Solros: A Data-Centric Operating System Architecture for Heterogeneous Computing

Changwoo Min[†] Woonhak Kang[§*] Mohan Kumar

Sanidhya Kashyap Steffen Maass Heeseung Jo[‡] Taesoo Kim

[†]*Virginia Tech* [§]*eBay.inc* *Georgia Tech* [‡]*Chonbuk National University*

ABSTRACT

We propose Solros—a new operating system architecture for heterogeneous systems that comprises fast host processors, slow but massively parallel co-processors, and fast I/O devices. A general consensus to fully drive such a hardware system is to have a tight integration among processors and I/O devices. Thus, in the Solros architecture, a co-processor OS (data-plane OS) delegates its services, specifically I/O stacks, to the host OS (control-plane OS). Our observation for such a design is that global coordination with system-wide knowledge (e.g., PCIe topology, a load of each co-processor) and the best use of heterogeneous processors is critical to achieving high performance. Hence, we fully harness these specialized processors by delegating complex I/O stacks on fast host processors, which leads to an efficient global coordination at the level of the control-plane OS.

We developed Solros with Xeon Phi co-processors and implemented three core OS services: transport, file system, and network services. Our experimental results show significant performance improvement compared with the stock Xeon Phi running the Linux kernel. For example, Solros improves the throughput of file system and network operations by 19× and 7×, respectively. Moreover, it improves the performance of two realistic applications: 19× for text indexing and 2× for image search.

ACM Reference Format:
Changwoo Min[†] Woonhak Kang[§*] Mohan Kumar Sanidhya Kashyap Steffen Maass Heeseung Jo[‡] Taesoo Kim . 2018. Solros: A Data-Centric Operating System Architecture for Heterogeneous Computing. In *Proceedings of Thirteenth EuroSys Conference 2018 (EuroSys '18).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3190508.3190523

1 INTRODUCTION

The exponential growth in data is already outpacing both the processor and storage technologies. Moreover, with the current semiconductor technology hitting its physical limitation, it is heterogeneous computing that is becoming the new norm to cater to the surging computational demands in a power-efficient manner. For example, various specialized processing architectures, or co-processors, such as GPUs, TPUs, FPGAs, and ASICs not only have different trade-offs (e.g., computing power, parallelism, memory bandwidth, and power consumption) but also are being proposed, developed, and commercialized.

Since their inception, these co-processors have helped host processors by offloading their tasks onto themselves. Hence, in this computation offloading model, a host application is responsible for mediating data between I/O devices and co-processors. Unfortunately, this mediation leads to both cumbersome application development and sub-optimal I/O performance without skilled developers' optimization (e.g., interleaving communication and computation). To mitigate the issue of sub-optimal I/O, while providing ease-of-programming, some research efforts have focused on natively supporting I/O operations (e.g., file system [58] and network stack [37]) on co-processors to enable applications to perform I/O operations directly to the corresponding devices.

Thus, our key observation is that *the centralized coordination of I/O operations is critical to achieving high performance while providing ease of programming. In addition, another insight, which we validate empirically, is that the current I/O stacks are not a good fit for running on data-parallel co-processors because system-wide decisions (e.g., deciding an optimal data path) are critical for performance and flexibility, and I/O stacks are complex, frequent control-flow divergent, and are difficult to parallelize.* Figure 1 shows the I/O throughput on a host and a Xeon Phi co-processor, which is a PCIe-attached co-processor running Linux; it illustrates that the throughput on the Xeon Phi is significantly slower than on a host because it is inefficient to run complex, full-fledged I/O stacks on its slow, but massively parallel, processors. To address this issue, prior works [8, 37] improve the I/O performance by designing a peer-to-peer (P2P) communication between devices over a PCIe bus. However, if the P2P communication is over a socket boundary (i.e., cross NUMA in Figure 1), we observe a surprisingly lower throughput (*8.5*×) than within a socket, which shows that we need extra insight: the centralized coordination using system-wide knowledge is critical to achieving high performance.

We propose the Solros architecture, in which the host and co-processor OSes adopt different roles to fully utilize the hardware resources. A co-processor OS, named *data-plane OS*, delegates its OS services such as file system and network stacks to the host OS, named *control-plane OS*. In our approach, the data-plane OS is a minimal RPC stub that calls several OS services present in the

[*] Work performed during his post-doctoral research at Georgia Tech.

EuroSys '18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5584-1/18/04…$15.00
https://doi.org/10.1145/3190508.3190523

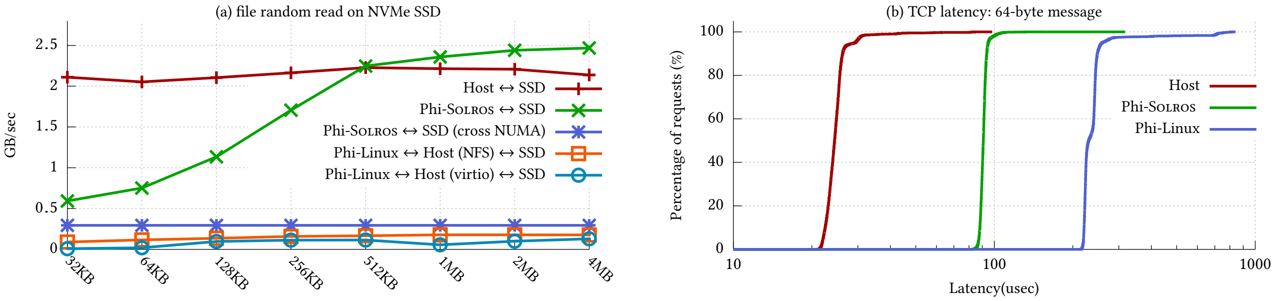


Figure 1: Comparison of I/O performance between Solros (Phi-Solros) and stock Xeon Phi (Phi-Linux). I/O performance of Xeon Phi, which runs a full-fledged Linux I/O stack (i.e., file system and TCP/IP), is significantly slower than Solros: 19× slower in file random read operations and 7× higher 99$^{\text{th}}$ percentile latency in TCP. Though recent studies [8, 37] show improving performance using peer-to-peer communication between an I/O device and a co-processor, Figure 1(a) shows that judicious use of peer-to-peer communication is required. For example, when an NVMe SSD directly accesses the memory of a co-processor *across a NUMA domain*, the maximum throughput is capped at 300MB/sec because a processor relays PCIe packets to another processor across a QPI interconnect. We observed the same problem in a few generations of x86 processors. For file IO performance with NVMe SSD, Solros shows even better performance than a host, which is supposed to be the maximum-possible performance. That is because the Solros file system architecture enables coalescing of multiple storage IO commands that reduces the number of interrupts from the device (see §4.3 and §5).

control-plane OS and provides isolation among co-processor applications, if necessary.[1] Meanwhile, the control-plane OS running on a host plays an *active* role of coordinating and optimizing communications among co-processors and I/O devices because it has a global view of a system (e.g., topology of PCIe network, load on each co-processor, and file access of co-processors), which provides an opportunity for system-wide optimizations.

The idea of delegating a part of OS services, especially I/O services, to other OSes is not new; it is widely used in virtualization [21, 38, 64], high-performance computing [22, 66], and heterogeneous computing with co-processors [37, 51, 58]. In particular, delegating I/O services on fast but less parallel host processors and using highly parallel co-processors leads to efficient utilization of such an architecture. The idea of a split OS architecture (i.e., control- and data-plane) was originally proposed by Arrakis [53] and IX [15] to achieve high IO performance for a data-center OS, which we revisit in the context of heterogeneous computing.

In this paper, we make the following contributions:

- We propose a Solros kernel architecture to provide optimized I/O services for massively parallel co-processors with a split-OS architecture, in which a co-processor (data-plane) delegates its I/O services to the host processor (control-plane), which uses system-wide knowledge to actively coordinate among devices.
- We design three core services for Solros. In particular, we design a highly optimized transport service for massively parallel processors and PCIe bus, and build two key OS services, namely, file system service and network service, on top of our transport service. Our file system service judiciously decides whether a data transfer path should use P2P or host-mediated I/O. Our network service supports the TCP socket interface and performs load balancing for a server socket, such that multiple co-processors are listening on the same address, based on user-provided load-balancing rules.
- We implement Solros using Xeon Phi co-processors and evaluate it using micro benchmarks and two realistic I/O-intensive applications. Solros achieves 19× and 7× higher throughput than Xeon Phi with a Linux kernel for file system and network services, respectively. Also, it improves the throughput of text indexing and image search by 19× and 2×, respectively.

The rest of this paper is organized as follows. In §2, we elaborate technical trends on heterogeneous computing and explain why high-performance I/O is critical. §3 describes problems in existing approaches in terms of I/O operations. We then describe Solros' design (§4) and implementation (§5). §6 presents our evaluation. §7 discusses and §8 compares Solros with previous research, and §9 provides the conclusion.

2 OBSERVATIONS

We are already observing that a wide range of computing systems, particularly data centers [9, 17, 39, 48, 52, 54, 62], are moving towards heterogeneous systems. Moreover, "we could be moving into the Cambrian explosion of computer architecture" [27], the era of processor specialization. We expect that these ongoing technical trends will further affect the architecture of modern operating systems; that is, 1) heterogeneous computing will become norm, 2) co-processors will be generic enough to support wider applications and ease of programming, and 3) I/O performance will no longer be a performance bottleneck at least in the hardware.

Specialization of general-purpose processors. Today, one of the most important aspects of application performance is application scalability. To achieve this, heterogeneous processors are now becoming an inevitable choice. This is happening for two reasons: 1) the end of single-core scaling and continued Moore's Law, which has led to the development of manycore processors and 2) the end of Dennard scaling and the Dark Silicon effect [19], which is driving the development of specialized, power-efficient processor architectures.

[1] In our implementation of Solros using Xeon Phi, the data-plane OS requires two atomic instructions for RPC and MMU for isolation among co-processor applications.

To this end, industry has developed various heterogeneous core architectures. One of the architectures uses a large number of simple cores that are similar to traditional CPU cores (i.e., smaller cache, in-order execution, wide SIMD unit, and high simultaneous multithreading level). For example, HP's Moonshot platform [29] comprises multiple server cartridges, each of which has four ARM cores and eight DSP cores. Other co-processors that belong to this category are Xeon Phi [5], Lake Crest [49], X-Gene [46], and Tilera [61]. Another one, unlike the conventional CPU-like co-processor design, uses a simpler but more massively parallel processor architecture, often called SIMT (single instruction, multiple threads). One such example is the GPGPU, which is extensively employed in deep learning these days. Additionally, industry is also adopting reconfigurable architectures, such as FPGAs, to provide even better performance per watt.

Thus, we expect that computer systems will become even more heterogeneous, as workloads determine the performance and efficiency of processors. For example, a general-purpose fat core is a good match for an application running a few complicated, control-flow divergent threads, but a specialized lean core fits the criteria of data-parallel applications.

Generalization of co-processors. The traditional use of co-processors was to offload computation for niche domains, as they had only a limited communication interface with the CPU. Unfortunately, communication complicates the co-processor programming and even makes it difficult and inefficient [13, 37, 58], such as manual allocation of the co-processor memory for data movement. This is important because of the drastic increase in the data size that co-processors should process. To overcome such limitations, co-processors are now adopting the functionality of general-purpose processors. For example, CPU-like co-processors such as Xeon Phi and Tilera run a general-purpose OS, notably Linux, to provide full-fledged OS services, including file-system and networking stacks. Similar advancements have been in the domain of GPGPUs. In particular, the Pascal architecture and CUDA 8.0 of NVidia tightly integrate both host and GPU memory management for simpler programming and memory model; GPUs now support unified virtual addressing to automatically migrate data between the CPU and GPU as well as memory over-subscription by handling page faults in the GPU [26].

Blazing fast I/O performance. Another performance-deciding criterion is I/O. Thus, I/O performance should be fast enough to keep such processors busy. Fortunately, recent advances in I/O devices has enabled fast I/O performance even in commodity devices. For instance, a single commodity NVMe SSD provides around a million IOPS and 5 GB/s bandwidth [60]. Moreover, the upcoming non-volatile memory technology promises orders of magnitude performance improvement [42], whereas on the network end, fast interconnect fabrics (e.g., InfiniBand, RoCE, Omni-Path [31], and GenZ [28]) provide extremely high bandwidth (e.g., 200 Gbps) and low latency (e.g., 600 nanoseconds [43]) among machines. Also, current PCIe Gen3 x16 already provides 15.75 GB/s and it will double (i.e., 31.51 GB/s) in PCIe Gen 4, approaching to the bandwidth of the QPI interconnect.

Summary. A machine today consists of fat host processors, lean co-processors, and fast I/O devices. To unveil the potential of hardware, tight integration of operations among processors and I/O devices is essential, and recent trends of generalizing co-processors enable such integration. However, the key challenge remains in software, specifically the operating system, to efficiently coordinate such devices.

3 THE PROBLEMS

Several few approaches enable efficient *computation* in heterogeneous systems. Computation offloading can be done with various parallel programming models and runtimes: OpenCL, OpenMP, MPI, and CUDA. More recently, Popcorn Linux [12, 13] demonstrated a seamless migration of computation between host processors and co-processors by providing a single system image. However, no well-known model and abstraction exist yet to perform *efficient I/O operations* in heterogeneous systems.

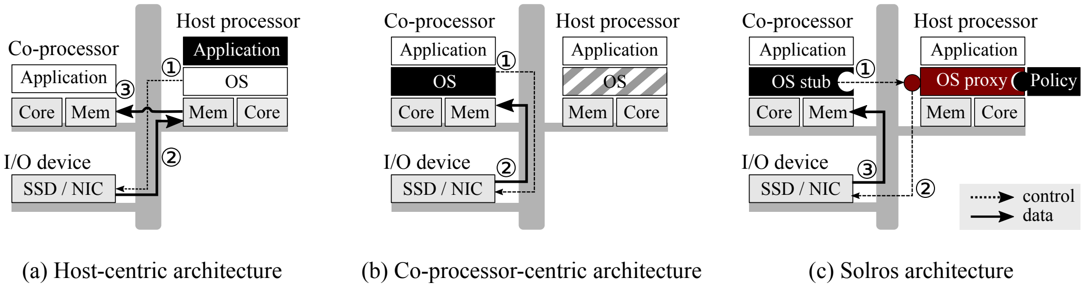
Host-centric architecture. In the traditional host-centric architecture, illustrated in Figure 2(a), a host application mediates I/O operations (e.g., file system and network operations) with co-processors since there is no I/O abstraction in co-processors. Unfortunately, this architecture complicates application development and results in sub-optimal use of hardware resources. For example, for a co-processor to read a file from an NVMe SSD, a host processor first reads the file to the host memory and then copies it to the co-processor memory, which doubles the use of PCIe bandwidth—a scarce resource for many data-intensive applications. Thus, a developer should spend significant time optimizing the application to achieve high performance, such as interleaving communication and computation.

Co-processor-centric architecture. General-purpose co-processors, like Xeon Phi or Tilera, run I/O stacks on their own, as illustrated in Figure 2(b), that simplify application development with proper I/O abstractions. For example, Xeon Phi runs Linux and supports various file systems (e.g., ext4 over `virtio`, NFS over PCIe) and network protocols (e.g., TCP/IP over PCIe) [33]. Unfortunately, I/O performance is not satisfactory because a lean, massively parallel co-processor is not a good fit to run I/O stacks that have frequent control-flow divergent code (e.g., parsing protocol header and on-disk layout of file system) and need to maintain a system-wide shared state that becomes a scalability bottleneck. Also, the co-processor-centric approach misses the opportunities of system-wide optimization and coordination. For example, as shown in Figure 1(a), P2P communication drastically degrades while crossing a NUMA boundary. Thus, system-wide optimization and coordination are critical to achieve high performance.

Summary. Providing proper I/O abstraction and high performance is essential to fully drive these heterogeneous systems. However, existing approaches miss the opportunities of system-wide optimization and coordination.

4 OUR APPROACH: SOLROS

Based on our observations and problems, we think that the heterogeneity of processors and the efficient use of I/O should be the first class consideration in designing an OS for heterogeneous systems. Considering the diversity of co-processor architectures, it is difficult (or too early) to define an operating system architecture that fits all heterogeneous architectures. Nevertheless, in an attempt



(a) Host-centric architecture (b) Co-processor-centric architecture (c) Solros architecture

Figure 2: Comparison of heterogeneous computing architectures. (a) In a traditional host-centric architecture, a host application mediates I/O data between I/O devices and co-processors. Application development is cumbersome because of no proper I/O abstraction in co-processors. In addition, performance is sub-optimal without developers' significant optimization because data is always first staged in the host memory ② and then copied into the co-processor memory ③, which doubles the use of interconnect (e.g., PCIe bus) and DMA engines. (b) In a co-processor-centric architecture, a co-processor OS has I/O stacks (e.g., file system and network stack) and initiates I/O operations. The OS delegates some of its functionalities to the host OS, which it cannot do (e.g., access I/O registers, such as a doorbell register). Performance is sub-optimal because data-parallel co-processors are not efficient to run complex and branch-divergent I/O stacks. More importantly, this approach misses opportunities of system-wide optimization and coordination. (c) In our Solros architecture, a co-processor OS (data-plane OS) is a lean RPC stub and provides isolation among co-processor applications if necessary; the host OS (control-plane OS) performs actual I/O operations. The control-plane OS actively coordinates in a system-wide fashion with a data-plane OS: it decides optimal data path considering the interconnect topology (e.g., peer-to-peer), manages shared resources among data-plane OSes (e.g., shared host-side buffer cache), and even shards incoming network packets to one the of co-processors based on user-provided load-balancing rules.

to shed some light on this, we propose our Solros architecture in which independent OSes running on processing domains (i.e., a host processor or co-processor) coordinate among themselves.

In our approach, the OSes of a host and co-processor perform different roles (see Figure 2(c)): the *data-plane OS* on a co-processor forwards OS services requests of an application (e.g., file system and network operations) to the *control-plane OS* on the host to reduce its complexity and runtime overhead. If necessary, the data-plane OS provides isolation among co-processor applications, whereas the control-plane OS on the host receives requests from data-plane OSes and performs requested operations on their behalf. A control-plane OS is not a simple proxy of a data-plane OS; since a control-plane OS has a global view of system resources, it takes a central role of optimizing communication paths among devices and load-balancing among devices. In particular, the Solros architecture has the following advantages:

Efficient global coordination among processors. A control-plane OS is a natural place for global coordination because it has global knowledge of a system, such as the topology of the PCIe network, load on each co-processor, and file access of each co-processor, which leads to system-wide optimization. For example, our file system service decides the data transfer path (e.g., peer-to-peer or host-staging copy), depending on the PCIe topology, and prefetches frequently accessed files from multiple co-processors to the host memory (see Figure 1(a)). Also, our network service performs load balancing of an incoming request of a server socket to one of the least loaded co-processors. Thus, Solros is a *shared-something* architecture, in which status in a control-plane OS is shared by multiple data-plane OSes. In Solros, only the control-plane OS directly controls IO devices (e.g., initiating DMA operations from NVMe SSD to accelerator memory), thereby protecting I/O devices from untrusted and unauthorized accesses from co-processors.

Best use of specialized processors. Our split-OS approach enables OS structures to be simple and optimized for each processor's characteristics. In particular, I/O stacks (e.g., file system and network protocol) are frequently control-flow divergent and difficult to parallelize, as shown by previous studies [15, 47] (see Figure 13). Thus, it is appropriate to run I/O stacks on fast but less parallel host processors rather than on slow but massively parallel co-processors.

Required hardware primitives. To support a wide range of co-processors, we design Solros with minimal hardware primitives: two atomic instructions (`atomic_swap` and `compare_and_swap`[2]), and the capability of the co-processor to expose its physical memory to a host for implementing a RPC stub. Thus, a data-plane OS becomes lean, as many OS services can be delegated to the control-plane OS, except essential task and memory management. Thus, adding a new OS service in a data-plane OS is mostly adding RPC stubs that communicate to the proxy in the control-plane OS. The data-plane OS should provide isolation among co-processor applications if necessary. In our implementation of Solros, we rely on the MMU of the Xeon Phi.

In the rest of this section, we first describe how a host processor sees devices (§4.1) and then elaborate on three operating services: transport service (§4.2), file system service (§4.3), and network service (§4.4).

4.1 View of the Host Processor on Devices

Modern computer systems have multiple physical address spaces to support various PCIe devices. This enables various PCIe devices (e.g., Xeon Phi, GPGPU, NIC, and NVMe SSD), which are equipped with large amounts of on-card memory, to form a separate physical address space. Besides the normal memory-mapped and I/O port spaces (MMIO and PIO), such on-card memory regions are mapped to special ranges of physical address space, called *PCIe window*, of the host. Similar to other physical memory regions, a host processor can access the mapped on-chip memory of devices using either

[2] `atomic_swap` can be emulated using a `compare_and_swap` loop.

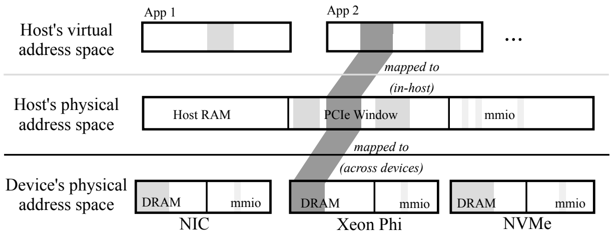

App 1
App 2
Host's virtual address space
...
mapped to (in-host)
Host's physical address space
Host RAM
PCIe Window
mmio
mapped to (across devices)
Device's physical address space
DRAM
mmio
DRAM
mmio
DRAM
mmio
NIC
Xeon Phi
NVMe

Figure 3: Illustration of the different address spaces in our setup, showing how the physical address spaces of a Xeon Phi, NIC, and NVMe are being mapped into the physical address space of the host and how applications may use it. A memory region of Xeon Phi is mapped to a PCIe window in the host; a host again maps the same PCIe window to the virtual address of a user application, which allows the user application to directly access the memory region of Xeon Phi.

`load`/`store` instructions or DMA operations. Also, these regions can have corresponding virtual address regions that allow user processes to access them (see Figure 3). Even in commodity systems without special hardware support, a host processor can control and coordinate any devices by accessing the exported physical memory and I/O ports of a device. For example, a host can initiate P2P communication between an NVMe SSD and a Xeon Phi co-processor by setting the target (or source) address of an NVMe operation to the system-mapped PCIe window of the Xeon Phi memory.

4.2 Transport Service

A control-plane OS running on a host can access each device's memory region through system-mapped PCIe windows. For the performance and simplicity of Solros, it is critical to efficiently transfer data among these memory regions in a unified manner. However, this is challenging because of the idiosyncratic performance characteristics of a PCIe network, asymmetric performance between the host and co-processors, and massively parallel co-processors.

Our transport service hides such complexity and exposes a simple interface for other OS services. In addition, it provides high scalability as well as low latency and high bandwidth with respect to manycores to run on massively parallel co-processors. To achieve high performance and high scalability over a PCIe bus, we handle four aspects in designing our transport service:

- Exploit PCIe performance characteristics (§4.2.1)
- Separate data transfer from queue operations to parallelize data access operations (§4.2.2)
- Adopt combining-based design for high scalability in massively parallel accelerators (§4.2.3)
- Replicate queue control variables (i.e., `head` and `tail`) to minimize costly PCIe transactions (§4.2.4)

We first describe the PCIe performance characteristics analysis and present the design of our transport service.

4.2.1 PCIe Performance Characteristics. System-mapped PCIe windows can be accessed using either memory operations (`load`/`store`) or DMA operations, which either a host processor or a co-processor initiates. Interestingly, PCIe performance is heavily affected by an access method and an initiator of operations.

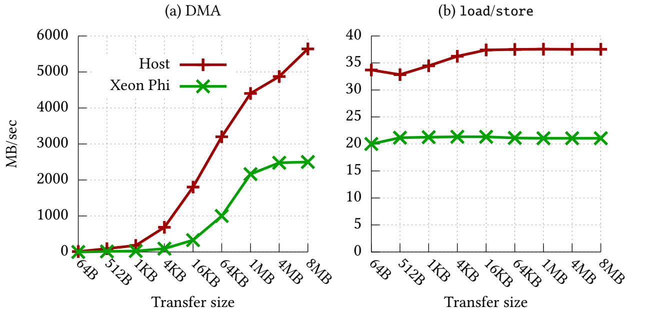


Figure 4: Bandwidth of bi-directional data transfer over PCIe between a host processor and a Xeon Phi co-processor. Bandwidth is significantly dependent on who initiates the transfer (i.e., either a host or a Xeon Phi) and transfer mechanism (i.e., either a DMA or a `memcpy` using `load`/`store` instructions).

Figure 4 shows our evaluation with a Xeon Phi co-processor (see the environment details in §6) measuring the bi-directional bandwidth of the PCIe. Each access mechanism has pros and cons. For large data transfer, a DMA operation is faster than a memory operation. For 8 MB data transfer, the DMA copy operation is 150× and 116× faster than `memcpy` in a host processor and Xeon Phi co-processor, respectively. However, for small data transfer, the latency of a memory operation is lower than a DMA operation. For a 64-byte data transfer, `memcpy` is 2.9× and 12.6× faster than a DMA copy in a host processor and a Xeon Phi co-processor, respectively. Since each `load`/`store` instruction on the system-mapped region eventually issues a PCIe transaction of a cache line size (64 bytes), there is no initialization cost (i.e., low latency for small data), but a large data will be transferred with multiple cache line-sized PCIe transactions (i.e., low bandwidth for large data). In contrast, a DMA operation requires initialization such as setting up a DMA channel (i.e., high latency for small data), but a large data will be transferred in a single PCIe transaction (i.e., high bandwidth for large data).

Another important factor that significantly affects PCIe performance is who initiates data transfer operations. As Figure 4 shows, in all cases a host-initiated data transfer is faster than a co-processor initiated one—2.3× for DMA and 1.8× for `memcpy`—because of the asymmetric performance between a host processor and a co-processor, i.e., a DMA engine and memory controller in a host processor is faster than a Xeon Phi co-processor.

4.2.2 Parallelizing Data Access Operations. We designed our transport service to achieve three goals: 1) a simple programming interface, 2) manycore scalability for massively parallel co-processors, and 3) high performance by taking care of the characteristics of the PCIe network. To this end, we provide a *ring buffer over PCIe*. As the API list in Figure 5 shows, our ring buffer is a fixed size for simple memory management but supports variable-size elements for flexibility. It allows concurrent producers and consumers. Also, it supports non-blocking operations; it returns `EWOULDBLOCK` when the buffer is empty upon a dequeue operation or when the buffer is full upon an enqueue operation so its users (e.g., file system and network stack) can decide to retry or not.

On one end, we create a *master ring buffer* that allocates physical memory. Then we associate a *shadow ring buffer* with the master ring buffer on the other end. We can send data in either direction.

```c
/** Create a master ring buffer with a given size
 *  return a master handle */
int        rb_master_init(struct rb *rb, int size);
/** Create a shadow ring buffer attached
 *  to a master ring buffer with a given master handle */
struct rb *rb_shadow_init(int master_handle);
/** Enqueue 1) enqueue data with a given size
 *  returning a ring buffer memory to fill in (rb_buf) */
int        rb_enqueue(struct rb *rb, int size, void **rb_buf);
/** Enqueue 2) copy data to the ring buffer memory (rb_buf) */
void       rb_copy_to_rb_buf(void *rb_buf, void *data, int size);
/** Enqueue 3) mark the enqueued data ready to dequeue */
void       rb_set_ready(void *rb_buf);
/** Dequeue 1) dequeue a data from the head of a ring buffer
 *  returning a ring buffer memory to copy out (rb_buf) */
int        rb_dequeue(struct rb *rb, int *size, void **rb_buf);
/** Dequeue 2) copy data from the ring buffer memory (rb_buf) */
void       rb_copy_from_rb_buf(void *data, void *rb_buf, int size);
/** Dequeue 3) make the dequeue data allow to reuse */
void       rb_set_done(void *rb_buf);
```

Figure 5: Solros transport API, which is a fixed-size ring buffer over the PCIe bus. A *master ring buffer* allocates physical memory and a *shadow ring buffer* accesses the master over PCIe. `enqueue/dequeue` operations are separated from `copy` operations so that multiple threads can access each element concurrently, and ring buffer operations and data access can be interleaved easily.

The shadow ring buffer accesses the associated master through system-mapped PCIe windows. This *master shadow buffer design* exposes a lot of flexibility for performance optimization. That is, we can exploit asymmetric performance by deciding where to put the master ring buffer. For example, when we allocate a master ring buffer at the co-processor memory, the co-processor accesses the ring buffer in its local memory and a fast host processor accesses the ring buffer via its faster DMA engine. Therefore, deciding where to locate a master ring buffer is one of the major decisions in designing OS services in Solros. Note that a master ring buffer can be located at either a sender or a receiver side, unlike previous circular buffers [18, 40] in RDMA, which only allows a receiver to allocate the buffer memory.

Unlike conventional ring buffers [18, 23, 40], we decouple data transfer from enqueue/dequeue operations to interleave queue operations with data access operations and to parallelize data access from each thread. To this end, the `rb_enqueue` and `rb_dequeue` do not copy data from/to a user-provided memory. Instead, they expose a pointer (`rb_buf`) to an element memory in a ring buffer so that a user can directly operate on the element memory. Once a user data (`data`) is copied from/to the element memory using `rb_copy_to_rb_buf`/`rb_copy_from_rb_buf`, the element needs to be set to ready to dequeue (`rb_set_ready`) or ready to reuse (`rb_set_done`).

4.2.3 Taming High Concurrency using Combining. One key challenge in designing transport service is dealing with the high concurrency of co-processors, e.g, a Xeon Phi has 61 cores (or 244 hardware threads). To provide high scalability on such co-processors, we design our ring buffer with the combining technique [20]. The key insight is that a combiner thread batches operations for other concurrent threads. Previous studies [20, 41, 55] have shown that for data structures such as stack and queue, a single thread outperforms lock-based and lock-free approaches, as it amortizes atomic operations while maintaining cache locality.

For combining, one ring buffer internally maintains two request queues for enqueue and dequeue operations, respectively. `rb_enqueue` (or `rb_dequeue`) first adds a request node to the corresponding request queue, which is similar to the lock operation of an MCS queue lock [44]. If the current thread is at the head of the request queue, it takes the role of a combiner thread and processes a certain number of `rb_enqueue` (or `rb_dequeue`) operations. As soon as it finishes processing an operation, it indicates its completion by toggling the status flag in the request node. A non-combining thread waits until its status flag is turned on. Our combining design requires two atomic instructions: `atomic_swap` and `compare_and_swap`, and significantly reduces contention on control variables (e.g., `head` and `tail`) of the ring buffer, which leads to better scalability.

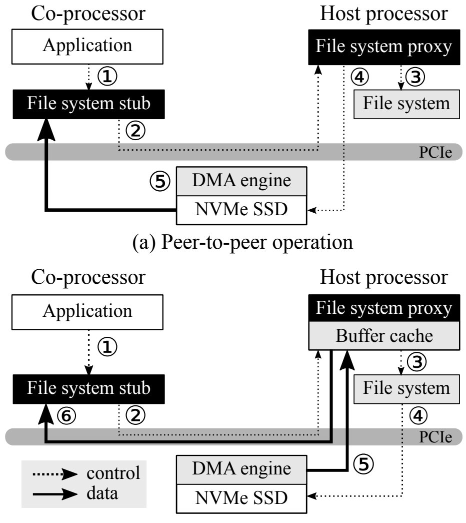
4.2.4 Minimizing PCIe Overhead by Replication. To exploit the performance characteristics of PCIe, we use two techniques. First, `rb_copy_to_rb_buf` and `rb_copy_from_rb_buf` use `memcpy` for small data and DMA copy for large data to get the best latency and throughput. In our implementation with Xeon Phi, we use a different threshold for a host and a Xeon Phi: 1 KB from a host and 16 KB from Xeon Phi because of the longer initialization of the DMA channel.

To further reduce costly remote memory accesses over PCIe, we replicate the control variables (i.e., `head` and `tail`) and defer their updates until required. This is critical because accessing a control variable over a PCIe bus issues a costly PCIe transaction. In our master shadow buffer model, a sender, which performs `rb_enqueue`, maintains the original copy of `tail` and the local replica of `head`. Similarly, a receiver, which performs `rb_dequeue`, maintains the original copy of `head` and the local replica of `tail`. This design improves performance because a sender and receiver can access `head` and `tail` in their local memory without crossing the PCIe bus. To ensure the correctness of operations, we should properly synchronize replicated variables. That is, whenever `rb_enqueue` sees if a ring buffer is full (or `rb_dequeue` sees if a ring buffer is empty), it updates its replicated variable by fetching the original value from remote. Also, a combiner thread always updates original values at the end of combining to amortize the remote access cost with the combining threshold.

4.3 File System Service

The Solros file system is designed for multiple co-processors to process a large volume of data efficiently. To this end, we run a lightweight file system stub on a co-processor and a full-fledged file system proxy on a host processor (see Figure 6). For high performance, we optimize the data transfer path between a co-processor and a disk. We perform P2P data transfer between a disk and a co-processor whenever possible, which is beneficial, as it allows a DMA engine of a disk to directly fetch data from/to memory in a co-processor. In addition, we use host-side buffer cache to improve the I/O performance of accessing data shared by multiple co-processors.

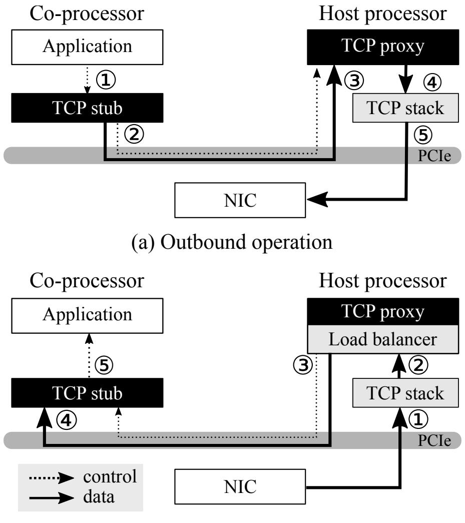
4.3.1 Data-plane OS. A lightweight file system stub transforms a file system call from an application to a corresponding RPC, as there exists a one-to-one mapping between an RPC and a file system call, which leads to trivial transformation of file system calls to RPC commands. Thus, a data-plane OS does not necessarily handle the complicated file system operations, such as maintaining directories,



(a) Peer-to-peer operation

(b) Buffered operation

Figure 6: File system service in Solros. For file system calls, the stub on the data-plane OS delegates these operations by sending RPC messages to the file system proxy in the control-plane OS. The control-plane OS decides an operation mode in either the peer-to-peer mode for zero-copy I/O between an SSD and co-processor memory or a buffered mode considering the PCIe network. If data should cross NUMA domains, it works in a buffered mode.



(a) Outbound operation

(b) Inbound operation

Figure 7: Network service in Solros. It is composed of a thin network stub on a data-plane OS and a full-featured proxy on a control-plane OS. Host DMA engines pull outgoing data to send and co-processor DMA engines pull incoming data to receive so we can fully exploit the hardware capability for data transfer. The control-plane OS performs load balancing among co-processors; it allows multiple co-processors to listen to the same port (i.e., a shared listening socket) and Solros makes load-balancing decision based on a user-provided forwarding rules.

disk blocks, and inode, which are delegated to a file system proxy in the control-plane OS.

A pair of ring buffers is used for RPC. We create their master ring buffers at the co-processor memory. Hence, RPC operations by a co-processor are local memory operations; meanwhile, the host pulls requests and pushes their corresponding results across the PCIe. Instead of transferring file data in the payload of the RPC, a data-plane OS sends the physical addresses of co-processor memory where the file data is read (or written) for zero-copy file system I/O operations.

4.3.2 Control-plane OS. A file system proxy server in a control-plane OS pulls file system RPC messages from co-processors. It executes the requested operations and then returns results (e.g., the return code of a file system call) to the requested data-plane OS.

For read/write operations that need large data transfer, the proxy provides two communication modes: 1) in the peer-to-peer communication, it initiates the P2P data transfer between a disk and memory in a co-processor, which minimizes the data transfer; 2) in the buffered communication, the proxy manages a buffer cache between disks and co-processors. The buffered communication is preferred in several cases: a cache hit occurs; a disk does not support the peer-to-peer communication (e.g., SCSI disk); a peer-to-peer communication is slower than the buffered communication depending on the data path on PCIe network (e.g., crossing a NUMA boundary); or files are explicitly opened with our extended flag `O_BUFFER` for buffered I/O operations. Besides these cases, the proxy initiates the peer-to-peer communication for zero-copy data transfer between a disk and a co-processor.

In the peer-to-peer communication, the proxy translates the file offset to the disk block address and also translates the physical memory address of a co-processor to the system-mapped PCIe window address (refer §4.1). It then issues disk commands with disk block addresses and system-mapped addresses of co-processor physical memory. Finally, the DMA engine of the disk will directly copy from/to the co-processor's memory. For file system metadata operations (e.g., `creat`, `unlink`, `stat`), the proxy calls the underlying file system and returns the results to a co-processor.

4.4 Network Service

Similar to the file system service, data-plane OS delegates its network service, particularly TCP in our current design, to the control-plane OS (see Figure 7). The data-plane OS uses RPC for its initiating socket operations, and socket events (e.g., a new client is connected or new data arrives) are delivered via an event-notification channel over our transport service. Similar to our file system service, there is a one-to-one mapping with a socket system call. The control-plane OS performs load balancing among co-processors; it allows multiple co-processors to listen to the same port and performs load balancing based on user-provided policy. Solros' network service is designed for high performance, manycore scalability, and seamless scalability across multiple co-processors.

4.4.1 RPC communication. We maintain two ring buffers for network RPC: an outbound ring buffer for `send` and `connect`, and an inbound ring buffer for `recv` and `accept`. We place those ring buffers to maximize performance; we create an outbound ring buffer as a master at a co-processor and an inbound ring buffer as a master at a host processor. By doing so, we can fully exploit the DMA engines of a host and a co-processor simultaneously; the host DMA

Module		Lines of code	
		Added lines	Deleted lines
Transport service		1,035	365
File system Service	Stub	5,957	2,073
	Proxy	2,338	124
Network Service	Stub	2,921	79
	Proxy	5,609	34
NVMe device driver		924	25
SCIF kernel module		60	14
Total		**18,844**	**2,714**

Table 1: Summary of lines of modifications

engines pull outgoing data and the co-processor DMA engines pull incoming data from the other end. In addition, the inbound ring buffer is large enough (e.g., 128 MB) to backlog incoming data or connections.

4.4.2 Event notification. We design the event notification of the Solros network service for high scalability by minimizing contention on the inbound ring buffer and high degree parallelism for data transfer. To this end, a data-plane OS has an event dispatcher thread that distributes inbound events (e.g., `recv` and `accept`) to corresponding sockets. It dequeues an event from the inbound ring buffer (`rb_dequeue`) and enqueues the ring buffer address (`rb_buf`), where data is located, to a per-socket event queue. When an application calls inbound system calls, its thread dequeues the per-socket event queue, copies the data on the inbound ring buffer (`rb_buf`) using `rb_copy_from_rb_buf`, and finally releases data on the inbound ring buffer using `rb_set_done`. This design alleviates contention on the inbound ring buffer by using a single-thread event dispatcher and maximizes parallel access of the inbound ring buffer from multiple threads. A potential problem is that the single-thread event dispatcher can be a bottleneck. However, we have not observed such cases even in the most demanding workload (i.e., 64-byte ping pong) with the largest number of hardware threads (i.e., 244 threads in Xeon Phi).

4.4.3 Load balancing. In Solros, running a network server on a co-processor is the natural way to expose the capability of a co-processor to the outside. Solros supports a *shared listening socket* for seamless scalability of network servers running on multiple co-processors. Solros allows for multiple co-processors to listen to the same address and port. When packets arrive, the network proxy decides which co-processor a packet is forwarded to. Solros provides a pluggable structure to enable packet forwarding rules for an address and port pair, which can either be connection-based (i.e., for every new client connection) or content-based (e.g., for each request of key/value store [36]). In addition, a user can use other extra information, such as load on each co-processor, to make a forwarding decision. Thus, our shared listening socket is a simple way to scale out network services using multiple co-processors.

5 IMPLEMENTATION

We implemented Solros for an Intel Xeon Phi co-processor, which is attached to a PCIe bus. A Xeon Phi co-processor runs Linux kernel modified by Intel. Thus, we implemented the control-plane OS and the data-plane OS by extending those Linux kernels (see Table 1).

Transport service. We implement our transport service using Intel SCIF [32], which provides `mmap` and DMA operations of Xeon Phi memory to a host (or vice versa). Our ring buffer is implemented as a fixed-size array that is mapped to the other side over PCIe through SCIF `mmap` API. We did two optimizations for performance. The first optimization parallelizes DMA operations on a ring buffer. As SCIF does not allow concurrent DMA operations on a mapped memory region over PCIe, we map the same memory region multiple times for concurrent DMA operations, which accounts for eight mappings because both a Xeon and Xeon Phi processor have eight DMA engines. One common problem of the array-based ring buffer is checking if an element spans at the end of the array. To avoid explicit range checking, we make our ring buffer truly circular; we `mmap`-ed the array twice to a contiguous address range so that the data access overrun at the end of the array goes to the beginning of the array.

File system service. For Xeon Phi, we implemented the file system stub under the VFS layer. For the RPC of the file system service, we extended the 9P protocol [6] to support zero-copy data transfer between a disk and a co-processor. In particular, we extended `Tread` and `Twrite` passing the physical address of a Xeon Phi instead of carrying data. We developed the file system proxy based on the `diod` 9P server [2].

For peer-to-peer operations, we translate a file offset to a disk block address and a physical memory address of Xeon Phi to a system-mapped PCIe window address. We get an inverse mapping from a disk block address from a file offset using `fiemap ioctl` [4]. One limitation using `fiemap` is that the file system proxy should run on an in-place-update file system (e.g., ext4, XFS), which does not change disk block address upon overwriting, so it is safe to directly access disk blocks. The SCIF kernel module translates the physical memory address of Xeon Phi to a corresponding system-mapped PCIe window address.

Optimized NVMe device driver. We added two `ioctl` commands (`p2p_read` and `p2p_write`) to the NVMe device driver for the file system proxy to issue peer-to-peer disk operations. These two new `ioctl` commands are IO vectors that contain all NVMe commands to process a read/write system call, which is required for a fragmented file, as multiple NVMe commands are necessary to serve one read/write file system call. Our IO vector design improves the performance by coalescing multiple `ioctl` commands into one, thereby reducing the context switching overhead to the kernel.

To further improve performance, we optimized the NVMe device driver to process these new `ioctl` commands. Our optimized driver batches all NVMe commands in one IO vector, which corresponds to one read/write system call, ringing the NVMe doorbell and receiving the interrupt only once. This approach improves performance because it reduces the number of interrupts raised by ringing the doorbell. Due to this optimization, which is only possible in Solros, the file system performance of Solros is sometimes better than that of the host (see Figure 1).

Network service. For the communication between a network proxy and a stub, we defined 10 RPC messages, each of which corresponds to a network system call, and two messages for event notification of a new connection for `accept` and new data arrival for `recv`. Our TCP stub is implemented as an INET protocol family in

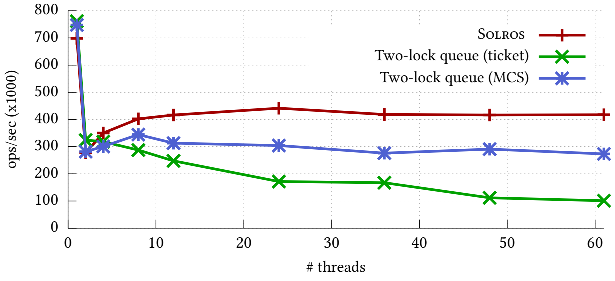


Figure 8: Scalability of Solros ring buffer for the enqueue-dequeue pair benchmark with 64-byte elements. Even under high core count, our combining-based ring buffer show superior scalability than the two-lock queue, which is a widely-implemented algorithm, without performance degradation. At 61 cores, Solros provides 1.5× and 4.1× higher performance than the ticket and the MCS-queue lock version for two-lock queues, respectively.

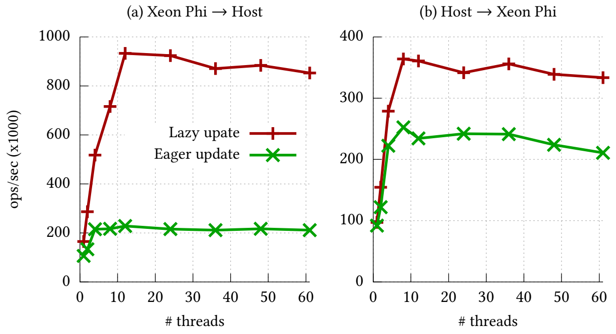


Figure 9: Performance of Solros's ring buffer over PCIe with 64-byte elements. In Figure 9(a), a master ring buffer is created at Xeon Phi and a host pulls the data over PCIe. Figure 9(b) shows the performance of the other direction. Our lazy update scheme, which replicates the control variables, improves the performance by 4× and 1.4× in each direction with decreased PCIe transactions.

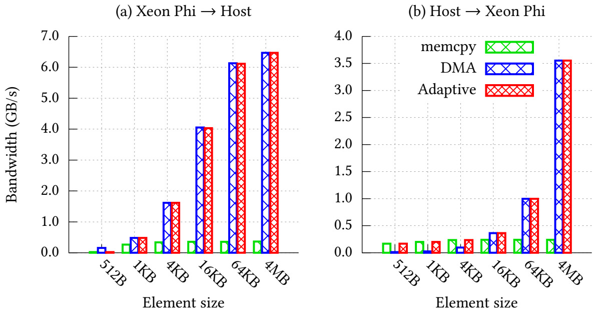


Figure 10: Unidirectional bandwidth with varying element size with eight concurrent threads. Similar to Figure 9, we create the master ring buffer at the sender so that the receiver pulls data over PCIe. For small-size data copy, `memcpy` performs better than DMA copy. For large-size data copy, it is the opposite. Our adaptive copy scheme performs well regardless of the copy size.

Linux kernel. In the TCP proxy, we implemented a connection-based round-robin load-balancing policy for a shared listening socket. The TCP proxy relies on the TCP/IP stack in the Linux kernel on a host.

6 EVALUATION

We evaluate Solros on a server running Linux kernel 4.1.0 with two Xeon E5-2670 v3 processors, each having 24 physical cores, eight DMA channels, and 768 GB memory. We use four Xeon Phi coprocessors with 61 cores or 244 hardware threads that are connected to the host via PCIe Gen 2 x16. The maximum bandwidth from Xeon Phi to host is 6.5GB/sec and the bandwidth in the other direction is 6.0GB/sec [63]. For a storage device, we use a PCIe-attached Intel 750 NVMe SSD with 1.2 TB capacity [34]. The maximum performance of the SSD is 2.4GB/sec and 1.2Gb/sec for sequential reads and writes, respectively. To evaluate network performance, we use a client machine with two Xeon E5-2630 v3 processors (16 cores or 32 hardware threads). The client machine runs Linux kernel 4.10.0 and is connected to the server through a 100 Gbps Ethernet. In all experiments running Xeon Phi with Linux TCP stack, we configured a bridge in our server so our client machine can directly access a Xeon Phi with a designated IP address.

In the rest of this section, we first show the performance consequences of our design choices for three Solros services (§6.1) and then illustrate how Solros improves the performance of two realistic applications (§6.2). Finally, we show the scalability of the control-plane OS (§6.3).

6.1 Micro benchmarks

We ran micro benchmarks to evaluate the performance and scalability of three Solros services with our design choices.

6.1.1 Transport Service. Our transport service is an essential building block for other services, so its performance and scalability is critical. We ran three micro benchmarks for our ring buffer.

Scalability. To evaluate whether our ring buffer is scalable, we ran an enqueue-dequeue pair benchmark on a Xeon Phi varying the number of threads. We created both master and shadow ring buffers at the Xeon Phi to opt out the effect of the PCIe bus. Each thread alternately performs enqueue and dequeue operations and we measured a pair of operations per second. We compare the performance with the two-lock queue [45], which is the most widely implemented queue algorithm, with two different spinlock algorithms: the ticket and the MCS queue lock.

Figure 8 shows a performance comparison with a varying number of threads. Our ring buffer performs better than the two-lock queues. The scalability of the ticket-lock variant degrades with increasing core count because of the cache-line contention on the spinlocks. Even the MCS lock, which avoids the cache-line contention, has sub-optimal scalability to Solros because our ring buffer uses the combining approach, in which a combiner thread performs batch processing for others, which results in less cache-line bouncing. Thus, at 61 cores, our ring buffer performs 4.1× and 1.5× faster than the ticket and the MCS lock variants, respectively.

Optimization for PCIe. Figure 9 shows the performance of our ring buffer over PCIe with and without replicating control variables (i.e., lazy update of `head` and `tail`). With the lazy update scheme, we maintain a replica of control variables of our ring buffer and defer the update of the replica when a ring buffer becomes full or empty, or when a combiner finishes its batch operations. The results show that the lazy update scheme significantly improves performance

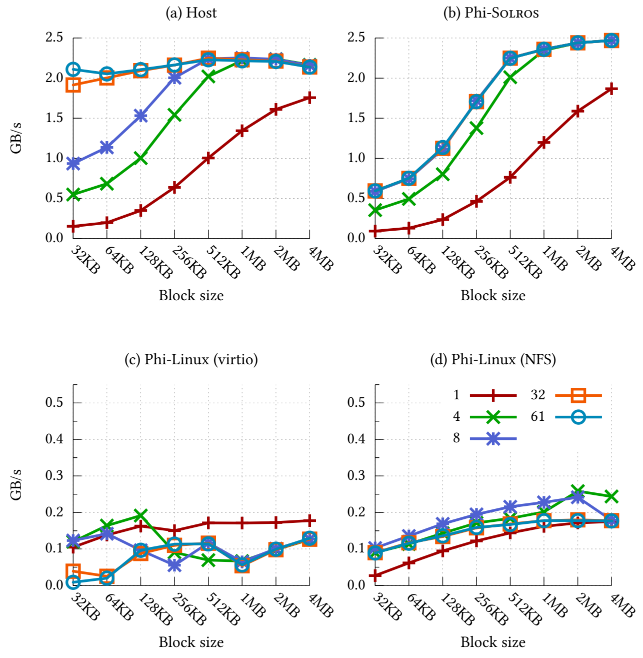


Figure 11: Throughput of random read operations on an NVMe SSD with a varying number of threads. Solros and the host show the maximum throughput of the SSD (2.4GB/sec). However, Xeon Phi with Linux kernel (`virtio` and NFS) has significantly lower throughput (around 200MB/sec).

by 4× and 1.4× in each direction by reducing the number of PCIe transactions in accessing the control variables. With the lazy update scheme, our ring buffer over PCIe performs as good as (or even better) the local version in Figure 8. We note that performance is asymmetric with respect to directions due to the asymmetric performance of the host and the Xeon Phi.

As we discussed in §4.2, the data transfer performance over PCIe is dependent on transfer mechanisms (i.e., DMA or `memcpy`). Thus, our ring buffer adaptively decides the transfer mechanism depending on the element size: use DMA if data is larger than 1KB or 16KB on a host and Xeon Phi, respectively. Figure 10 shows that our adaptive copy schemes consistently performs well regardless of the copy size.

6.1.2 File System Service. We evaluated the file system service of Solros using a micro benchmark. Figure 11 and 12 show the throughput of random read and random write operations on a 4 GB file, respectively. We compare the performance in two settings. 1) Xeon Phi with `virtio`: `ext4` file system is running on Xeon Phi and controls an NVMe SSD as a virtual block device (`virtblk`). An SCIF kernel module on the host drives the NVMe SSD according to requests from the Xeon Phi. An interrupt signal is designated for notification of `virtblk`. 2) Xeon Phi with NFS: the NFS client on Xeon Phi accesses the host file system over the NFS protocol. We also include the throughput of the host to show the maximum achievable throughput.

Solros shows significantly higher throughput over the conventional approach, running a whole file system on Xeon Phi (`virtio`).

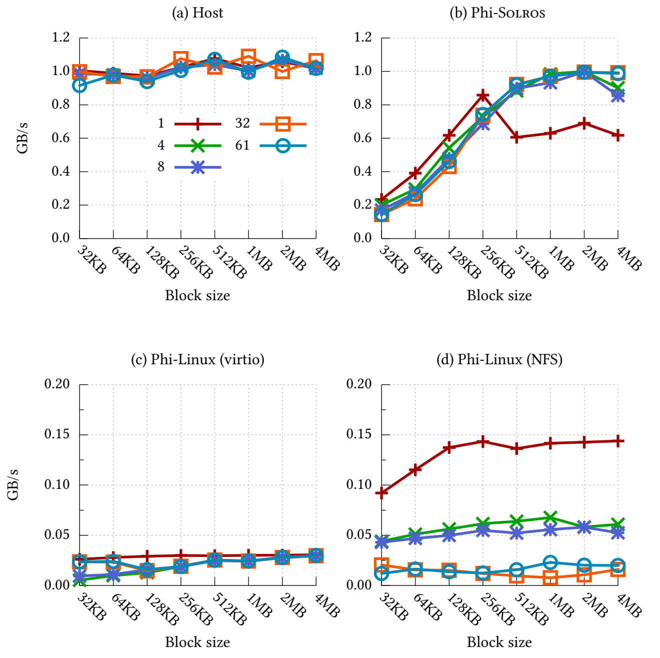


Figure 12: Throughput of random write operations on an NVMe SSD with a varying number of threads. Solros and the host show the maximum throughput of the SSD (1.2GB/sec). However, Xeon Phi with Linux kernel (`virtio` and NFS) shows significantly lower throughput (less than 100MB/sec).

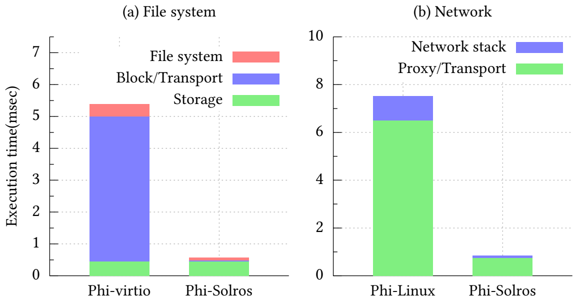


Figure 13: Latency breakdown of I/O sub-system between Solros (Phi-Solros) and stock Xeon Phi (Phi-Linux). We issued 512KB random read operations using the `fio` benchmark to profile the file system latency and ran our TCP latency benchmark with 64B message size.

In particular, when the request size is larger than 512KB, the additional communication overhead between the data- and the control-plane OSes over the PCIe bus is hidden and we achieve the maximum throughput of our NVMe SSD (2.4GB/sec for random read and 1.2GB/sec for random write operations). As Figure 13(a) shows, our zero-copy data transfer performed by the NVMe DMA engine is 171× faster than the CPU-based copy in `virtio`, and our thin file system stub spends 5× less time than a full-fledged file system on the Xeon Phi. In summary, the Solros file system service achieves the maximum throughput of an SSD and significantly outperforms the conventional approach by as much as 14× and 19× for NFS

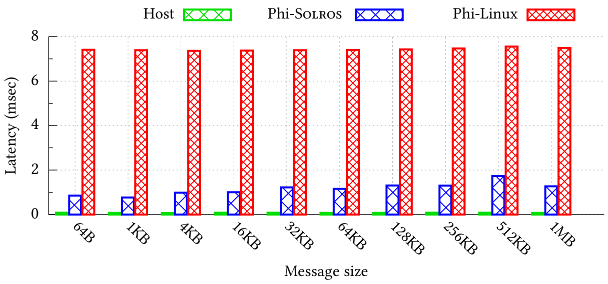


Figure 14: TCP latency with different message sizes using 61 concurrent connections. SOLROS shows superior throughput over the stock version, while running a full TCP stack on Xeon Phi.

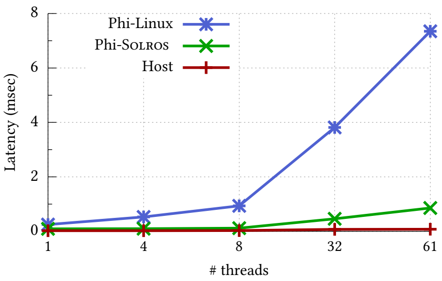


Figure 15: TCP latency for a 64-byte message varying the number of threads. SOLROS shows superior scalability over Xeon Phi with the Linux kernel. However, SOLROS also shows performance degradation as threads increase. That is because of the limitation of our implementation. The common socket layer, under which our TCP stub is implemented, becomes a scalability bottleneck.

and `virtio`, respectively. Also, it scales well without performance degradation with concurrent threads.

6.1.3 Network Service. We evaluated SOLROS' network service using a micro benchmark for TCP latency and scalability. Figure 14 shows TCP latency with different message sizes using 61 concurrent threads. We present the performance of a host for reference and compare SOLROS with the conventional approach running a full TCP stack on Xeon Phi (Phi-Linux). A TCP client runs on a different machine and is connected to the TCP server through a 100 Gbps Ethernet. Results show that our approach has significantly lower latency than the conventional one. Figure 1(b) shows the cumulative distribution of response; the 99 percentile latency of host, SOLROS, and Xeon Phi with Linux kernel are 41.2 usec, 103.1 usec, and 713 usec, respectively. As Figure 13(b) shows, our transport service is 8.7× faster and our thin network stub takes 10.9× less CPU time than the stock network stack on Xeon Phi.

Figure 15 shows the scalability of our network service with a varying number of threads. Compared to the latency at a single core, latency at 61 cores increases 9.2× and 29.5× for SOLROS and Xeon Phi with a full TCP stack, respectively. SOLROS shows significantly higher manycore scalability than the stock Xeon Phi version. However, SOLROS' scalability is limited because the TCP stub of SOLROS is implemented as an INET protocol family under the common socket layer, which becomes a scalability bottleneck.

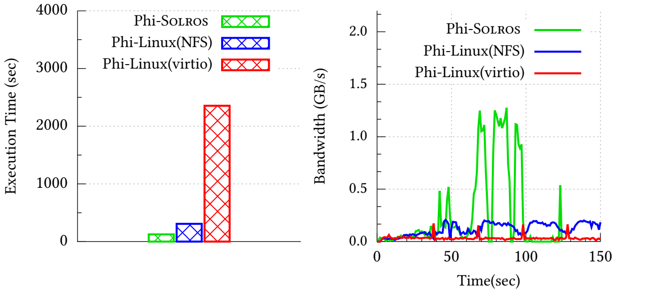


Figure 16: Performance comparison of CLucene, text search engine, with SOLROS, NFS, and `virtio`. SOLROS performs 19× faster than ext4/`virtio` running on Xeon Phi.

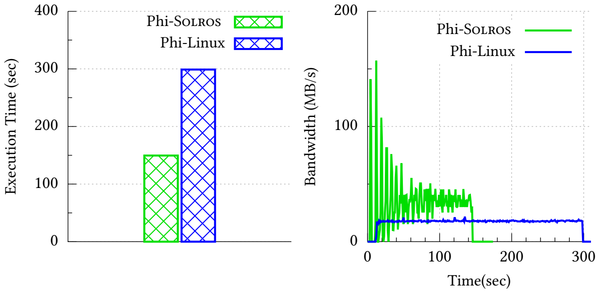


Figure 17: The performance of our image search server running on a Xeon Phi co-processor and network bandwidth over time. SOLROS version is 2× faster than the version with Linux kernel.

6.2 Application Benchmarks

We now consider two realistic I/O-intensive applications: text search and image search. In particular, we investigate how SOLROS services help to improve application performance and seamlessly scale with multiple Xeon Phi co-processors.

6.2.1 Text Search. CLucene [1] is a text search engine written in C++. To see how the SOLROS file system service affects the performance of CLucene, we perform full-text indexing in parallel. CLucene indexers, running on a Xeon Phi co-processor, read text files from an NVMe SSD and then create index files on the NVMe SSD. We created 122 indexer threads for our evaluation. The text files are composed of 20,000 files in a directory (20 GB in total). CLucene relies on read, write, and directory enumeration file system calls for indexing files.

For comparison, we ran CLucene on three different file systems: `virtio` and NFS for Linux on Xeon Phi, and SOLROS. As Figure 16 shows, CLucene with SOLROS significantly outperforms CLucene with the `virtio` and NFS file systems. The I/O bandwidth shows that the SOLROS file system are critical to achieving superior performance by keeping these massively parallel co-processors busy. In particular, `virtio` shows significantly lower performance even than NFS on Linux. That is because the interrupt notification of `virtio` has significant overhead as more concurrent threads perform IO operations. This was also observed in Figure 11, where a single-thread read in `virtio` is faster than multi-thread read operations. Figure 19(a) shows the execution time breakdown and confirms that IO performance is critical to achieve high performance by making massively parallel co-processors fully busy.

6.2.2 Image Search. For an image search server, we extended the `netferret` content-based image similarity search in PARSEC 3.0 [16] and optimized its processing pipeline to remove scalability bottlenecks with the large number of threads. The image search server loads an image database from an NVMe SSD during its initialization and then receives a JPEG image from a client and finally returns the five most similar images in the image database. It is composed of three pipeline stages: 1) a single-threaded `listen/accept` stage, 2) a multi-threaded `recv` and similarity comparison stage, and 3) a multi-threaded `send` stage to send a list of similar images to the client. The client also consists of three stages: 1) a single-threaded scan stage to get a list of JPEG files in a directory, 2) a multi-threaded `send` stage to send JPEG files one by one, and 3) a multi-threaded output stage to get a response from the image search server. We measure the time to query 1,000 JPEG images (5 GB in total).

We ran experiments in two configurations: Xeon Phi with the Solros network service and Xeon Phi with Linux TCP stack. As Figure 17 shows, the performance of the image server with Solros network service is 2× faster than that with Linux TCP stack on Xeon Phi. Figure 19(b) shows the execution time breakdown and confirms that IO performance is critical to achieve high performance by making massively parallel co-processors fully busy.

To see how the Solros network service helps scale-out of a server application running on multiple co-processors, we ran our image search server on multiple Xeon Phi co-processors up to four. When two or more Xeon Phi co-processors are involved, the image search server running on each Xeon Phi listens on the same port, so the Solros network service treats it as a shared listen socket. When a new client connection comes in, the Solros network service forwards a connection to one of Xeon Phi co-processors in a round-robin way. As Figure 18 shows, the performance of the image search scales linearly as more Xeon Phi co-processors are used.

6.3 Scalability of Control-Plane OS

In the Solros architecture, the control-plane OS serves requests from multiple data-plane OSes for performance and better global coordination. However, one potential problem of this architecture is that the control-plane OS would become a scalability bottleneck. The scalability of the control-plane OS will be determined by the scalability of its services, the performance of interconnect, PCIe, and the performance of IO devices. To see how much the control-plane OS in Solros is scalable, we ran the `fio` benchmark with random read operations varying the numbers of Xeon Phi and NVMe SSD pairs. As Figure 20 shows, Solros scales linearly at least up to four pairs of Xeon Phi and NVMe SSD. For example, in the case that 61 threads per Xeon Phi issue 512KB read requests, Solros shows 2×, 2.9×, and 3.8× higher performance as we increase the number of pairs from two to four, respectively. We note that the number of installable Xeon Phis is limited by the number of PCIe slots.

7 DISCUSSION

Our prototype implementation relies on file system and network stacks of the Linux kernel in a host. Thus, the I/O performance of Solros is bounded by that of the Linux kernel. An alternative implementation choice would be to use highly optimized user-space I/O stacks such as mTCP/DPDK [3, 35] for networking and SPDK [7] for storage.

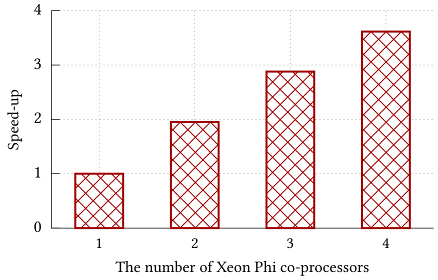


Figure 18: Scalability of our image search varying number of Xeon Phi co-processors. Our shared listening socket mechanism forwards client connection requests to multiple Xeon Phis in a round-robin manner so search queries are evenly distributed to each Xeon Phi. As more Xeon Phis are used, the performance increases linearly.

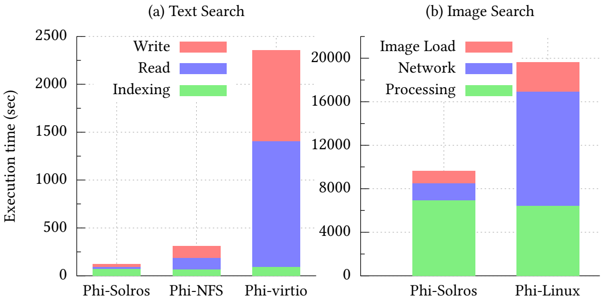


Figure 19: Execution time breakdown of application benchmarks

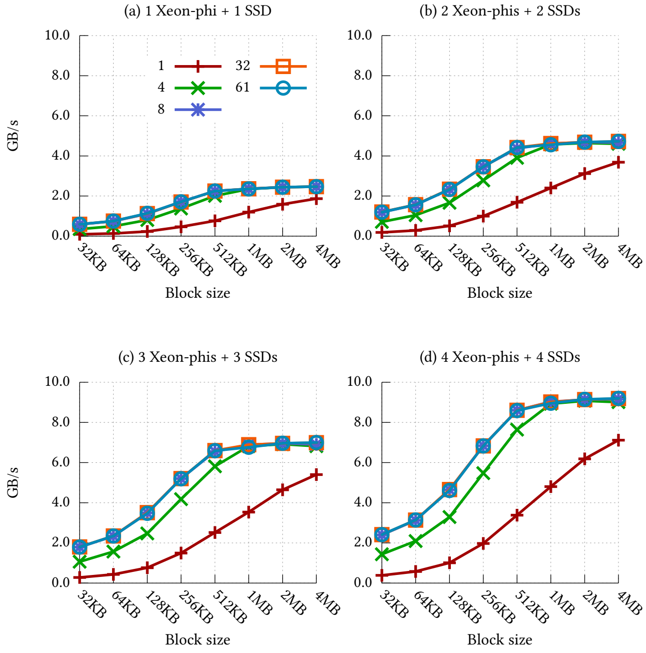


Figure 20: Scalability of `fio` random read operations varying the number of Xeon Phi co-processors and NVMe SSDs. Solros linearly scales at least up to four pairs of Xeon Phi and NVMe.

One interesting future direction is to extend the design of SOLROS to different types of co-processors other than Xeon Phi. The generality of our approach will be decided by required hardware primitives to run the data-plane OS and underlying assumptions of our design. Since all SOLROS services are built on top of its transport service, SOLROS requires atomic instructions (i.e., `atomic_swap` and `compare_and_swap`) used in our ring buffer and the co-processor's capability to expose some of its physical memory to the host to run OS services in the data-plane OS. For isolation among co-processor applications, currently SOLROS relies on MMU of Xeon Phi. It would be interesting to incorporate software-based isolation [30, 51] to SOLROS. We expect that the key performance characteristics that SOLROS exploits will remain in the future. At a high level, SOLROS relies on three performance assumptions: 1) cross-NUMA is expensive; 2) `load`/`store` is good for small data access and DMA is good for large data access because of the high DMA setup cost; and 3) using hardware resources (e.g., a host DMA engine) of faster processors is better for performance.

8 RELATED WORK

SOLROS emerged from a broader trend of heterogeneous computing and enabling fast I/O services in co-processors. This work is inspired by previous research, including OS design for manycore systems [12–15, 22, 24, 53, 59, 66] and heterogeneous systems [11, 51, 57], I/O in virtualization [10, 21, 25, 38, 64], I/O path optimization [8, 50, 56, 65], and I/O support in GPGPUs [37, 58]. The main difference between SOLROS and others is that the control-plane OS running on a host takes active roles to make better system-wide decisions and to exploit asymmetric hardware performance.

OSes for manycore systems. The data-plane and control-plane in OS design were first introduced by Arrakis [53] and IX [15]. In their work, the OS kernel takes the essential control-plane functions such as scheduling and management of data-plane OSes. The kernel involvement on the data-plane operations, such as network process, is minimized. Arrakis and IX rely on a hardware virtualization feature, SR-IOV and VT-x, respectively, for safe and isolated direct access of IO devices. Giceva et al. [24] adopt this approach to design a customized OS for database systems where performance-critical portions run on the customized lightweight, compute-plane OS. The control-plane OS in SOLROS plays an active role to optimize I/O and make better sharing and load balancing decisions for multiple co-processors. In high-performance computing, the mOS (multi-OS) approach [22, 66] runs both a full-weight kernel (FWK) and lightweight kernel (LWK) simultaneously. It reduces OS noise from FWK by running applications on LWK for better performance; it also supports rich system calls by delegating system calls from LWK to FWK. Cerberus [59] and Popcorn Linux [13] replicate OS kernels to multiple processing domains (e.g., a group of cores [59] or a Xeon and Xeon Phi processors [13]) for better scalability or seamless migration of a task. Their main focus is on providing a single system image across multiple processing domains, which is not the goal in SOLROS.

OSes for heterogeneous systems. Helios [51] is an OS for heterogeneous systems with multiple programmable devices. Access to I/O services is done via message passing as with SOLROS. However, Helios requires hardware primitives, timer, interrupt, and trap, which are not required in SOLROS. M3 [11] is a hardware/OS co-design for heterogeneous manycore systems. It integrates cores and memories into a network-on-chip and uses a per-core data transfer unit (DTU) as a message-passing device. OS services are built using message passing via DTU.

I/O path optimization. Hydra [65] optimizes complex layouts of computations among programmable devices to maximize offloading and bus usage. PTask [56] provides OS abstraction for GPGPU to minimize data movement among devices. DCS [8] is a custom hardware engine enabling direct communication among devices (e.g., SSD and NIC). Unlike SOLROS, they do not take special care for the idiosyncratic performance of PCIe and asymmetric processing power in programmable devices.

I/O support in GPGPU. GPUfs [58] provides POSIX-like file system API for GPGPUs, which requests file system calls to a host via RPC. Unlike SOLROS, peer-to-peer data transfer with storage is not considered. GPUnet [37] provides `rsocket` interface, which is a socket-compatible data stream over RDMA/Infiniband, for GPGPUs. In SOLROS, we provide a TCP socket API and focus on supporting multiple co-processors including seamless scale-out using a shared listening socket.

I/O virtualization. Delegating I/O operations to another OS is a recurring design pattern in virtualization to ease the burden of developing device drivers [21, 38, 64] or to specialize I/O operations of JVM for high performance [10]. Since these approaches were developed for homogeneous processor architectures, there is no consideration to exploit heterogeneity.

9 CONCLUSION

This paper describes the design and implementation of SOLROS, a data-centric OS architecture for heterogeneous computing. In SOLROS, OS services, in particular I/O operations, are delegated to a control-plane OS on a host so a data-plane OS on a co-processor is thin and lightweight. The control-plane OS of SOLROS is designed to take an active role to perform global coordination among devices. We found that this separation of I/O operations is the best use of specialized processors. To this end, our file system service judiciously makes a decision whether to perform peer-to-peer communication or host-side buffered I/O. Our network service provides load balancing for the shared listening socket from multiple co-processors using a user-provided rules. This functionality makes the scale out of multiple co-processors easy and seamless. We implemented the SOLROS prototype with Xeon Phi co-processors and NVMe SSDs. Our evaluation results with micro benchmarks and realistic I/O-intensive applications confirm that SOLROS provides significant performance improvement over the conventional design: for example, up to 19× performance improvement for file random read operations and 7× decrease in 99th percentile latency for TCP operations. The source code of SOLROS will be publicly available on Github.

10 ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd, Pramod Bhatotia, for their helpful feedback. This research was supported, in part, by the NSF award DGE-1500084, CNS-1563848, CNS-1704701 and CRI-1629851, ONR under grant N000141512162, DARPA TC

(No. DARPA FA8650-15-C-7556), Institute of Information & Communications Technology Promotion grant funded by the Korea government under contract ETRI MSIP/IITPB[2014-0-00035], and gifts from Facebook, Mozilla, and Intel.

REFERENCES

[1] clucene the open-source, C++ search engine. http://clucene.sourceforge.net/.

[2] Distributed I/O Daemon - a 9P file server. https://github.com/chaos/diod.

[3] DPDK: Data Plane Development Kit. http://dpdk.org/.

[4] Fiemap Ioctl. https://www.kernel.org/doc/Documentation/filesystems/fiemap.txt.

[5] Intel Xeon Phi Coprocessor 7120A. http://ark.intel.com/products/80555/Intel-Xeon-Phi-Coprocessor-7120A-16GB-1238-GHz-61-core.

[6] intro - introduction to the Plan 9 File Protocol, 9P. http://man.cat-v.org/plan_9/5/intro.

[7] Storage Performance Development Kit. http://www.spdk.io/.

[8] Ahn, J., Kwon, D., Kim, Y., Ajdari, M., Lee, J., and Kim, J. DCS: A Fast and Scalable Device-Centric Server Architecture. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Waikiki, Hawaii, Dec. 2015).

[9] Amazon. EC2 F1 Instances (Preview): Run Custom FPGAs in the AWS Cloud. https://aws.amazon.com/ec2/instance-types/f1/.

[10] Ammons, G., Appavoo, J., Butrico, M., Da Silva, D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Van Hensbergen, E., and Wisniewski, R. W. Libra: A Library Operating System for a Jvm in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA, June 2007), pp. 44–54.

[11] Asmussen, N., Völp, M., Nöthen, B., Härtig, H., and Fettweis, G. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA, Apr. 2016), pp. 189–203.

[12] Barbalace, A., Lyerly, R., Jelesnianski, C., Carno, A., ren Chuang, H., and Ravindran, B. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Xi'an, China, Apr. 2017).

[13] Barbalace, A., Sadini, M., Ansary, S., Jelesnianski, C., Ravichandran, A., Kendir, C., Murray, A., and Ravindran, B. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)* (Bordeaux, France, Apr. 2015).

[14] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Dec. 2008), pp. 29–44.

[15] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

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

[17] Burt, J. Intel Begins Shipping Xeon Chips With FPGA Accelerators, 2016. http://www.eweek.com/servers/intel-begins-shipping-xeon-chips-with-fpga-accelerators.html.

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

[19] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th ACM/IEEE International Symposium on Computer Architecture (ISCA)* (San Jose, California, USA, June 2011).

[20] Fatourou, P., and Kallimanis, N. D. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)* (New Orleans, LA, Feb. 2012).

[21] Fraser, K., Hand, S., Neugebauer, R., Pratt, I., Warfield, A., and Williamson, M. Safe hardware access with the Xen virtual machine monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)* (2004).

[22] Gerofi, B., Takagi, M., Ishikawa, Y., Riesen, R., Powers, E., and Wisniewski, R. W. Exploring the design space of combining linux with lightweight kernels for extreme scale computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers* (2015), ACM, p. 5.

[23] Giacomoni, J., Moseley, T., and Vachharajani, M. FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)* (Salt Lake City, UT, Feb. 2008), pp. 43–52.

[24] Giceva, J., Zellweger, G., Alonso, G., and Rosco, T. Customized OS Support for Data-processing. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (2016), DaMoN '16.

[25] Hale, K. C., and Dinda, P. A. Enabling Hybrid Parallel Runtimes Through Kernel and Virtualization Support. In *Proceedings of the 12th International Conference on Virtual Execution Environments* (Atlanta, Georgia, USA, Apr. 2016), pp. 161–175.

[26] Harris, M. CUDA 8 Features Revealed: Pascal, Unified Memory and More, 2016. https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/.

[27] Hemsoth, N. The Next Wave of Deep Learning Architectures, 2016. https://www.nextplatform.com/2016/09/07/next-wave-deep-learning-architectures/.

[28] HP. GenZ interconnect. http://genzconsortium.org/.

[29] HPE. ProLiant m800 Server Cartridge - Overview. http://h20564.www2.hpe.com/hpsc/doc/public/display?docId=emr_na-c04500667&sp4ts.oid=6532018.

[30] Hunt, G. C., and Larus, J. R. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev. 41*, 2 (Apr. 2007), 37–49.

[31] Intel. Intel Omni-Path. http://www.intel.com/content/www/us/en/high-performance-computing-fabrics/omni-path-architecture-fabric-overview.html.

[32] Intel. *Symmetric Communications Interface (SCIF) For Intel Xeon Phi Product Family Users Guide*, 2013. http://registrationcenter.intel.com/irc_nas/4633/scif_userguide.pdf.

[33] Intel. *Intel Xeon Phi Coprocessor System Software Developers Guide*, 2015. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf.

[34] Intel. SSD 750 Series Technical Specifications, 2018. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/750-series/750-1-2tb-aic-20nm.html.

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

[36] Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High Performance Packet Processing with FlexNIC. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA, Apr. 2016).

[37] Kim, S., Huh, S., Zhang, X., Hu, Y., Wated, A., Witchel, E., and Silberstein, M. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014), pp. 201–216.

[38] Kuperman, Y., Moscovici, E., Nider, J., Ladelsky, R., Gordon, A., and Tsafrir, D. Paravirtual Remote I/O. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA, Apr. 2016), pp. 49–65.

[39] Lee, K., and Piantino, S. Facebook to open-source AI hardware design. https://code.facebook.com/posts/1687861518126048/facebook-to-open-source-ai-hardware-design/.

[40] Liu, J., Jiang, W., Wyckoff, P., Panda, D. K., Ashton, D., Buntinas, D., Gropp, W., and Toonen, B. Design and implementation of mpich2 over infiniband with rdma support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* (2004), IEEE, p. 16.

[41] Lozi, J.-P., David, F., Thomas, G., Lawall, J., and Muller, G. Fast and Portable Locking for Multicore Architectures. *ACM Trans. Comput. Syst. 33*, 4 (Jan. 2016), 13:1–13:62.

[42] Mearian, L. Intel unveils its Optane hyperfast memory, 2017. http://www.computerworld.com/article/3154051/data-storage/intel-unveils-its-optane-hyperfast-memory.html.

[43] Mellanox. ConnectX-6 EN 200Gb/s Adapter Card, 2016. https://www.mellanox.com/related-docs/user_manuals/PB_ConnectX-6_EN_Card.pdf.

[44] Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* (1991), 21–65.

[45] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (1996), PODC '96, ACM, pp. 267–275.

[46] Micro, A. X-Gene. https://www.apm.com/products/data-center/x-gene-family/x-gene/.

[47] Min, C., Kashyap, S., Maass, S., Kang, W., and Kim, T. Understanding Manycore Scalability of File Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)* (Denver, CO, June 2016).

[48] Morgan, T. P. Why Google Is Driving Compute Diversity. https://www.nextplatform.com/2017/01/10/google-driving-compute-diversity/.

[49] Mujtaba, H. IntelâĂŹs Lake Crest Chip Aims At The DNN/AI Sector âĂŞ 32 GB HBM2, 1 TB/s Bandwidth, 8 Tb/s Access Speeds,

More Raw Power Than Modern GPUs, 2017. http://wccftech.com/intel-lake-crest-chip-detailed-32-gb-hbm2-1-tb/.

[50] Neuwirth, S., Frey, D., Nuessle, M., and Bruening, U. Scalable communication architecture for network-attached accelerators. In *Proceedings of the 21st IEEE Symposium on High Performance Computer Architecture (HPCA)* (San Francisco, CA, Feb. 2015), IEEE, pp. 627–638.

[51] Nightingale, E. B., Hodson, O., McIlroy, R., Hawblitzel, C., and Hunt, G. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009), pp. 221–234.

[52] Novet, J. Google is bringing custom tensor processing units to its public cloud, 2016. http://venturebeat.com/2016/05/18/google-is-bringing-custom-tensor-processing-units-to-its-public-cloud/.

[53] Peter, S., Li, J., Zhang, I., Ports, D. R., Krishnamurthy, A., Anderson, T., and Roscoe, T. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, Colorado, Oct. 2014).

[54] Putnam, A., Caulfield, A., Chung, E., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture (ISCA)* (Minneapolis, MN, June 2014), pp. 13–24.

[55] Roghanchi, S., Eriksson, J., and Basu, N. Ffwd: Delegation is (Much) Faster Than You Think. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (Shanghai, China, Oct. 2017), pp. 342–358.

[56] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: Operating System Abstractions to Manage GPUs As Compute Devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal, Oct. 2011), pp. 233–248.

[57] Rossbach, C. J., Yu, Y., Currey, J., Martin, J.-P., and Fetterly, D. Dandelion: A Compiler and Runtime for Heterogeneous Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA, Nov. 2013), pp. 49–68.

[58] Silberstein, M., Ford, B., Keidar, I., and Witchel, E. GPUfs: Integrating a File System with GPUs. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 485–498.

[59] Song, X., Chen, H., Chen, R., Wang, Y., and Zang, B. A Case for Scaling Applications to Many-core with OS Clustering. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)* (Salzburg, Austria, Apr. 2011), pp. 61–76.

[60] Tallis, B. Intel Announces SSD DC P3608 Series, 2015. http://www.anandtech.com/show/9646/intel-announces-ssd-dc-p3608-series.

[61] Technologies, M. NPU & Multicore Processors Overview. http://www.mellanox.com/page/npu_multicore_overview.

[62] Tilley, A. Intel Takes Aim At Nvidia (Again) With New AI Chip And Baidu Partnership. http://www.forbes.com/sites/aarontilley/2016/08/17/intel-takes-aim-at-nvidia-again-with-new-ai-chip-and-baidu-partnership/#4417f8364f5a.

[63] Vladimirov, A., Asai, R., and Karpusenko, V. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors.* Colfax International, 2015.

[64] Warfield, A., Hand, S., Fraser, K., and Deegan, T. Facilitating the Development of Soft Devices. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)* (Anaheim, CA, Apr. 2005), pp. 22–22.

[65] Weinsberg, Y., Dolev, D., Anker, T., Ben-Yehuda, M., and Wyckoff, P. Tapping into the Fountain of CPUs: On Operating System Support for Programmable Devices. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Seattle, WA, Mar. 2008), pp. 179–188.

[66] Wisniewski, R. W., Inglett, T., Keppel, P., Murty, R., and Riesen, R. mOS: An architecture for extreme-scale operating systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (2014), ACM, p. 2.