud, which requires high security and density of applications. It extends Caladan by adding Shinjuku's capability of interrupting runtime instances when they've exceeded their timeslice quantum. It incorporates new techniques such as a timer wheel and NIC event queue for monitoring instances in a scalable way, reducing the storage necessities of Caladan and allowing it to support far more applications. Using these techniques to reduce our storage needs would be an interesting avenue to becoming more scalable as well.

7 Conclusion

In this thesis, we have evaluated several methods of integrating Ensō into a software-based core reallocation system that maximizes CPU efficiency and minimizes latency. While some systems performed better than others in certain metrics, it was clear that the overall winner was Hermes Indirect: the system that iterates over all queues to detect packet and thread queueing delay, and that has kthreads distribute notifications and completions among the uthreads. This is due to its ability to handle bursts in load well, have good runtime efficiency, and handle service-time variability the best. Moreover, it is apparent given the evaluation of these various methods that eliminating the IOKernel as an intermediary when transmitting notifications would be a strong route to consider, as that would enable kthreads to forward notifications rather than packets to the notification buffers of uthreads in a potentially more efficient manner than Caladan's.

References

[1] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[2] DPDK. Data Plane Development Kit, 2023. `https://dpdk.org`.

[3] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Inigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. Making kernel bypass practical for the cloud with junction. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 55–73, Santa Clara, CA, April 2024. USENIX Association.

[4] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[5] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for µsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[6] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[7] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. RingLeader: Efficiently offloading Intra-Server orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1293–1308, Boston, MA, April 2023. USENIX Association.

[8] Amy Ousterhout. Achieving high cpu efficiency and low tail latency in datacenter, 2019.

[9] Amy Ousterhout, Joshnua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium for Netowkred Systems Design and Implementation*, NSDI '19, pages 361–377. USENIX Association, February 2019.

[10] George Prekas, Marios Kogias, and Edouard Bugnion. ZygOS: Achieving low tail latency for microsecond-scale networked tasks, 2017.

[11] Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Ensō: A streaming interface for NIC-application communication. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '23, pages 1005–1025. USENIX Association, July 2023.

[12] Xilinx. OpenOnload, 2023. `https://openonload.org`.