Accio: Rethinking OS-Architecture Co-Design for Next-Gen I/O

Amirmohammad Nazari
Yale University, New Haven, CT, USA

Rajit Manohar
Yale University, New Haven, CT, USA

Robert Soulé
Yale University, New Haven, CT, USA

Abstract

In this paper, we propose a novel software/hardware design to improve I/O performance while maintaining existing POSIX-standard APIs. Our approach stands in contrast to existing kernel-bypass strategies that improve performance at the expense of abandoning familiar programming abstractions. Our key insight is that navigating the performance-functionality trade-off requires changes to the processor; it cannot be done without support of the CPU micro-architecture. Our design, called Accio, includes: dedicated hardware for interrupt management, a hardware assist for thread scheduling, tables in hardware that manage I/O state, and modifications to the operating system to support the new hardware. Our evaluation demonstrates that Accio saturates the bus bandwidth, reduces CPU usage by up to 66% compared to state-of-the-art kernel-bypass systems, and reduces latency to 1/12th of that of the Linux kernel, matching that of kernel-bypass systems.

2012 ACM Subject Classification Hardware → Networking hardware; Computer systems organization → Processors and memory architectures; Software and its engineering → Operating systems

Keywords and phrases Networks, Operating Systems, I/O Optimization

Digital Object Identifier 10.4230/OASIcs.NINeS.2026.21

Funding This work was partially supported by the National Science Foundation under awards #2212235, #2420367, #2415262, and #2449506.

Acknowledgements We thank the anonymous NINeS reviewers and our shepherd, Justine Sherry, for their valuable feedback.

1 Introduction

Raw I/O bandwidth for both networking and storage has increased significantly over the past few decades [39, 1, 18, 10], while single-threaded CPU performance has stagnated. As a result, the relative percentage of time spent in I/O vs. CPU has changed. Conventional operating system abstractions for I/O – once viewed as low-overhead – are now seen as barriers to meeting performance goals [17, 40, 42, 41, 3, 27, 33].

To be concrete, let's consider what happens when an application wants to send a packet over the network using the traditional kernel-based network stack. Using the sockets API, the application relies on the OS to: (i) copy data in and out of the kernel via blocking systems calls and (ii) execute network protocol logic. In the past, the OS would have also copied data from the server's memory onto the NIC, but today, it is more common that the OS can write to NIC memory directly (a "zero-copy" approach). Prior work has focused on two natural areas for performance improvement, addressing the overheads introduced in (i) and (ii) above respectively.

© Amirmohammad Nazari, Rajit Manohar, and Robert Soulé;
licensed under Creative Commons License CC-BY 4.0

The first class of solutions, focusing on (ii), the network protocol logic, leverages hardware accelerators [2, 29] and offload engines [43]. Transport protocol offload has long been a contentious topic [32], as advocates and critics debate whether or not the potential performance benefits outweigh the costs of the added hardware complexity. Regardless of which side one takes, it is clear that offload engines improve the performance of header processing, but do not address inefficiencies associated with data transfer, interrupt handling, or the system call API.

A second class of solutions focuses on (i) by making the operating system more efficient while preserving the sockets API. A classic example of this approach is the aforementioned zero-copy mechanism that leverages shared access to a memory pool between the OS and NIC [48]. However, modern kernel developers have spent decades optimizing the software on the I/O path [31], so we are unlikely to find significant further improvements.

Hence, a third class of solutions has become increasingly popular: bypass the OS completely [50, 30, 42, 38, 14, 7, 49]. With kernel-bypass, the application is given direct access to the I/O device, which allows the application to transfer data without performing additional copies or system calls, resulting in significant performance improvement. On the other hand, this kernel-bypass puts a significant burden onto application developers. RDMA, for example, requires the use of lossless Layer 2 protocols (e.g., InfiniBand, RoCE), custom programming interfaces (e.g., verbs API), and out-of-band coordination (e.g., for negotiating queue pair IDs). DPDK uses polling to avoid interrupt overhead at the expense of higher CPU utilization, and forces developers to either work with raw packet data or provide userspace libraries to implement the network stack. Other important functionality, such as ARP caching and maintaining routing tables, is elided and responsibility for multiplexing access to the NIC is shifted to the application-level software and is typically done using shared-memory segments in userspace. This creates significant burden on the user, and eliminates inter-application isolation guarantees – the traditional function of the OS.

Clearly, it is important to provide both high performance as well as user-friendly networking abstractions for software developers. Hence, an ideal solution to address the I/O bottleneck in a modern computer system would offer kernel-bypass like access to the network hardware while preserving the OS abstractions that provide the key properties such as isolation, hardware multiplexing, etc. that provides reliable operation in the multi-programmed context. Our key observation is that modifying both the CPU architecture along with the NIC allows us to get the best of both worlds.

As both inspiration and analogy, we looked at hardware and operating system support for virtualizing the memory system [13, 23]. Virtual memory provides a level of indirection between the address space of user threads and the physical memory, while providing almost the same performance as direct physical memory access in the common case. The OS is responsible for managing the core data structures (i.e., page tables) for virtual memory. To ensure high-performance, the hardware uses a translation look-aside buffer (TLB) as a cache to avoid OS intervention in the common case for memory accesses. The OS ensures the TLB is in a valid state, and the hardware uses the exception mechanism to call the OS when the TLB information is insufficient to complete a memory request. Virtual memory provides a classic example of co-design between the operating system and hardware to provide a safe and performant critical service. This naturally leads to the question: *can we do the same for I/O?*

In this paper, we propose a novel software/hardware design to improve I/O performance that preserves existing POSIX-standard APIs. Our contention is that navigating the performance-functionality trade-off requires small yet important changes to the processor;

it cannot be done without support from the CPU micro-architecture. CPU support is necessary because achieving high performance requires hardware data structures that are accessible from both user and kernel space, but the access in user space is limited to enforce OS protection mechanisms. We refer to this approach as Accio. Accio includes dedicated hardware for interrupt management, a hardware assist for thread scheduling, and dedicated tables in hardware that manage I/O state. Additionally, we implement changes in the operating system to support this new hardware and system calls. This design allows the CPU to minimize the overhead of interrupt processing by only using interrupts when a blocked high-priority thread can be woken up by I/O operations performed by the NIC on its behalf. This way we can mitigate the performance degradation that occurs due to slow interrupt handling such as unnecessary high latency.

We have implemented a prototype of Accio on an FPGA-based RISC-V system with a 100 Gbps NIC. The prototype consists of a custom NIC with a fast-path UDP offload engine, a new software-visible hardware structure called the I/O Connection Table (ICT), and direct userspace access to DMA ring buffers. The FPGA runs a modified version of Linux.

We have evaluated our prototype by running UDP micro-benchmarks and a ping-pong application and found that Accio reduces CPU usage by up to 66% while lowering packet processing latency by 31% compared to DPDK, while allowing applications to continue using traditional POSIX networking APIs.

Overall, we make the following contributions:

- We present new architectural support for fast communication, modifying the way the CPU, NIC, and OS cooperate to provide hardware and software support for networking.
- We show that our new design provides performance comparable to existing kernel-bypass schemes like DPDK and RDMA.
- We show that our design provides this performance while preserving the traditional role of the OS in providing protection and hardware multiplexing, and mostly preserving existing networking APIs for user code.

The remainder of the paper is organized as follows. We begin with an overview of current I/O systems and their inefficiencies in §2. Next, we detail the packet flow in Accio in §3, transitioning to the specific designs of the CPU, NIC, and software components in §4, §5, and §6. We then present our FPGA-based implementation in §7, and conclude with an evaluation of the prototype in §8.

2 Background and Motivation

Before presenting the Accio design, we first briefly discuss the state-of-the-art and identify opportunities for I/O performance improvements.

2.1 Linux Networking Stack

We first discuss what happens when two endpoints want to communicate with one another using the Linux networking stack. To be concrete, we focus on the UDP protocol.

From the programmer's perspective, using a standard POSIX API, both the client and server code would use the `socket` data structure to set up IP addresses and ports, and then call `send`/`recv`, passing a socket descriptor and a data buffer as parameters, to perform the communication.

From the operating system perspective, the socket setup associates the IP and port with the user application. The operating system is also responsible for maintaining the mapping between IP addresses and physical hardware routing addresses (e.g., Ethernet MAC

addresses, ARP tables, etc.). On a `send/recv` call, the socket information is converted into the appropriate NIC-device specific I/O mapped memory operations (e.g., providing packet headers, setting up DMA, etc.). Since all the network operations flow through the operating system, the OS handles the multiplexing of NIC resources and provides a convenient interface to the programmer.

From a hardware perspective, when the OS initiates a send via the NIC device driver, it performs a DMA copy of the packet from the user's data buffer to a shared-memory buffer that can be directly accessed by the NIC. The NIC hardware accesses this data and sends it physically over the Ethernet port. On the receiver side, when there is inbound data, the NIC hardware writes the data to a pre-allocated buffer that is shared with the OS and notifies the CPU via an interrupt that data is available. The interrupt triggers the OS execution, which in turn, notifies the application that data can be read.

This is a very simplified, high-level view. There are many references, backed by decades of research, which provide more detail about the latest and most-optimized version of this communication path [48, 31, 4].

2.2 Inefficiencies in the POSIX Interface

The `send/recv` paths described above include several inefficiencies. First, for every data transfer, the userspace applications must transfer control to the OS via a system call, which results in overheads associated with the context switch. Second, the data to be sent/received are copied twice, once between userspace and the kernel, and once between the kernel and the NIC. Finally, interrupts from the NIC to the CPU are used to notify the operating system of state changes in the NIC hardware (e.g., data arrival), which once again, requires a context switch from a running process to the operating system, incurring the overhead of saving context state.

To quantify some of these costs, we instrumented a Linux kernel to detail the latency breakdown for UDP packet processing from the moment the packet is received at the NIC until it is received by the application. The results are shown in Table 1. We see that nearly 60% of the time is spent on userspace delivery of the data, while 15% of the time is spent on header processing. It takes 4.9μs before a packet that has been received by the NIC starts header processing in the kernel – nearly 25% of the time spent on the whole RX packet processing.

2.3 Reducing Per-Packet Overhead

Over decades of work, there have been many proposals from both academia and industry to reduce these well-known overheads. For example, as mentioned earlier, the two copies described above can be reduced to one via a "zero-copy" interface [48]. Given space limitations, we highlight a few additional representative solutions, focusing on those that are most relevant to this paper.

Transport Protocol Offload

Network protocol offload has been the subject of a robust debate in the networking community [32]. However, the combination of the end of Moore's law, the increase in network speeds, and the widespread adoption of RDMA has recently led to revived support for transport protocol offload. There are several contemporary proposals for offloading network stacks and basic network functions [2, 11, 28].

Table 1 Latency breakdown of UDP RX packet processing in the Linux kernel, from packet entry at the NIC to reception by the userspace application. The table shows the time spent in each phase, including NIC-to-kernel transfer, header processing in the kernel, and delivery from kernel space to userspace application after header processing is complete. Each entry shows the total time along with the respective percentage of total latency. Experiments were conducted on two servers directly connected to each other, each with 2x Intel Xeon Gold 6240 CPU @ 2.60GHz with 72 cores using an Intel E810-C 100GbE NIC.

Component	Time (μs)	Percentage (%)
NIC to Kernel	4.90	25.00%
Kernel Header Processing	2.95	15.06%
Kernel Space to Userspace (Excluding Header Processing)	11.75	59.94%
Total Time	**19.60**	**100.00%**

Interrupt Processing

Modern network drivers utilize a more efficient interrupt processing mechanism called New API (NAPI) [35] that batches the NIC interrupts and schedules a soft interrupt to process multiple receive packets at the same time. This reduces the time that the CPU stays in interrupt processing mode, leading to fewer CPU context switches and better cache hit ratios. However, it increases the latency of packet processing, since a packet stays in memory until NAPI processes it. Other techniques used by modern network drivers to reduce overheads include Generic Receive Offload (GRO), Large Receive Offload (LRO), checksum offloads, Generic Segmentation Offload (GSO), and TCP/UDP Segmentation Offload (TSO/USO) [37].

Kernel-Bypass

To remove the overheads associated with the OS, including scheduling, multiplexing, context switching, etc., developers have increasingly turned to kernel-bypass strategies. These approaches skip the kernel entirely and expose the "bare metal" hardware of the NIC to userspace applications. Two prominent kernel-bypass technologies are the Data Plane Development Kit (DPDK) and Remote Direct Memory Access (RDMA).

DPDK gives an application direct access to a network card, allowing it to transfer data directly without memory copies or system calls. However, the application must handle all the networking protocol logic and any multiplexing of the hardware resources. Moreover, to achieve the best performance, a DPDK application must continuously poll the NIC, resulting in high CPU utilization [8].

RDMA focuses on removing the overhead of data copies by allowing the NIC to directly transfer data to and from pinned memory pages without the involvement of the operating system. RDMA uses a custom protocol stack relying on lossless network delivery, which can exhibit problems at scale (e.g., deadlock, livelock, and PFC storms [16]). Like DPDK, RDMA also forces applications to use a non-POSIX API (i.e., "verbs") that is more complex than UNIX sockets. Finally, RDMA's queue-pair abstraction limits the number of user applications can take advantage of RDMA's benefits.

To evaluate kernel-bypass strategies, we implemented a basic ping/pong application using a pair of servers with Intel X710 NICs. Figure 1 shows the effect of removing the kernel from the I/O path using DPDK. The baseline is the OS networking stack used with sockets. As one

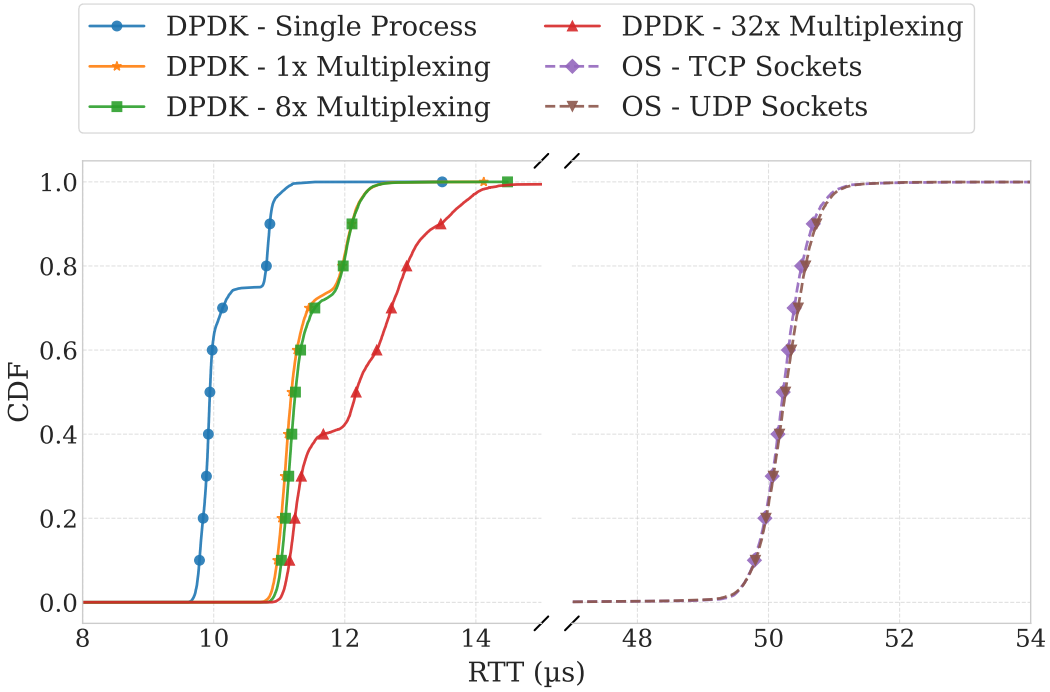


Figure 1 A simple ping/pong application running on two FreeBSD servers with Intel X710 NICs. We can see the effect of adding more features to a DPDK program. The baseline is the OS networking stack used with sockets.

would expect, we can see that DPDK achieves significantly lower latency ($\approx 1/6^{th}$) compared to the kernel networking stack. However, this performance improvement comes at a cost: the application has exclusive access to the NIC hardware and all of the functionality typically handled by the OS (e.g., header processing, name resolution, etc.) must be performed by the application itself in userspace.

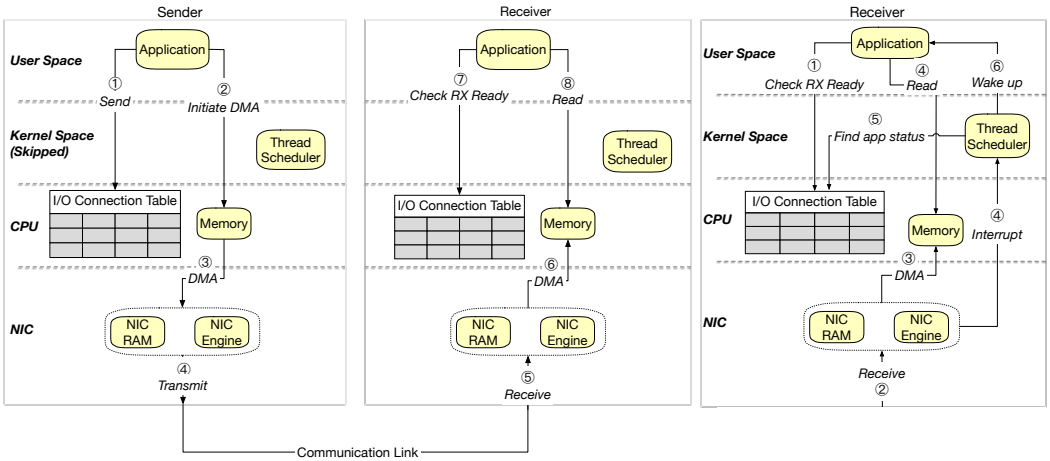
To evaluate the cost of implementing some of this functionality, we followed the example provided by the DPDK documentation to multiplex access to the NIC [34]. The suggested approach is to allocate a hugepage shared memory region, and implement queues that multiple processes can access for data transfer. As can be seen in Figure 1, as we add more processes, the measured latency of the application becomes progressively worse. We would expect that adding additional functionality, such as header processing, would further offset the benefits of DPDK.

We've limited our discussion in this Section to DPDK. However, prior work has shown that a parallel set of issues arise with RDMA [26, 20, 25]. For example, contention on NIC caches and processing units result in performance anomalies.

3 The Life of a Packet in Accio

To explain the Accio design, we now walk through the life of a packet from a sender to receiver in the "best case scenario," illustrating the new division of responsibility among the application, OS, CPU hardware, and NIC hardware.

As a starting point, we look to RDMA (or DPDK) as the ideal path for performance we might achieve, in which the application directly accesses the network hardware without overhead imposed by the OS or CPU. Hence, Accio's "fast path" looks similar to kernel-bypass mechanisms.



(a) Accio TX and RX in "best case" path. **(b)** RX when a thread is blocked.

Figure 2 Accio TX and RX paths. The numbers indicate the sequence in which different actions occur during communication. The I/O connection table is the additional hardware state maintained by the CPU used for I/O acceleration.

Our approach differs from kernel-bypass in that we additionally support NIC multiplexing, protection, and a POSIX network API. In our design, rather than bypass the kernel, we enlist the kernel to implement the control path. The operating system can determine if the conditions are met that allow a userspace application to be able to support direct data transfer to/from the NIC without additional intervention. Essentially, the kernel continues to implement the control plane, while the data plane can bypass the kernel and leverage hardware acceleration. Returning to our virtual memory analogy, this is similar to the division of responsibility between the OS and hardware for address translation and memory protection: the OS sets up page-table entries in the TLB, which in turn enables hardware accelerated address translation without any OS intervention in the common case.

TX Path

We start on the transmit (TX) path, as illustrated in Figure 2a. We assume for now that all the necessary hardware and software structures have been appropriately initialized by the operating system during the usual POSIX connection establishment calls. This functionality can be hidden from the user by augmenting the traditional `open` API.

The sending application writes data to a userspace buffer. The memory backing this buffer must have been reserved and pinned in the usual way. To notify the NIC that new data is available, an I/O-mapped control operation (e.g., an uncached store word instruction) is used. Note that we need hardware state (detailed in §4) so that an application can determine that this optimized path is enabled without communicating with the OS.

Like prior work (e.g., Tonic [2], RingLeader [29], etc.), we assume the NIC has the ability to completely offload all protocol processing steps. This allows the NIC to directly transfer data from the pinned memory, apply the necessary packet headers, and transmit the correctly framed packet directly over the wire. Note that in the entire transmit process, there is no involvement of the operating system, similar to kernel-bypass, but the POSIX abstractions are preserved.

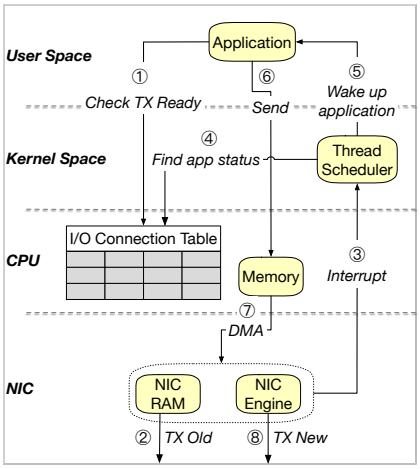


Figure 3 Accio TX slow path. Numbers denote the sequence of operations.

RX Path

The receive (RX) side, illustrated in Figures 2a and 2b, is more complicated. When a packet arrives at the destination NIC, we once again leverage protocol offload on the NIC to perform header processing, and then directly write the data into the appropriate userspace memory. Again, we assume that the memory has been reserved and pinned during connection establishment. If the receiving thread is running and it attempts to read the data after the NIC has transferred it to user-memory (Figure 2a), then the application can simply read the data and continue execution. In order to do so, it needs metadata to know that data is available. We maintain a hardware data structure, called the *I/O Connection Table* (§4) that holds metadata about the state of the connection, the availability of data, thread information, and protection, and which is accessible from userspace. Returning to our virtual memory analogy, the I/O Connection Table is similar to a TLB, but for I/O acceleration.

RX Slow Path

The more complex case occurs if the receiving thread is blocked waiting for data (Figure 2b). In this case, once the data is in memory, Accio is designed to minimize the overhead associated with interrupt processing and thread scheduling. Therefore, we move some of this logic into specially designed hardware components. The I/O Connection Table includes a hardware ready queue that prioritizes accelerated I/O threads, to simplify the OS process scheduler. If an accelerated I/O thread is suspended waiting for data to become available, the hardware ready queue alerts the operating system via a new interrupt type, and the OS simply restores the state of the context and resumes execution. One could imagine doing the context switch in hardware [5], but our initial approach aims to preserve hardware independence from the specifics of operating system to the extent possible.

TX Slow Path

The TX slow path, shown in Figure 3, is similar to the RX slow path. When an accelerated I/O thread tries to send data, but the TX buffer is full, the thread blocks until the NIC delivers a TX completion signal. Then, the hardware ready queue uses a dedicated interrupt to quickly notify the OS to resume the thread.

4 Accio CPU Design

Below, we discuss the hardware changes to enable I/O acceleration in Accio. We first present the discussion in the context of a single-core CPU, and then discuss modifications in the modern multicore context.

I/O Connection Table (ICT)

The key hardware data structure that enables I/O acceleration in Accio is the I/O connection table (ICT). To continue the virtual memory analogy, the ICT serves the same purpose as the TLB for virtual memory – namely, it is a hardware cache for the key OS information, supporting fast I/O operations without requiring OS intervention. Each ICT entry contains a socket identifier, a process/thread identifier (an address space identifier or ASID in MIPS/RISC-V terminology), the communication flow direction (outgoing or incoming), a DMA buffer address and length, read and write pointers into this DMA buffer, and state bits to indicate the status of the connection. The state bits record information that indicates if the connection is in an accelerated state or not, as well as if there is a thread suspended on I/O for this ICT entry either waiting to send data or receive data. The ICT also contains a flag to indicate whether interrupts should be used to wake up suspended threads. ICT entries with the interrupt flag set are viewed as higher priority than ordinary entries. Note that in general, data in the ICT structure could be held either by the CPU or by the NIC, given that Accio co-designs both.

To track if the CPU is running an ordinary or accelerated I/O thread, we add an extra bit to the ASID. This bit determines if a NIC operation should trigger an interrupt and context switch. It is updated by the operating system on a context switch in the usual way along with the other ASID bits, and can be simply viewed as an extra bit in the ASID that does not participate in address translation/protection.

Hardware I/O Ready Queue

The ICT includes information about the ready-to-execute threads that were previously blocked on an accelerated I/O operation. To store this information, we maintain a hardware queue of ready-to-execute threads with a simple round-robin scheduler. Accessing the queue to get the next ready thread is supported via memory-mapped registers that are only accessible in kernel mode. Threads are added to this hardware ready queue by NIC actions (§5) that trigger thread wake-ups.

ISA Modifications

To avoid introducing new instructions or mechanisms for CPU/NIC communication, access to the ICT is handled through memory-mapped operations similar to standard memory-mapped I/O (MMIO). A section of the address space is mapped to the ICT, and entries can be accessed using load and store operations. Loads and stores to an ICT entry succeed in kernel mode, and are protected in user mode via the same mechanism used for TLB memory protection. For example, we use the ASID field in the ICT to check that the currently running process has permission to perform a load/store to the ICT entry, and raise a standard memory protection exception on failure.

Logically, the CPU interrupts that wake up accelerated I/O threads should be given high priority. This can be designed in either of two ways: (i) a new interrupt type, with an associated interrupt vector, or (ii) by modifying the hardware interrupt controller to treat these specific interrupts as higher priority than others.

CPU to NIC Notifications

When a process does a write operation with new data, a notification is sent from the CPU to the NIC in the usual way (i.e. via a MMIO control operation) with the ICT entry identifier. When a process reads data from the DMA buffer making space available, a notification is sent from the CPU to the NIC as well. These notifications are triggered when the ICT entry read/write pointers are updated by the user process. This prevents the NIC from having to scan through the ICT entries, polling for state changes, which does not scale to a large number of cores.

To maximize performance, we are assuming an integrated CPU/NIC architecture but this approach could just as easily be adapted to scenarios such as PCIe communication between the CPU and NIC.

5 Accio NIC Design

We now discuss hardware changes in the NIC. These changes mostly relate to managing the state of the ICT.

TX Path

The CPU notifies the NIC using MMIO when new data is available (§4), initiating the data transfer operation. The ICT entry information in the notification is used to determine which flow and DMA address has new data through memory-mapped reads from the NIC to the CPU, which are routed by the CPU to the ICT. This data is used to construct the network packet (using hardware protocol processing) that is transmitted over the physical network port. Once the packet has been transmitted, the read pointer in the ICT entry is updated (MMIO). In the event that an I/O thread was suspended waiting for additional DMA space (also recorded in the ICT entry), the NIC performs a MMIO operation to add the ICT entry to the hardware ready queue. If this is a high priority thread, an interrupt is delivered to the CPU as long as it is not currently running another higher-priority or equal priority I/O thread, determined via the extra bit in the ASID field (§4).

RX Path

The RX path is analogous. Data is written to the DMA space in the CPU, and the DMA location is obtained from the ICT by the NIC using MMIO. If a thread is suspended, the same wake-up logic as in the TX path is used. The ICT entry is updated to reflect the new data arrival.

Mitigating Interrupt Overhead

The NIC uses the new prioritized I/O interrupt to wake up I/O optimized threads by inserting them into the hardware ready queue. When the interrupt is delivered to the kernel, the thread is available for prioritized scheduling, bypassing the normal OS scheduler.

The Multicore Context

The multicore context is more complicated, because, typically, one NIC is shared across multiple cores. Therefore, each core must maintain its own ICT entry, and the optimized I/O threads will be (by default) pinned to specific cores. (Note that this is a common step to mitigate caching effects.) To allow optimized I/O threads to migrate to other cores, Accio

uses a mechanism for ICT management similar to those used for TLBs. The CPU-to-NIC path does not change beyond the NIC knowing which CPU core (and hence, core-specific ICT) provided notifications. The NIC must also know which flow is associated with which core ID, so as to contact the appropriate core-specific ICT. Alternatively, the NIC could send ICT requests to all cores simultaneously, and only one core would in fact respond – an ICT request to a core that doesn't match would simply be ignored by that core. Finally, OS protection units (processes) and scheduling units (threads) are different in a modern OS, and the appropriate identifiers must be used by the modified hardware structures outlined above.

6 Accio Software Design

To complete the design discussion, we describe the software changes to support I/O acceleration in Accio in more detail.

User-Application API

One of our primary design goals is to leave the traditional, POSIX application-level API unchanged to the extent possible. This means that application developers should be able to use the standard functions, such as `socket()`, `listen()`, `connect()`. `send()`, and `recv()`, etc. for network communication. To support this functionality, we implemented a POSIX-compatible, userspace networking library that avoids operating system calls in the fast path for `send()` and `recv()`. Functions involved in operations such as connection setup, tear-down, etc. continue to use a system-call interface.

The library API deviates from the standard in two minor details. First, when invoking `send()` and `recv()`, users must pass data buffers that have been allocated from a dedicated memory pool in which the backing memory pages have been pinned to prevent the OS from swapping them out. The initialization of this memory is performed during connection setup and hidden from the user code. Second, users must indicate that the socket should be accelerated via a socket `ioctl()`, and therefore, added to the I/O Connection Table.

OS Changes

The operating system needed three modifications to support the Accio design. First, as described above, the implementations of the POSIX networking functions were slightly modified. These changes required corresponding changes to the OS system calls involved (e.g. `ioctl()` calls that enable fast I/O must now perform ICT updates). Second, the software ready queue maintained by the OS is augmented by the ICT-maintained hardware ready queue. The OS prioritizes I/O accelerated threads over other threads. To do so, the operating system process/thread scheduler was modified to accommodate fast path scheduling. Concretely, the operating system is modified to check if there is a ready-to-execute accelerated I/O thread. If there is such a thread, it is scheduled for execution; otherwise, the normal scheduling algorithm is invoked. Finally, when a thread with an active ICT entry terminates, the appropriate ICT structures must be released.

Memory Pool

As mentioned in the application-level API above, userspace applications need to read and write data from a dedicated memory pool. Hence, there must be a mechanism so that the application knows when memory can be reused. In contrast to RDMA, which uses an explicit

completion queue, we implemented the memory pool as a per-connection, circular buffer. This design allows us to suspend on either read or write operations when data is unavailable or the allocated memory is full respectively.

Exception Handling

Accio is intended to accelerate the common case for I/O in which there are no errors. However, the accelerator design itself is not robust to failures. Rather, we assume that in the event of some exceptional case or failure (e.g., a cable is unplugged, etc.), the system will revert to the standard I/O path in which the operating system is responsible for network data processing. Hence, Accio needs a mechanism for both exception handling and for reverting to fast path behavior when it becomes available again.

To provide this fallback, the ICT maintains a control event queue that records link state changes, errors, and other exceptional conditions. Modern NICs already implement mechanisms such as link state detection, and these events are forwarded to the ICT queue. A dedicated interrupt line notifies the software of new events, allowing it to read event, handle failures, and revert to the standard I/O path when necessary by changing the state of the socket to indicate it is no longer in an accelerated state. To support this, we maintain a "fallback" traditional socket structure even when the socket is marked as accelerated to permit seamless switching between the accelerated and non-accelerated state.

ICT Resource Management

The ICT, being a hardware structure, contains a finite number of slots. If the number of connections to accelerate exceeds the number of slots, we must time-multiplex ICT slots in a manner analogous to the way TLB entries are multiplexed to support a large page table. Two standard approaches used for TLB management include a hardware page table walker, and a fast exception path with a dedicated exception vector for quick TLB updates. Either of these approaches can be adapted for sharing ICT slots if there are too many accelerated I/O sockets in the system.

7 Implementation

We have implemented an FPGA prototype of Accio on an unmodified RISC-V Rocket Core using a Xilinx VCU118 board with a 100 Gbps QSFP+ interface. To enable the I/O ports of the VCU118, we used the GTY transceiver and the CMAC wrapper implementation from Corundum [12], and connected it to a modified version of a standard NIC, IceNIC [21]. Since IceNIC is designed only for simulation scenarios, we extended it to support connections to the QSFP+ port on the FPGA.

We have also implemented an offload engine for Accio's fast-path UDP traffic and integrated it with IceNIC. Upon receiving packets, our NIC communicates with the ICT to determine where packets should be routed to the traditional network path or to the offloaded UDP engine. The NIC is connected directly to the system bus and bypasses the PCIe interconnect. We chose to avoid PCIe, as connecting to the system bus was a more natural design, given that we are changing the CPU and NIC. It is known that PCIe round-trip latency between the CPU and NIC for 128 byte transfers is $\approx 0.9\mu sec$ [36]. The equivalent measurement on our system is 70 cycles round-trip, which at our FPGA clock frequency of 50 MHz corresponds to $1.4\mu sec$.

The normal traffic engine is based on IceNIC, with one key difference: whereas IceNIC pre-allocates a single packet to the RX engine and reallocates it again on each interrupt, we pre-allocate multiple buffers. This eliminates the need for repeated software allocations under high incoming packet rates. The offload engine communicates with the ICT through a sideband request interface, exchanging information such as ring buffer sizes and physical addresses for received or transmitted packets.

The ICT is implemented as a peripheral device connected directly to the system bus, bypassing the PCIe interconnect and allowing the system to use MMIO to access its registers. An ICT kernel driver creates DMA-capable ring buffers in memory for fast-path connections and assigns an available ring to each application that requests one. The driver memory-maps the ring into userspace, enabling direct read and write access without additional kernel involvement. It also routes interrupts from the NIC to the system for each connection, triggering them only when the corresponding application is suspended. Our implementation aggregates interrupts for multiple fast-path connections on a single CPU core, and the interrupt handler in the kernel acknowledges all the connections in the interrupt.

Both the ICT and NIC drivers expose registers to userspace as files in the */dev* directory. Applications open these files and map ICT ring buffer memory regions into their address space for direct access, eliminating system call overhead after this initial mapping.

Overall, our Accio's prototype required a fairly modest hardware implementation effort. The Chisel RTL comprises 506 lines of code for ICT, 843 lines for the UDP offload engine, and 3,600 lines in total for the NIC and ICT system.

Prototype Limitations

Our FPGA prototype is handicapped in comparison to servers in two ways. First, it runs at 50 MHz as opposed to the servers, which run at 2.6 GHz. Hence, instructions take a lot longer to execute and we would expect computations to be considerably slower. Second, it uses the existing Rocket system bus, which has a bit-width of 64 bits and a maximum bandwidth 400 MBps (at 50 MHz) or 3.2 Gbps. This is not an inherent limitation in the Accio architecture, but an artifact of the prototype. For context, PCIe Gen 4x16 used in the servers can achieve a bandwidth of 64 GBps. Some of the measurements of Accio performance are limited by the bus bandwidth, and therefore appear slower than a full implementation.

In Section 8.2, we discuss how we conservatively scale our computations to account for the frequency difference. However, our experimental results do not compensate for the bandwidth limitations in the Accio measurements.

Unimplemented Features

Some aspects of the Accio design in Sections 4, 5, and 6 are not implemented in our prototype. We stress that none of the missing features are required for our evaluations, as they do not affect the performance-critical path. First, the prototype does not include a mechanism to time-multiplex ICT slots. The driver is currently restricted to a maximum of 64 simultaneous connections. Second, we have not implemented the exception handler that reverts to the standard I/O path in the event of system error. Third, we have not implemented the TX completion interrupts, which happen when the TX ring buffer of our application is full and we need to free some memory for TX path. Currently, since our NIC is faster than our FPGA-based CPU, we never hit this scenario in our experiments. Finally, we have not implemented the mechanism to trigger an interrupt based on checking the ASID bits. Rather, we implement it as a memory-mapped register in the ICT that gets updated if there is a context switch to/from an accelerated I/O thread.

Figure 4 Experimental setup: the FPGA prototype (50 MHz) is connected via QSFP links to two servers (2.6 GHz), enabling FPGA-to-server and server-to-server communication measurements.

8 Evaluation

In our evaluation of Accio using our FPGA prototype, we address the following key questions:

1. How does the performance of our FPGA prototype compare to enterprise-grade Linux servers (§8.2)?
2. How does the latency of Accio compare to DPDK and RDMA (§8.3)?
3. What is the maximum throughput achieved that Accio can sustain (§8.4)?
4. What is the CPU utilization of Accio compared to alternative I/O implementations (§8.5)?
5. How does Accio performance scale with an increasing number of concurrent fast-path connections (§8.6)?

8.1 Methodology

Our FPGA prototype features four RISC-V Rocket cores and 2 GB of external DDR memory, which is provided by the board. The cores, system bus, peripheral bus, and all peripheral devices – including ICT and our modified IceNIC – operate at a frequency of 50 MHz. As a result, all DMA requests are issued at this bus frequency. We use a 64-bit bus width for system bus transactions, memory accesses, and peripheral accesses. The prototype uses the default Rocket Chip cache configurations: 32 KiB level 1 instruction (L1i) and data caches (L1d), and a single-bank level 2 (L2) cache with 512 KiB capacity. The prototype does not include an L3 cache. In addition, we allocate 8 KB to each ring buffer used by our UDP offload engine.

The system configuration is illustrated in Figure 4. We use two enterprise-grade servers, each equipped with a 72-core Intel Xeon Gold 6240 CPU and a 100 Gbps Intel E810 NIC. In addition, both servers are equipped with Mellanox ConnectX-5 NICs, which we use specifically for RDMA experiments since they support RDMA hardware timestamping. The server cache hierarchy includes 1.1 MiB of L1I and 1.1 MiB of L1D, 36 MiB of L2, and 49.5 MiB of L3. Our Accio prototype connects to one server via QSFP, while the second server connects to the first server via QSFP, mirroring the FPGA link.

In terms of OS, on the servers, we use Ubuntu 24.04 with Linux kernel 6.6.0. For the FPGA, we use a modified version of a Linux distribution built with Buildroot [6] (so that it fits on our FPGA hardware) with kernel 6.6.0. Each kernel can be built with debugging features enabled or disabled. Debugging features do not affect the kernel-bypass measurements (i.e. the kernel is bypassed). On the FPGA, we disabled debugging unless otherwise specified.

On the first test server, we run a UDP echo application using standard Linux sockets. Clients run on both the FPGA and the second server, and we measure packet processing time on the client side. The processing time of the echo server is always attributed to the network and excluded from our measurements. For comparison, we also run DPDK and RDMA implementations on the second server and compare their results with Accio running on the FPGA.

Table 2 CoreMark benchmark results comparing the FPGA prototype with the two test servers.

Platform	CoreMark
FPGA	108.51
Test Server 1	23018.49
Test Server 2	23369.94

We do not rely on simulation tools for benchmarking. Instead, to account for the slower FPGA clock, we record packet timestamps at the FPGA network interface and measure the time spent inside the FPGA. Since all subsystems run at 50 MHz, we scale these measurements by the ratio of the FPGA frequency (50 MHz) to that of a standard server CPU (2 GHz) to estimate server-side latency. Specifically, we apply the following formula to scale FPGA latencies to the target frequency:

$$T_{adj} = (\mathrm{RTT}_{\mathrm{sw}} - \mathrm{RTT}_{\mathrm{hw}}) \times \frac{\mathrm{Frequency}_{\mathrm{FPGA}}}{\mathrm{Frequency}_{\mathrm{Standard}}}$$

where T_{adj} denotes the adjusted packet processing time on the FPGA, $\mathrm{RTT}_{\mathrm{sw}}$ the round-trip time measured with software timestamps, and $\mathrm{RTT}_{\mathrm{hw}}$ the round-trip time measured with NIC hardware timestamps.

Measuring the processing time on the client is complicated for DPDK. Ideally, we would collect timestamps at the NIC at the time of packet transmission and reception. Unfortunately, the Intel NIC doesn't support TX timestamps for UDP packets. Therefore, we use a more optimistic estimate for DPDK by taking the TX timestamp in the DPDK library running on the CPU before the packet is sent. This methodology provides a slightly favorable view of DPDK processing time (essentially removing the PCIe on the TX side).

8.2 Frequency-Scaling Validation

We adjust the time taken on the FPGA by scaling the clock frequency of the FPGA prototype to that of a standard server. Our FPGA prototype includes both the CPU and the NIC. This makes accurate scaling of the performance to a server-class system more challenging. The CPU would have a more aggressive microarchitecture in a server compared to the simple RISC-V core, and hence its performance would be better than simple clock frequency scaling might indicate. The performance of the NIC, on the other hand, would have a scaling closer to clock frequency ratios.

Hence, we take a conservative approach – simple clock frequency scaling – for our evaluation. In this section we compare the baseline performance of a standard server to our FPGA-based RISC-V system, showing that scaling FPGA results by clock frequency provides an approximate measure of standard server performance.

To understand the differences in our server microarchitecture performance vs. the simple RISC-V core, we run the CoreMark benchmark [9] on both platforms using identical input parameters and default settings for each system. Table 2 summarizes the results: the FPGA prototype achieves roughly 200× lower performance than the test servers. Although our servers are labeled 2.6 GHz, the actual clock frequency varies dynamically at runtime. On average, we observe that the average clock frequency reported by the kernel was ≈ 2 GHz. Therefore, we scale the FPGA results by only the frequency ratio of 40× (2 GHz / 50 MHz). Because this is a conservative estimate that does not account for CPU architecture differences (in CoreMark), our scaling approach handicaps the Accio prototype.

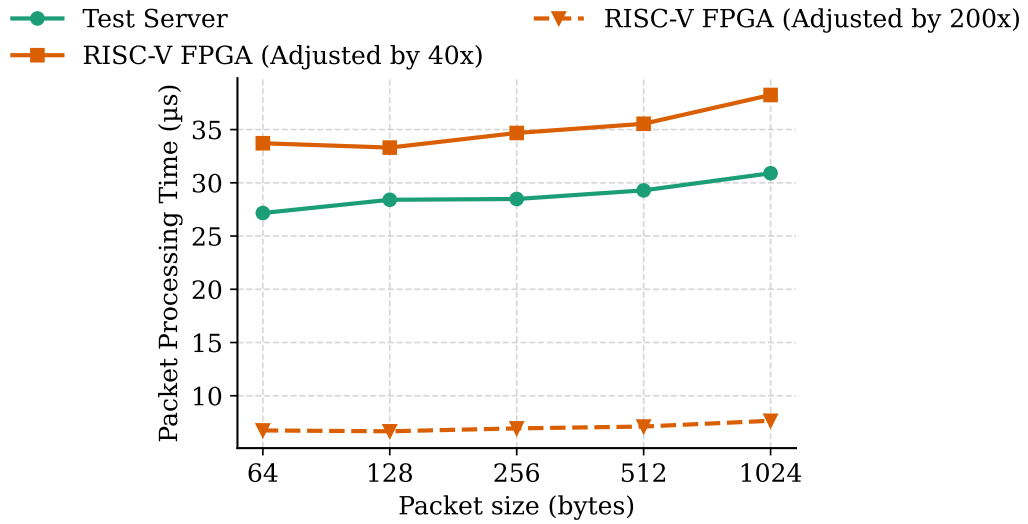


Figure 5 Average packet processing time across message sizes, comparing the FPGA prototype with test servers running Linux sockets.

To validate this frequency-scaling approach in a networking context, we use a simple ping-pong experiment implemented with Linux sockets. This is the only experiment where we enable kernel debugging features on the FPGA, because the baseline server setup also had kernel debugging features enabled. Our benchmark sends 10K UDP packets of varying sizes to the first test server, waits for responses, and measures the round-trip time (RTT). We run the same Linux socket application on both the FPGA and standard servers.

Figure 5 presents the findings. Across all packet sizes, the 40× scaled FPGA RTTs are consistently worse than, but remain in the same order of magnitude as, the server RTTs. We also present the FPGA results scaled by the ratio of our computational power compared to that of our test servers, which is 200×, indicating a lower bound on the latency achievable by Accio. Using 40× as the scaling factor, the benefit of removing PCIe for Accio is $\approx 0.86 \mu sec$ in round-trip latency.

Takeaway: Frequency scaling provides a conservative yet meaningful estimate of FPGA-based performance relative to standard Linux servers in packet processing workloads. The 40× and 200× scaling curves provide bounds on Accio's performance, with the actual results falling in between. For networking-intensive workloads, performance is closer to the 40× scaling, since execution time is dominated by the NIC.

8.3 Accio Latency

Next, we evaluate the latency of Accio, compared to standard Linux sockets, DPDK, and RDMA. We run a UDP echo server on the first test server, while the client side runs either Accio on the FPGA or one of the three points of comparison on the second server. We measure RTTs for packet sizes ranging from 64 B to 1024 B and using NIC hardware timestamps. We report the processing time, which includes just the time spent on the client processing the packet.

Figure 6 presents the results. The actual performance of Accio is expected to lie between 40× and 200× curves, since even scaling the Rocket cores to 2 GHz would not approach the computational power of a standard server processor. Accio 40× reduces latency to $1/12^{th}$

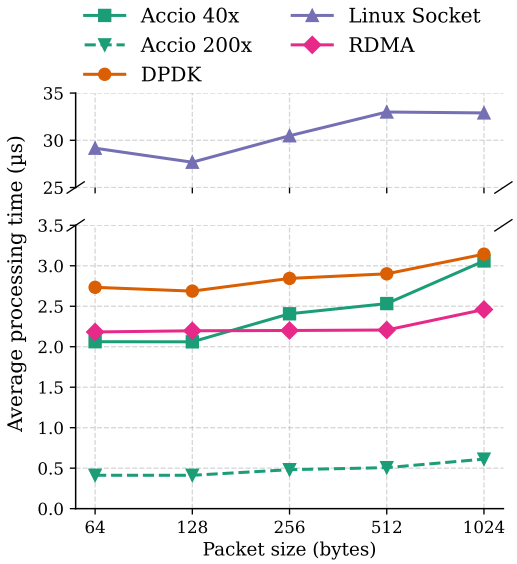


Figure 6 Packet processing latency of a UDP ping-pong application across packet sizes. The top subfigure compares Accio, DPDK, and RDMA, while the bottom subfigure additionally includes Linux sockets as a baseline.

of that of Linux sockets and achieves performance comparable to DPDK and RDMA. For small 64 B packets, Accio outperforms DPDK by about 31%. As packet sizes increase, this advantage diminishes, as expected, since our Rocket Core bus width is only 64 bits and requires more cycles to transmit larger packets. The 200× curve reaches around 0.5 μs, illustrating the upper bound of Accio's potential performance.

Takeaway: Accio preserves the simplicity of blocking I/O primitives while significantly reducing latency relative to Linux sockets, reaching performance on par with kernel-bypass. Accio effectively bridges the gap between ease of use and high-performance network I/O.

8.4 Accio Throughput

To evaluate Accio's throughput, we run a UDP sink application on Accio that continuously receives and discards incoming messages on the FPGA. On the first test server, we deploy a client application that transmits a constant stream of 64 B UDP messages to the FPGA over the network. We observe that Accio saturates the FPGA bus bandwidth, achieving a throughput of 3.1 Gbps.

Takeaway: Scaling up 40×, Accio could easily saturate a 100 Gbps link, providing throughput comparable with RDMA. This is achieved by always staying in the fast path, and by allocating a sufficiently large ring buffer for each connection.

8.5 CPU Utilization

In this experiment, we focus on evaluating the single-core CPU utilization of a process using Accio. Specifically, we measure actual CPU usage on the FPGA, rather than adjusting for latency. To do so, we run a UDP echo client on the FPGA that sends 10,000 packets of 64 B

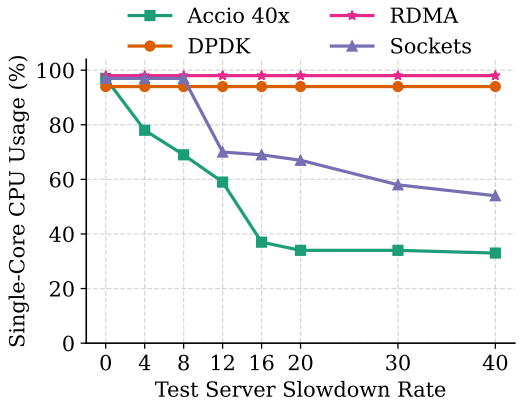


Figure 7 CPU utilization of a UDP ping-pong application under varying test server slowdowns, comparing Accio, DPDK, RDMA, and Linux sockets.

to the first test server, testing both Accio and Linux sockets. Since the FPGA is much slower than the server, we artificially slow down the server response time in controlled increments. This setup allows us to approximate Accio's CPU usage if the server were to operate at the same speed as the FPGA.

Figure 7 shows that Accio reduces CPU usage as the server slows down. At a 40× slowdown, Accio uses 33% CPU compared to 54% for Linux sockets, nearly increasing efficiency by 1.6×. By contrast, DPDK and RDMA use 100% CPU due to busy polling.

Takeaway: Accio reduces CPU usage through interrupts, enabling scalable and efficient multicore processing without compromising performance.

8.6 Connection Multiplexing

One of the claims of Accio is that it allows the OS to manage hardware resources (as it was intended to do). To evaluate this claim, we conducted an experiment in which we increased the number of concurrent connections by running multiple UDP client/server pairs for an echo server. Each UDP client generates 100K 64-byte packets.

DPDK requires that an application have direct control of the NIC. The recommended solution for multiplexing is to have one process that reads from the NIC and distributes messages to other processes via shared memory FIFOs. Our implementation follows the recommended approach [34]. RDMA hardware supports multiple queue pairs, which we used in this experiment. Note that RDMA imposes an upper bound (based on hardware constraints) on the number of queue pairs, while Accio imposes no such limitations. This experiment did not exceed the RDMA queue pair limit.

As shown in Figure 8, RDMA offers the best performance, as expected, since queue pair capacity is not a problem. Accio performance for 1 and 4 connections is similar. We see some performance loss for Accio at 64 connections, because there are only 4 cores on the FPGA, so there is context switch overhead. DPDK performance is the slowest, due to the additional shared-memory copy.

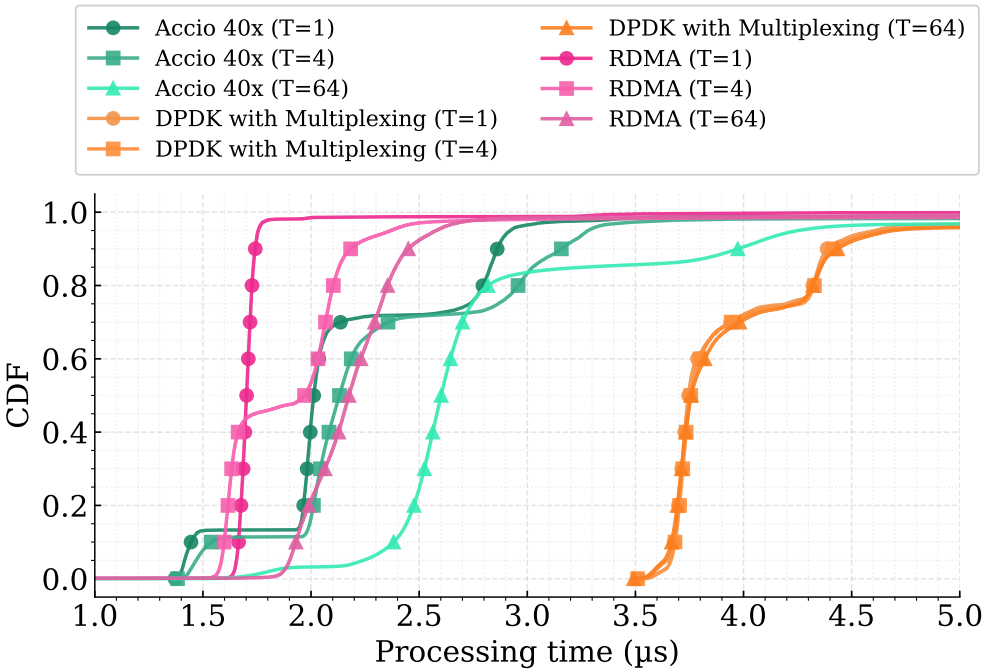


Figure 8 CDF of packet processing time for Accio, RDMA, and DPDK under varying process loads (T = number of concurrent processes).

Takeaway. Accio provides instant wake-up and maintains near single-process latency up to the core count, providing performance close to RDMA. Unlike DPDK, Accio multiplexes transparently and efficiently, avoiding shared-memory communication and enabling scalable, isolated connections.

9 Related Work

Accio introduces a novel OS-Architecture co-design to enable fast I/O while maintaining traditional APIs. In this section we review relevant related work.

Software Optimizations

Software methods for overcoming network bottlenecks are well-studied [48]. Zero-copy interfaces eliminate redundant copies [46] and often rely on techniques such as copy-on-write or deferred writes, which postpone costly memory operations until they are actually needed. Hardware offloads are also common; most modern NICs handle checksum operations, which reduces CPU load. Further optimizations include batching and specialization. For example, integrated layer processing saves effort by handling groups of packets at once, and interrupts are batched for collective software processing [35].

However, these techniques only partially reduce the inherent overheads. Kernel-based stacks are limited by the kernel's infrastructure; packets traverse multiple layers of accounting, debugging, and generalization before reaching the application. In contrast, Accio co-designs the OS and hardware so that the data plane bypasses the kernel, while remaining compatible with standard APIs. This approach delivers low latency and resource efficiency.

Kernel-Bypass Tools

Kernel-bypass techniques avoid OS overhead by giving applications direct access to NIC buffers and removing the costs of metadata handling, system calls, and redundant copies [42, 14, 7, 15].

Building on this approach, Accio also maps NIC buffers into application memory and removes redundant copies. While Accio retains a single system call for waiting on I/O, it still achieves single-threaded latency comparable to kernel-bypass tools. Offloading connection multiplexing to hardware further reduces the overhead of using shared memory between processes, thereby improving overall efficiency.

Userspace Networking Stacks

Instead of optimizing the kernel itself, several recent systems relocate the network stack into userspace, trading OS isolation for lower latency and higher throughput [30, 22].

TAS [22] avoids privilege switches and software interrupts. It implements TCP as a userspace process using DPDK [14], separating the fast path for common-case send/receive operations from the slow path for management mechanisms. This design dedicates cores to a busy polling loop, enabling efficient and low-latency data transfer.

Snap [30], in contrast, adopts a centralized, microkernel-inspired design to prioritize performance and deployment velocity. Like TAS, it dedicates cores to busy polling loops for low-latency data transfer. Applications interact with Snap through shared-memory queues or Unix domain sockets, and a custom packet injection driver enables efficient userspace communication with the NIC.

Both TAS and Snap utilize busy polling loops for low-latency application data transfer, sacrificing overall utilization. Accio achieves this instead via a hardware connection table that routes interrupts directly to application cores and utilizes small NIC offload engines to communicate directly with the NIC via MMIO, thereby avoiding kernel network stack overhead while maintaining better CPU utilization.

Offloading Features to NIC

A growing body of work explores pushing functionality traditionally handled by the OS into the NIC itself [29, 2, 44, 24]. RingLeader [29] is designed to offload intra-server orchestration – software-informed load balancing and request scheduling – onto a programmable NIC, using a lightweight OS-NIC interface for close coordination and NIC-assisted CPU scheduling. By deploying fine-grained queuing and scheduling logic in NIC hardware, RingLeader reduces CPU overheads and improves tail latency compared to software-only orchestrators.

In contrast, Tonic [2] emphasizes flexibility by enabling programmable transport protocols in high-speed NICs via a hardware template and a simple API. Rather than relying on fixed-function transport offloads (e.g., TOE/RDMA), Tonic provides a flexible, NIC-resident pipeline that leverages transport state machines and enables per-packet processing within the NIC hardware.

Accio utilizes hardware assistance for scheduling decisions and moves the transport protocol to hardware. However, by using blocking I/O and routing interrupts to correct cores, we achieve significantly better CPU utilization.

Hardware Redesign

Another line of work explores hardware redesigns. These approaches introduce CPU-NIC and software-NIC co-designs to address I/O bottlenecks at the architectural level. For example, Enso [43] modifies the NIC-application communication protocol over PCIe and dedicates

continuous buffers for streaming protocols to improve cache locality. ZeroNIC [45] separates the control and data paths. It directly places packet payloads into application buffers and posts completions to a spin polling userspace library. Nebula [47] and NanoPU [19] target small RPCs by bypassing PCIe and delivering requests directly into the L1 cache or CPU registers, thereby immediately waking applications.

Accio falls into this category by co-designing the CPU, software stack, and NIC. It introduces lightweight hardware structures such as the I/O Connection Table achieving kernel-bypass-level performance. Unlike the above systems, however, Accio is not restricted to tiny RPCs or specialized streaming protocols. This makes it more broadly applicable.

10 Conclusion

In this paper, we have presented Accio. While prior work has historically been reluctant to modify the CPU, Accio proposes a novel software/hardware co-design inspired by virtual memory systems. The Accio design includes dedicated hardware for interrupt management, a hardware assist for thread scheduling, tables in hardware that manage I/O state, and modifications to the operating system to support the new hardware. Overall, Accio dramatically improves I/O performance while maintaining existing POSIX-standard APIs.

References

1 AMD. High-Bandwidth Memory (HBM). `https://web.archive.org/web/20240830074601/https://www.amd.com/system/files/documents/high-bandwidth-memory-hbm.pdf`. Accessed: September 2025.

2 Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, pages 93–110, 2020. URL: `https://www.usenix.org/conference/nsdi20/presentation/arashloo`.

3 Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI'99, pages 45–58, 1999. URL: `https://dl.acm.org/citation.cfm?id=296810`.

4 Michael Beck, Harold Bohme, Ulrich Kunitz, Robert Magnus, Mirko Dziadzka, and Dirk Verworner. *Linux Kernel Internals*. Addison-Wesley Longman Publishing Co., Inc., USA, 1996.

5 Sharon Brunett, John Thornley, and Marrq Ellenbecker. An initial evaluation of the tera multithreaded architecture and programming system using the C3I parallel benchmark suite. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–19, USA, 1998. IEEE Computer Society.

6 Buildroot - making embedded linux easy. `https://buildroot.org/`. Accessed: September 2025.

7 Intel Corporation. Persistent memory development kit (PMDK). `https://pmem.io/pmdk/`. Accessed: September 2025.

8 Poll mode driver. `https://doc.dpdk.org/guides-22.07/prog_guide/poll_mode_drv.html`. Accessed: September 2025.

9 EEMBC. CoreMark Benchmark. `https://www.eembc.org/coremark/`, 2009. Accessed: September 2025.

10 Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. I/o is faster than the cpu: Let's partition resources and eliminate (most) os abstractions. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 81–87, New York, NY, USA, 2019. Association for Computing Machinery. `doi:10.1145/3317550.3321426`.

11 Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'18, pages 51–66, Renton, WA, April 2018. USENIX Association. URL: `https://www.usenix.org/conference/nsdi18/presentation/firestone`.

12 Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. Corundum: An open-source 100-Gbps NIC. In *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, FCCM'20, 2020.

13 John Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Commun. ACM*, 4(10):435–436, October 1961. `doi:10.1145/366786.366800`.

14 Linux Foundation. Data Plane Development Kit (DPDK). `http://www.dpdk.org`. Accessed: September 2025.

15 Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo 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, 2024. URL: `https://www.usenix.org/conference/nsdi24/presentation/fried`.

16 Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM'16, pages 202–215, New York, NY, USA, 2016. Association for Computing Machinery. `doi:10.1145/2934872.2934908`.

17 Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 135–148, 2012. URL: `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/han`.

18 Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. PASTE: a network programming interface for non-volatile main memory. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 17–33, 2018. URL: `https://www.usenix.org/conference/nsdi18/presentation/honda`.

19 Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The NanoPU: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'21, pages 239–256, 2021. URL: `https://www.usenix.org/conference/osdi21/presentation/ibanez`.

20 Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference*, USENIX ATC'16, pages 437–450, Denver, CO, June 2016. USENIX Association. URL: `https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia`.

21 Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, et al. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, ISCA'18, pages 29–42. IEEE, 2018.

22 Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference*, EuroSys'19, pages 1–16, 2019. `doi:10.1145/3302424.3303985`.

23 T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F H SUMNER. One-level storage system. *IEEE Transactions on Electronic Computers*, EC-11(2):223–235, April 1962. doi:10.1109/TEC.1962.5219356.

24 Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. Rearchitecting the TCP stack for I/O-Offloaded content delivery. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'23, pages 275–292, 2023. URL: https://www.usenix.org/conference/nsdi23/presentation/kim-taehyun.

25 Xinhao Kong, Jingrong Chen, Wei Bai, Yechen Xu, Mahmoud Elhaddad, Shachar Raindel, Jitendra Padhye, Alvin R. Lebeck, and Danyang Zhuo. Understanding RDMA microarchitecture resources for performance isolation. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'23, pages 31–48, Boston, MA, April 2023. USENIX Association. URL: https://www.usenix.org/conference/nsdi23/presentation/kong.

26 Xinhao Kong, Yibo Zhu, Huaping Zhou, Zhuo Jiang, Jianxi Ye, Chuanxiong Guo, and Danyang Zhuo. Collie: Finding performance anomalies in RDMA subsystems. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'22, pages 287–305, Renton, WA, April 2022. USENIX Association. URL: https://www.usenix.org/conference/nsdi22/presentation/kong.

27 I.M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, 1996. doi:10.1109/49.536480.

28 Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM'16, pages 1–14, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2934872.2934897.

29 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, April 2023. URL: https://www.usenix.org/conference/nsdi23/presentation/lin.

30 Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 399–413, 2019. doi:10.1145/3341301.3359657.

31 Marshall Kirk McKusick, George V Neville-Neil, and Robert NM Watson. *The design and implementation of the FreeBSD operating system*. Pearson Education, 2015.

32 Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, page 5, 2003.

33 David Mosberger and Larry L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI'96, pages 153–167, 1996. doi:10.1145/238721.238771.

34 Multi-process sample application. https://doc.dpdk.org/guides/sample_app_ug/multi_process.html. Accessed: September 2025.

35 NAPI, linux kernel documentation, version 6.6. https://www.kernel.org/doc/html/v6.6/networking/napi.html. Accessed: September 2025.

36 Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM'18, pages 327–341, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3230543.3230560.

37 NIC offloads. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-nic-offloads. Accessed: September 2025.

38 ntop. PF_RING. https://www.ntop.org/products/packet-capture/pf_ring/. Accessed: September 2025.

39 NVidia. ConnectX NICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/. Accessed: September 2025.

40 Vivek S Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000. doi:10.1145/332799.332895.

41 Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys'12, pages 337–350, 2012. doi:10.1145/2168836.2168870.

42 Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 9, 2012.

43 Hugo Sadok, Nirav Atre, Zhipeng Zhao, Daniel S. Berger, James C. Hoe, Aurojit Panda, Justine Sherry, and Ren Wang. Enso: A streaming interface for NIC-Application communication. In *17th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'23, pages 1005–1025, July 2023. URL: https://www.usenix.org/conference/osdi23/presentation/sadok.

44 Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'22, pages 87–102, 2022. URL: https://www.usenix.org/conference/nsdi22/presentation/shashidhara.

45 Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, et al. High-throughput and flexible host networking for accelerated computing. In *18th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'24, pages 405–423, 2024.

46 Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'22, pages 431–445, 2022. URL: https://www.usenix.org/conference/osdi22/presentation/stamler.

47 Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The NeBuLa RPC-optimized architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA'20, pages 199–212. IEEE, 2020. doi:10.1109/ISCA45697.2020.00027.

48 George Varghese and Jun Xu. *Network Algorithmics: an interdisciplinary approach to designing fast networked devices*. Morgan Kaufmann, 2022.

49 Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science*, CloudCom'17, pages 154–161. IEEE, 2017. doi:10.1109/CLOUDCOM.2017.14.

50 Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel datapath OS architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP'21, pages 195–211, 2021. doi:10.1145/3477132.3483569.