Omnix: an accelerator-centric OS for omni-programmable systems

Mark Silberstein

Technion – Israel Institute of Technology

Future systems will be *omni-programmable*: alongside CPUs, GPUs and FPGAs, they will execute user code near-storage, near-network, near-memory, or on other *Near-X accelerator Units,* **NXUs**. This paper explores the design space of OS support for omni-programmable systems, aiming to simplify the development of efficient applications that span multiple heterogeneous processors and near-data accelerators. *OmniX* is an accelerator-centric OS architecture that extends standard OS abstractions, such as task execution and I/O, into NXUs while maintaining a coherent view of the system among all the processors. OmniX enables NXUs to directly invoke tasks and access I/O services among themselves, excluding the CPU from the performance-critical control plane operations. The host CPU serves as a controller – for protection, device configuration and monitoring. We discuss the hardware trends that motivate our work, outline OmniX design principles, and sketch the core implementation ideas while highlighting missing hardware features, in the hope of motivating hardware vendors to implement them soon.

Introduction

With CMOS scaling officially set to run its course by 2021 [39], future systems will rely on hardware specialization and *near-data processing* to achieve their performance goals. New "smart" peripherals such as smart NICs and smart SSDs are already becoming commercially available [14, 15, 25]. They will soon join the mix of programmable accelerators such as GPUs and FPGAs that are already deployed in systems of all scales and flavors [12, 35]. These new kinds of accelerators, which we call *Near-X accelerator Units, or NXUs*, enable execution of user programs on network (NICs) and storage (SSDs) I/O devices to perform custom processing on the data while it is being transferred or where it is stored. The benefits of near-data processing have long been explored in research, in a wide range of applications ([16, 21, 28, 40, 45] and references therein). It is only now, however, with the stagnation of CPU performance scaling, that these ideas are finally being more widely adopted.We are approaching the era of *omni-programmable systems*, where applications will run on a multitude of CPUs, GPUs and NXUs.

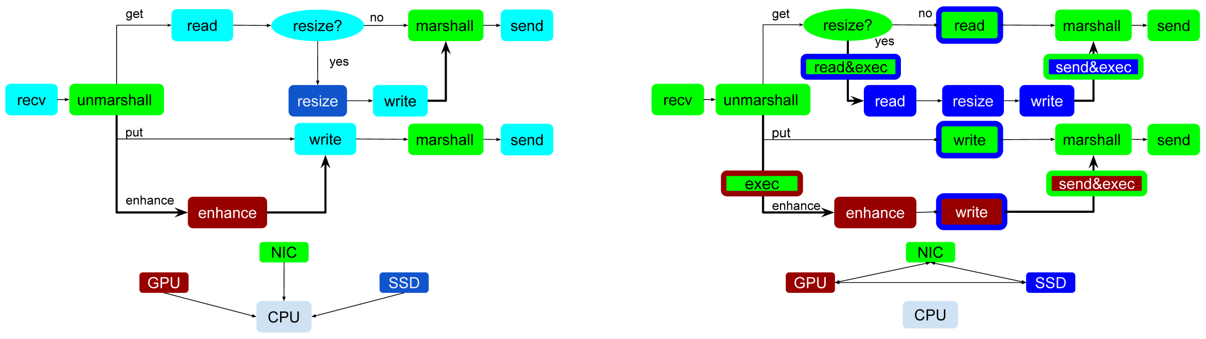
Unfortunately, such heterogeneity of computing resources will pose tremendous challenges to software developers. There is already a zoo of technologies for programming GPUs alone [13, 24, 36], each providing its own set of interfaces and abstractions. To reflect their inherent near-data processing nature and I/O oriented architecture, new NXUs will require an entirely different set of abstractions, as has been the case, for example, for smart NICs [28].

We argue, however, that the challenges will transcend the intricacies of programming individual NXUs. Rather, it will become exceedingly hard to achieve high performance of an application *as a whole* while gluing together shards of optimized code scattered across NXUs and CPUs, and to run multiple such applications securely and efficiently on a single machine. Operating systems have always served as an all-encompassing substrate for building efficient applications from a variety of hardware and software components. A general and systematic approach to OS support for omni-programmable systems is thus the key to realizing their performance potential without heroic development efforts.

To illustrate this point, we sketch a design of an image database server that performs simple image processing functions like resizing and contrast enhancements in real time, while serving retrieving and storing images respectively. Similar systems are reportedly deployed in Flickr [11], for example. The system design is straightforward, with a tweak to keep a cache of already resized images to avoid redundant computations. Figure 1(a) shows a sketch of the main functional blocks and their interactions.

The server is a reasonable candidate for acceleration on NXUs. First, image enhancement operations can be offloaded to a GPU. Second, (un)marshalling of network requests is a good fit for smart NICs: it is a relatively simple, stateless computation that operates on a data stream and reduces the amount of data transferred to the host. Moreover, (un)marshalling operations are worthwhile to accelerate as they form about a third of the so-called *data center tax* [27]. Last, image resizing tasks can be performed by smart SSDs because of their large internal bandwidth to storage [45] and the potential to dramatically reduce storage-to-host bandwidth requirements (consider generating thumbnail from large images).

Unfortunately, it is too hard to implement such an NXU-reach design in CPU-centric systems of today. Smart NICs lack the standard socket abstractions, and expose raw packets instead [15, 28]. Therefore, one would have to implement a network transport layer to process incoming messages. Furthermore, sending the data from the NIC requires access to an ARP table (on the host).



(a) (b)

Figure 1: A functional diagram of an image database server (a) without and (b) with OmniX. (a) The lack of system abstractions requires heavy CPU involvement (b) Distributed execution over multiple NXUs that directly invoke I/O operations without CPU mediation. The shape color and the frame color denote the caller and the callee NXU respectively.

Similar issues arise with smart SSDs, which operate on blocks rather than files. Moreover, the code running on the SSD NXU may observe a stale version of the data, because of accessing it directly in the storage layer. Finally, allowing untrusted application code to access raw packets and storage blocks directly is obviously insecure. These issues disappear if we resort to using the CPU OS for network and file I/O, but then the very benefits of near-data execution on NXUs would become out of reach.

We present *OmniX*, an OS architecture which allows applications such as the image server to achieve their performance potential on omni-programmable systems. OmniX hides the system heterogeneity by providing a set of homogeneous basic OS services and abstractions across all processors. These abstractions focus on accessing files, performing network I/O, and invoking new tasks on the same or other NXUs. Figure 1(b) demonstrates the image server implementation on top of OmniX. Here, the CPU does not run the application logic at all, serving only for initialization and setup, while the NXUs interact directly with each other. For example, the NIC invokes the read and resize operations on the SSD, while the SSD sends the results back to the NIC, which in turn marshalls and sends them back to the client. Figure 2 shows the code sketches for the NIC, SSD and CPU.

OmniX is a single application OS, similar to EbbRT [38] and Unikernel [30]. It comprises several library OSes that provide highly optimized implementation of system services for each NXU. The library OSes interact to maintain a unified and coherent view of the system for applications running on NXUs.

Each library OS implements *private* and *public* OS services. Private services provide OS abstractions accessible to the programs running on the same NXU. For example, a GPU may run GPUfs [43] which enables GPU programs to access files while also providing an on-GPU buffer cache integrated with the buffer caches of other NXUs and CPUs. Public services are accessible to *other* NXUs, and usually match the I/O functionality of the NXU on which they execute, i.e. a file system service running on a smart SSD. NXU programs transparently access remote services via remote procedure calls (RPCs), as in Helios [32]. As a result, OmniX *distributes the control plane across NXUs*, allowing direct NXU-to-NXU communications that bypass the main CPU entirely. This idea resembles that of excluding the OS from the data path [34], but in addition "exterminates the CPU" [1] from inter-NXU interactions.

The OmniX design combines the principles of centralized and distributed systems, following the exokernel [22] model of a single privileged entity and late bindings. It relies on a single coherent shared virtual memory across all NXUs, but has no centralized task scheduler; it uses the CPU to configure and manage all system devices and perform privileged operations, but employs capabilities to allow each NXU to access system resources directly; and it provides shared socket and file descriptors namespace across all NXUs, but does not support cross-NXU task migration.

In the following sections we analyze current hardware trends to understand the expected properties of future omni-programmable systems that will dictate the OS design, describe OmniX design principles, discuss the limitations of existing hardware that preclude building it today, and conclude.

Hardware trends

What architectural support for running OS services will be available in NXUs? While any answer to this question would be highly speculative, we offer some insights we learn from analyzing the last ten years of the General Purpose GPUs (GPGPUs) evolution, as a prominent example of the architecture and software develop-

[1] Paraphrasing the Exokernel "exterminating the OS" [22]

ment path from special-purpose to general-purpose acceleration platform.

GPGPUs emerged as a byproduct of introducing programmable shaders to fixed-function GPUs. Over the years, GPU vendors introduced numerous features that helped improve general purpose programmability, such as support for memory pointers and double precision. However, GPGPU computing owes its success primarily to major architectural improvements in purely graphics-oriented hardware. In fact, most of the new architectural enhancements mainly serve to boost the performance of computer graphics workloads, and improve the GPGPU applications only as a side effect. Moreover, today, vendors are reluctant to improve features essential for GPGPUs but mostly unused for graphics (the poor performance of in-GPU function calls is one notable example). For the same reason, after all these years there is still no architectural support for implementing OS services on GPUs, e.g., on-GPU virtual memory management. Evidently the expectations that GPUs would eventually gain more OS-friendly self-management capabilities, as suggested in prior works on OSes for heterogeneous systems [32], have not materialize so far.

Emerging NXUs seem to be following the same evolutionary path: they originated from high-end fixed-function I/O devices, and their vendors are primarily interested in boosting their performance, rather than in adding proper architectural support for running an OS. For example, Mellanox's Innova smart NIC is essentially a ConnectX-4 Host Channel Adapter (HCA) with a "bump-on-the-wire" FPGA, without any OS support on the device. Similarly, smart SSDs [25, 45] execute application logic on the existing storage controller-resident ARM cores, which do not run an OS. The low-risk, pragmatic approach to NXU development, along with pressing power constraints, force the vendors to focus on purely functional system requirements rather than OS-friendly self-management capabilities.

More generally, to paraphrase the conclusions of the venerable "The wheel of Reincarnation" paper [31]: the functionality that can be implemented efficiently elsewhere should not be implemented on an accelerator. Thus, future NXUs will likely become highly efficient for applications in their respective I/O domains, e.g., string processing on NICs, but will *remain poor candidates for general purpose computations, and will have limited support for running systems software*.

Discrete or on-die? Why not add the necessary I/O processing capabilities to the CPU, and by doing so obviate the need to run applications on I/O devices? This CPU *on-loading* approach [42] is the one used, for example, in recent Xeon-Phi Intel processors, which integrate Omni-Path fabric on-die [44].

While the jury is still out, there are a few reasons to believe that *discrete accelerators will remain relevant in the near future, and in the long run will co-exist with integrated devices*. First, the severe power constraints, which already today lead to the dark silicon effect [46], make it hard to add the large amount of logic necessary for high performance I/O support. This is one of the reasons why on-die devices such as integrated GPUs are much weaker computationally than their discrete counterparts. Second, scaling up a system is more easily done by adding more discrete devices to it. Last, not all the devices can be integrated on-die or even on-package, consider large storage chips, for example.

What kind of interconnect will NXUs have? All major memory, CPU and networking hardware vendors are actively developing new standards for intra-node communications, such as CAPI and CCIX [1, 4, 10]. A key feature of these technologies will be their support for *coherent shared virtual memory and remote memory atomics* across all system processors and NXUs. In particular, memory coherence will allow NXUs to synchronize, to cache each other's data and to communicate without the costly driver-mediated synchronization necessary today, eliminating the CPU from their interaction. Proprietary interconnects with a subset of these properties have already been deployed in production, e.g., NVLINK in IBM Power 9, which connects CPUs and GPUs.

However, despite the global virtual address space, omni-programmable systems will have distinctive *NUMA characteristics*, in particular in terms of memory bandwidth. For example, inter-GPU bandwidth across NVLINK is about 40GB/s, NVIDIA P100 GPU local bandwidth to its memory is up to 750GB/s, and the bandwidth to GPU memory from the NIC (over the PCIe-v3) is 12GB/s. Data locality thus will remain the key optimization goal.

Design

OmniX aims to enable efficient execution of *user applications* on omni-programmable systems. Thus, our primary goal is to achieve high application performance at low development cost.

Design alternatives

Distributed vs. centralized. There are two extremes on the spectrum of possible design approaches.

First is a fully distributed multi-kernel design [17, 32]. Each system processor is treated as an independent self-contained processing unit that runs its own OS kernel. Every NXU on a particular I/O device provides a Remote Procedure Call interface that offers the respective I/O services. For example, a smart NIC may expose services to send/receive packets, a smart SSD may implement a file or block storage server.

The generality and scalability of the multi-kernel approach are appealing. However, to run a full-fledged OS kernel, NXUs must provide architectural support for self-management, such as interrupt handling, privileged execution, and virtual memory management. As we explain in Section 2, we might need to wait a few hardware generations until such support becomes available, if at all.

The second extreme is a centralized CPU-centric design in which the CPU fully controls all the system processors and their interaction. Here, data transfers between NXUs can still be performed in a peer-to-peer manner, but all NXU accesses to OS services are relayed via the CPU OS.

The CPU-centric design is clearly feasible already today. Furthermore, it does not preclude the use of traditional accelerator programming paradigms, thus allowing gradual transition toward the accelerator-centric approach of OmniX. In fact, previous works on OS abstractions for GPUs (GPUfs [43] and GPUnet [29]) follow this design to make their implementation on real hardware possible.

However, the CPU-centric centralized design does have some fundamental limitations.

The primary problem is that the inter-NXU control messages, e.g., network I/O requests from the GPU to the NIC, are relayed via the CPU by a special daemon which actively copies the messages between the NXUs. This daemon turns into a critical element with dramatic influence on the performance of the system as a whole. First, it causes the inter-NXU request latency to vary substantially due to the background CPU noise, which might cause intermittent context switches and daemon eviction. Moreover, the CPU becomes the system bottleneck, as it might be overwhelmed by NXU requests, similarly to the interrupt flooding problem from high-speed I/O devices today. For example, GPUfs [43] requires at least 4 CPU cores per GPU to achieve maximum I/O throughput from the SSD to the GPU for 4K reads [41]. Last, the reliance on the CPU greatly affects the system scaling with respect to the number of GPUs it may support efficiently. For example, an optimized low-latency server that performs K-Nearest-Neighbor search queries on multiple GPUs fails to scale beyond nine GPUs because of the GPU management overheads, even when invoked on a server with 64 CPU cores [47].

The performance is further affected by the increased latency of control messages between NXUs due to the extra PCIe hop and memory copies in the CPU. For example, in GPUnet [29], which relays GPU network I/O requests via the CPU, the end-to-end network latency is more than three times higher than the 5μsecs latency of GPUrdma [20], in which the GPU accesses the NIC directly. As a result, invoking fine-grain I/O and processing tasks among NXUs becomes inefficient and requires batching, thereby complicating the code and increasing the I/O buffer memory consumption to hide the overheads. In the case of GPUnet, the maximum bandwidth of data transfers from the GPU is achieved with network buffers of size 128KB, versus 16KB for GPUrdma.

In OmniX, we combine both distributed and centralized designs while striving to reap the benefits of both.

Alternatives to NXU OS abstractions. Earlier alternative approaches to building multi-accelerator runtime systems [37, 48] focus on higher-level programming abstractions and are less general. For example, PTask [37], works well for static dataflow applications like video streaming, but is less convenient for implementing server applications such as the image server in Figure 1 because of the data-dependent control flow. In fact, OmniX can be used to implement the PTask framework.

Design principles

OmniX is a **single-application OS**, built as a set of library OSes linked with the application and executed in user space. There is one library OS for every processor (including the CPU), which implements a set of common interfaces for accessing system I/O services and invoking NXU tasks.

An application is invoked on the host CPU (see Figure 2(b) for the code sketch). All the NXUs involved in the application get initialized and configured by the standard host OS which runs device drivers, and performs all privileged operations. In addition, the host OS interacts with the library OSes to establish a uniform, coherent view of the system for all processors. For example, all the processors share file descriptors and sockets.

To achieve high efficiency, *OmniX distributes the I/O and task invocation control plane across NXUs*. Specifically, it allows NXUs to invoke operations, and transfer data and control messages in a peer-to-peer manner, without disturbing the CPU. However, the host OS, being the only entity that may execute privileged operations, is involved in the setup and the first access to any resource. The OS generates a capability that can be cached by the library OS on the NXU, allowing access to the resource while bypassing the host OS. The capability may be subsequently revoke at any point. In the next section we discuss a possible approach to supporting capabilities with the help of a virtual memory mechanism on NXUs.

Shared coherent virtual memory. All the processors running application code share a global virtual address space, safely passing memory references among them. However, with highly non-uniform memory access performance, NXUs will naturally cache some of the data they access in remote memories, relying on memory coherence among NXUs and remote memory atomics for synchronizing accesses to shared memory. The host OS is responsible for the address space initialization and management, as well as for handling page faults from NXUs, because all these operations require privileged access to hardware.

Distributed scheduling and task management. Each NXU manages its own set of run queue(s), which are accessible from other NXUs and are also used by the NXU itself. When a run queue runs out of space, the ap-

```
// code snippet for smart NIC
RECV(){
void oninit(...){
 // get access to the image db
 int db=open(dbfile);
 // map the maximum size of the file
 void dbptr*=mmap(db, max_size);
 // a hash table key: imgname value:offset
 Cache cache=initcache();
}
// called when data ready to receive
void onrecv(soc, bufsize){
 recv(sock, indata, bufsize); // indata: NIC  buffer
 Request req=user_parse(indata); // NIC user function
 if (req.type==GET){
  // check if the requested image should be resized
  if ((offset=cache.get(req.name)) == FOUND)
  {// found in cache - respond from NIC
   // enqueue task to itself
   nx_exec(sock,MARSHAL, req, dbptr+offset  );
  }else{
   // pass execution to SSD
   NX_TASK ssd=nx_exec(db,RESIZE,sock,req);
   nx_run_after(ssd,[]{// wait efficienty on promise
     size_t result=nx_result(ssd);
     if (result>0){ // if successful, add to the cache
        cache.add{req.name,result}
     } else nx_exception();
    }
   }
  ....
}
```

(a)

```
// code snippet for smart SSD
RESIZE()
{
....
size_t onexec(fd, soc, req){
 // resize the image and write it to local tmpbuf
 user_resize(req,tmpbuf);
 // write to disk - enqueue to itself
 NX_TASK ssd=nx_exec(fd,FSAPPEND,tmpbuf);
 // send the response in parallel with the disk write
 nx_exec(soc,MARSHAL,reg,tmpbuf);
 // wait for the write
 size_t result=nx_wait(ssd);
 return result;
}

// code snippet for CPU
int main(){
 // load all the kernels
 nx_load(SSD,RESIZE);
 nx_load(NIC,RECV);
 ....
 // install error handlers
 NX_EXCEPTION rep=[]{ printf("error")};
 nx_install_handler(&rep);
 // init the socket
 int soc=accept(...);
 while(1){
   // invoke the server
   nx_wait(nx_exec(soc, RECV));
 }
....
}
```

(b)

Figure 2: Code sketch of the image server implementation for handling get() requests. New OmniX calls are prefixed with 'nx_'

propriate exception is forwarded to the CPU. All NXUs may enqueue new tasks to each other directly without host OS mediation. The enqueued tasks are then invoked asynchronously, while the calling processor may wait for their termination if necessary.

In this model, task scheduling must be performed in a distributed manner. It is not yet clear whether task migration across NXUs can be supported, but the diversity of underlying hardware platforms makes such a cross-architecture migration extremely challenging. On the other hand, task migration across NXUs with the same architecture might be feasible, yet it poses other challenges similar to those encountered in process migration in distributed environments. For example, a task that sends data from the NIC using a specific socket cannot be easily migrated to another NIC.

Protection and Isolation. As a single-application OS, OmniX relies on the hardware virtualization support in NXUs, i.e., SRIOV, to achieve protection and isolation from other applications on the same machine, similarly to EbbRT [38] and Unikernel [30]. This approach might be too strict, requiring that devices be space partitioned across applications [8]. Preemptive task scheduling is possible in theory, but employing one on high speed devices is likely to perform poorly.

OS services on NXUs

Task discovery. NXUs need to discover the tasks that can be invoked on other processors or on themselves. Unlike CPUs that load new binaries for running a program, NXUs effectively invoke pre-loaded routines. This is useful for supporting accelerators that currently lack dynamic binary loading capabilities, e.g., GPUs and FPGAs. In Figure 2, the parameter to `nx_exec` contains the task name as a constant denoted in all caps.

Task execution and scheduling. Each processor may asynchronously invocation a task or an I/O request on any other processor, including itself. I/O operations such as read and write from smart SSD can be also implemented as tasks, invoked on the respective I/O device. The calling task may wait using the invocation handler (NX_TASK handler in Figure 2), and retrieve the return value after termination. While the wait call is supported, it is more efficient to use NX_TASK handler as a *promise* (using the `nx_run_after()` API call) with the continuation as *lambda*. This would make it easier to support I/O preemption for the waiting tasks in order to improve hardware utilization. As we do not require timer interrupts on NXUs, we envision a simple version of hardware-assisted cooperative scheduling implemented using queue pairs discussed in the next section.

Error handling. NXUs have rather limited ability to handle system errors. In OmniX, all critical failures are delivered to the CPU, which may then reset the NXU, or perform some other privileged operations to recover from errors.

Storage/Network-specific OS services. Smart storage and network NXUs must implement special support for internal access to the I/O device they reside on from their own code, while providing the same high-level OS abstractions such as files and sockets. For example, the

NXU on the SSD must be able to access the files stored on that SSD via a simple file interface. Among the important but still unanswered research questions is how to expose higher-level abstractions without reimplementing a significant part of the network and file system stack on the device. We outline one possibility in the next section when discussing in-storage memory mapped files.

Can we build OmniX today?

In this section we discuss the hardware properties essential for OmniX implementation, and how realistic our expectations are to see them in real systems soon.

Peer-to-peer NXU interaction. Peer-to-peer DMA across NXUs is necessary to allow direct data and control message transfers between them.

Feasibility: This feature is already supported in many accelerators today [5], and has been evaluated in earlier works for both data and control messages [7, 18, 20, 29].

Shared virtual memory, coherence and atomics. In OmniX all NXUs share a single application virtual address space and support page faults. Using virtual rather than physical addresses across the NXUs is essential to allow NXUs to run untrusted application code. Moreover, with a single CPU-managed virtual address space one might partially leverage existing PCIe translation mechanisms for managing NXU-resident IOMMUs.

Virtual memory serves the basis for implementing capabilities. For example, a storage device may expose the stored data as a virtual memory region, but later revoke the access without notifying any other NXU holding an active mapping to that region.

Producer-consumer workloads between NXUs require hardware support for memory ordering across the devices to ensure data integrity and avoid inconsistent updates reported earlier in direct GPU-NIC interaction [20]. Remote atomic operations and inter-device memory coherence are necessary for data sharing across NXUs.

Feasibility: Fortunately, shared virtual memory across NXUs is already becoming a reality. It is available in GPUs [9, 19, 26], and will be available soon in NICs [6]. Moreover existing PCIe-v4 standard and emerging chip-to-chip interconnect technologies targeting future accelerators include the functionality to support virtual memory addressing and memory coherence [1, 10].

Queue Pair control interface. Each processor exposes multiple Queue Pair (QP)/Completion Queue (CQ) control structures to allow task invocation from multiple NXUs. A new task enqueued into the QP is asynchronously executed and its completion is reported via a CQ. The QP/CQ data structure is mapped to an arbitrary virtual memory location, but its physical placement is optimized for a particular application, as has been explored in GPUrdma [20]. QP/CQ creation is a privileged operation, and can be used as a capability for granting/revoking access to a particular NXU.

Feasibility: The QP/CQ interface is prevalent in NICs, NVMe SSDs, and beyond [23, 33, 42]. An important addition to the existing QP/CQ implementations is the ability to enqueue requests *by the NXU to itself*, to allow I/O operations to be generated on the device, such as the MARSHAL call in Figure 2. While not available publicly, this idea has been used in NVM over Fabric [2].

In-storage memory mapped files. It is challenging to provide efficient file access *inside smart SSD* without accessing host OS for obtaining block mappings. One way of solving this issue is to require a storage device to expose files as regions of virtual memory (and perform address-to-block translation in firmware). If available, this functionality would allow the SSD to become a DMA slave, thereby making possible unmediated access to files via DMA. We leverage this storage feature to allow the NIC to read data directly from the SSD (see In Figure 2), similarly to the way storage is accessed using DAX [2].

Feasibility: While such a functionality is not available in SSDs today, emerging NVRAM drive technologies do allow accessing storage drive via virtual memory interface, e.g., Microsemi Flashtec [3].

Summary

Omni-programmable systems are becoming a reality. OmniX provides uniform OS abstractions across all system processors, transforms NXUs into peer-processors and reduces programming complexity. By distributing the control plane among the NXUs, OmniX seeks to reap the scaling and performance benefits of a distributed system design, while retaining the convenience of a coherent view of system resources across all processors, with modest NXU hardware requirements.

We take a pragmatic approach to OmniX design by making relatively safe short-term bets about future hardware architectures that we believe are likely to become available soon. Our hope is that this work will encourage NXU designers to extend their architectural support for running systems software in the future. Thus, as NXU hardware acquires more OS-friendly features, our design choices might change, for example, by shifting some core OS functionality to NXUs. We believe, however, that the main principles behind the OmniX design will remain valid regardless of the future changes in the NXU architecture.

Acknowledgements

We would like to thank Boris Grot, Haggai Eran, Amir Watad and the members of the Technion Accelerator Computer Systems Lab for useful discussions and feedback on the early versions of the paper.

References

[1] *Cache Coherent Interconnect for Accelerators (CCIX).*

[2] Mellanox, personal communications

http://www.ccixconsortium.com/.

[2] *DAX: Page cach bypass for filesystems on memory storage*. https://lwn.net/Articles/618064/.

[3] *Flashtec NVRAM Drives*. http://www.microsemi.com/products/storage/flashtec-nvram-drives/flashtec-nvram-drives.

[4] *Gen-Z*. http://genzconsortium.org/.

[5] *GPUDirectRDMA technology*. http://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[6] *Interconnect Your Future with Mellanox 100Gb EDR Interconnects and CAPI*. https://openpowerfoundation.org/blogs/interconnect-your-future-mellanox-100gb-edr-capi-infiniband-and-interconnects/.

[7] MVAPICH2: High Performance MPI over InfiniBand, iWARP and RoCE. http://mvapich.cse.ohio-state.edu.

[8] *NVIDIA GRID Virtual GPU Technology*. http://www.nvidia.com/object/grid-technology.html.

[9] *NVIDIA TESLA P100*. https://www.nvidia.com/object/tesla-p100.html.

[10] *OpenCAPI*. http://opencapi.org/.

[11] A Year Without A Byte. https://code.flickr.net/2017/01/05/a-year-without-a-byte/. Accessed: 2017-01-25.

[12] *AWS announces seven new compute services*, Nov 2016. http://www.businesswire.com/news/home/20161130006132/en/.

[13] CUDA C Programming guide, 2016. Nvidia.

[14] *Mellanox Innova SmartNIC*, Jun 2016. http://www.businesswire.com/news/home/20160615005424/en/.

[15] Netronome Brings Efficient Hardware-Accelerated OpenStack Networking to a Wide Range of Cloud Applications. April 2016.

[16] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro*, 34(4):36–42, July 2014.

[17] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 29–44. ACM, 2009.

[18] S. Bergman, T. Brookman, T. Cohen, and M. Silberstein. SPIN: Seamless Operating System Integration of Peer-to-Peer DMA Between SSDs and GPUs. In *Proceedings of the Seventeenth USENIX Annual Technical Conference*, USENIX ATC '17, page to appear. USENIX, 2017.

[19] P. Boudier and G. Sellers. Memory system on APUs. *AMD fusion developer summit*, 2011.

[20] F. Daoud, A. Watad, and M. Silberstein. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, page 6. ACM, 2016.

[21] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[22] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. *Exokernel: An Operating System Architecture for Application-level Resource Management*. SOSP '95. ACM, New York, NY, USA, 1995.

[23] S. D. Girolamo, P. Jolivet, K. D. Underwood, and T. Hoefler. Exploiting Offload-Enabled Network Interfaces. *IEEE Micro*, 36(4):6–17, July 2016.

[24] K. Group. *OpenCL - the open standard for parallel programming of heterogeneous systems*. http://www.khronos.org/opencl.

[25] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, et al. Biscuit: A framework for near-data processing of big data workloads. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 153–165. IEEE, 2016.

[26] S. Junkins. The Compute Architecture of Intel® Processor Graphics Gen9. *Intel whitepaper v1*, 2014.

[27] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 158–169. IEEE, 2015.

[28] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 67–81, New York, NY, USA, 2016. ACM.

[29] S. Kim, S. Huh, Y. Hu, X. Zhang, E. Witchel, A. Wated, and M. Silberstein. GPUnet: Networking Abstractions for GPU Programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216, Berkeley, CA, USA, 2014. USENIX Association.

[30] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[31] T. Myer and I. E. Sutherland. On the design of display processors. *Communications of the ACM*, 11(6):410–414, 1968.

[32] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS*

22Nd Symposium on Operating Systems Principles, SOSP '09, pages 221–234, New York, NY, USA, 2009. ACM.

[33] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 3–18, New York, NY, USA, 2014. ACM.

[34] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):11, 2016.

[35] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, 2015.

[36] P. Rogers and A. Fellow. Heterogeneous system architecture overview. In *Hot Chips*, volume 25, 2013.

[37] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 233–248, 2011.

[38] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo. EbbRT: a framework for building per-application library operating systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 671–688. USENIX Association, 2016.

[39] Semiconductor Industry Association. *2015 International Technology Roadmap for Semiconductors (ITRS)*, 2015.

[40] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, 2014. USENIX Association.

[41] S. Shahar, S. Bergman, and M. Silberstein. ActivePointers: a case for software address translation on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 596–608. IEEE Press, 2016.

[42] G. Shainer. Offloading vs. Onloading: The Case of CPU Utilization. `https://www.hpcwire.com/2016/06/18/offloading-vs-onloading-case-cpu-utilization/`. Accessed: 2017-01-25.

[43] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: integrating file systems with GPUs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013.

[44] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights landing: Second-generation Intel Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.

[45] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson. Morpheus: creating application objects efficiently for heterogeneous computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 53–65. IEEE, 2016.

[46] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. pages 205–218, 2010.

[47] A. Watad and M. Silberstein. GPUmore: Scalable Multi-GPU Dataset Centric Network Servers. In *Proceedings of the 2017 Workshop on Multi-core and Rack-scale Systems*, MARS'17, 2017.

[48] Y. Weinsberg, D. Dolev, T. Anker, M. Ben-Yehuda, and P. Wyckoff. Tapping into the fountain of CPUs: on operating system support for programmable devices. In *13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Mar. 2008.