The Last CPU

Joel Nider
University of British Columbia
joel@ece.ubc.ca

Alexandra (Sasha) Fedorova
University of British Columbia
sasha@ece.ubc.ca

ABSTRACT

Since the end of Dennard scaling and Moore's Law have been foreseen, specialized hardware has become the focus for continued scaling of application performance. Smart memory, smart disks, and smart NICs are common examples of programmable accelerators that are now being integrated into our systems. Many accelerators can be programmed to process their data autonomously and require little or no intervention during normal operation. Chaining different accelerators together can enable entire applications to be offloaded from the CPU, leaving it only the responsibilities of initialization, coordination and error handling.

We claim that these responsibilities can also be handled in simple hardware other than the CPU and that it is wasteful to use a CPU for these purposes. We explore the role and the structure of the OS in a system that has no CPU and demonstrate that all necessary functionality can be moved to other hardware. We show that almost all of the necessary pieces for such a system design are already available today. The responsibilities of the operating system must be split between self-managing devices and a system bus that handles privileged operations.

ACM Reference Format:
Joel Nider and Alexandra (Sasha) Fedorova. 2021. The Last CPU. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465291

1 INTRODUCTION

In the beginning, CPUs were designed for a single purpose: performing logical operations on stored memory. Co-processors (such as IO channel processors, interrupt controllers, floating point processors), were added to the system to provide functionality that was not provided by the CPU because they could perform certain specialized tasks faster and more efficiently. They enhanced system performance by allowing the CPU to focus on application logic and general system functions, while the simpler co-processors handled mundane tasks.

For high-performance applications, the situation is completely reversed. Programmable devices are responsible for application logic, while the CPU is needed only to support them in these tasks. It is well known that application-specific hardware can perform tasks more efficiently than software running on a general purpose CPU. Accelerators for many applications such as image recognition [42], computer vision [22], key-value stores [14, 29], data warehouses [26], big data [10, 18, 41], deep learning [12], neural networks [23] (and many more) are commonly used to reduce overall system cost and increase performance orders of magnitude beyond the capabilities of a general-purpose instruction set.

In the past, only the most computationally-intensive portions of the program were offloaded to accelerators. More recently, it is becoming common to offload entire applications to accelerators such as SSDs, GPUs and FPGAs that the CPU is needed only for initial setup and error handling [8, 10, 11, 14, 16, 29, 41]. We believe that systems have evolved to the point that the CPU is an appendage that can be completely removed.

At first glance, extending the design of the CPU to include additional accelerator functionality seems like a viable alternative. To improve performance, extensions have already been made to the base instruction set to provide accelerator-like capabilities on general purpose CPUs, such as vector instructions (AVX, ARM Neon, POWER VMX) and encryption [1, 3, 5]. Chiplets are now being used to further increase the density of CPUs and reduce manufacturing costs [31]. These additional functions complicate the verification of the already complicated monolithic CPU and require more silicon area (and possibly more components such as interposers in the case of chiplets), which increases the CPU's base cost and energy consumption even if these functions are never used [17, 25]. In addition, the development cycle to release a new CPU can take many years, meaning that existing systems will not easily be upgraded or changed. The general purpose approach of CPU design has been successful but has started to hit some hard limits. Hardware components that are designed to solve specific problems are becoming prevalent, because they can do the job more efficiently (and

HotOS '21, May 31–June 2, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05…$15.00
https://doi.org/10.1145/3458336.3465291

often much faster) than general purpose CPUs. From the perspective of efficiency, system management is no different: many CPUs are far too powerful and expensive once the critical processing tasks have been offloaded to other hardware.

Accelerator-centric systems with centralized control, such as Omni-X [37], M^{3}X [4] and IX [7], rely on the CPU to handle only the mundane tasks of initialization, coordination and error handling. We believe that decentralized control breaks the dependency on an expensive general-purpose CPU and can improve performance isolation [2]. Control tasks can be boiled down to simple operations that can be handled in other hardware, with the cooperation of the accelerators and programmable devices.

Operating systems provide three key functions: *virtualization*, which includes multiplexing and address translation, *isolation* and *resource management*. We propose that these functions shift from the centralized OS kernel to a decentralized model that consists of self-managed hardware. The missing component is the *system management* bus that is needed for devices to cooperate with one another. It is this bus that performs security-sensitive configuration and is responsible for task life cycle management (initialization, setup, teardown). The introduction of the system management bus as a specialized control plane in combination with self-managing devices for a simpler data plane enables the complete removal of the CPU from the system. The operating system is still the control plane[34] but no longer runs on the CPU.

Our contributions are:

(1) Understanding the role and form of the operating system in a system without a CPU. In particular, we show how cooperation between a system management bus and self-managing devices can replace the functions of a modern OS. We arrive at the specification of functions that the OS must perform in a CPU-less system and *where* this functionality can be implemented.
(2) Enumerating the division of responsibility (i.e. memory isolation, context isolation and resource management) between the system management bus and self-managing devices that enable the implementation of the CPU-less system.

In the rest of the paper, Section 2 outlines the design of system that does not rely on a CPU to coordinate or configure the devices and Section 3 describes a complete end-to-end application example, showing how all the components work together. Section 4 outlines open questions.

2 DESIGN

To break dependence on the CPU, two things must happen:

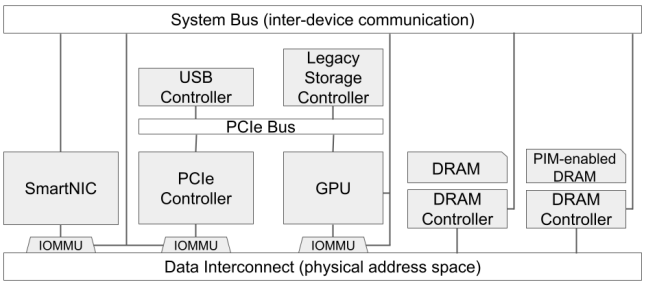

Figure 1: Proposed architecture without a CPU

(1) **Devices must be self-managed** A device must manage its own internal state. It must expose the services it provides, and provide a separate context for each instance of a service (multiplexing) to ensure isolation between applications.
(2) **Devices must communicate autonomously** A device must be able to discover services it needs and request them without relying on an external entity to configure it.

Each of the OS functions - *virtualization* (which includes multiplexing and address translation), *isolation* and *resource management* are essential to create a secure and scalable system. We break up the responsibility for these functions and distribute them among the devices and the bus. The system is thus composed of self-managing devices coordinated by the system bus. The following description explains how the responsibilities of the operating system are divided among the hardware components.

2.1 Self-Managing Devices

A device is responsible for running application logic and offering one or more resources that other devices may use. A device can offer any combination of resource types (physical memory, FPGA blocks, GPU cores, storage space, etc.), exposing each one as a service. Devices can also offer management services for system maintenance. For example, devices that store their applications internally (i.e., on-board flash) must expose a loader service that can be used to upload a new binary image. To be considered self-managing, a device must be able to manage allocation of their resources on behalf of devices in the system and expose them in a standardized way. To do so, each device must implement logic to multiplex its resources into multiple instances, provide isolation between the instances and handle error conditions. This echos the requirements of a *resource monitor* as in the LegoOS splitkernel design [36].

Isolation. Devices will likely support multiple clients or connections to any particular service. For example, a smart SSD that exposes a file system can allow multiple files to be

opened simultaneously by multiple applications. In such a case, it is important that the device implement an isolation mechanism to prevent data leakage between instances. Devices can implement isolation of their resources in hardware or software. Fine-grained resource allocation in hardware has already been implemented in devices such as RDMA controllers and SR-IOV (single root I/O virtualization) NICs in which hardware is partitioned into a fixed number of instances that are exposed and controlled independently. Dynamic isolation of FPGA resources for multiple applications has been described in AmorphOS [27]. Software techniques (such as time sharing) can be used if the device contains an embedded CPU (which is common on devices like smart SSDs). No matter the implementation, the device must be able to expose its functionality in a systematic and standard way that can be consumed easily by other devices wishing to use that functionality.

VIRTIO. VIRTIO is a standardized protocol for managing paravirtual devices from a virtual machine [40]. VIRTIO can provide an ideal interface for exposing resources from self-managing devices. Similar to other protocols, it is based on a set of unidirectional queues of memory descriptors [20, 37]. Hardware vendors are now starting to make real devices that comply with the VIRTIO standard [19, 30]. The main advantage is that many VIRTIO compliant devices (from different hardware vendors) can be operated with a single driver. Exposing all resources and services in a standard and consistent way simplifies the logic needed to use these devices to the point that their services may be consumed by devices with modest hardware. The VIRTIO protocol is able to describe a wide range of devices (more than 20 device types have already been specified) across different levels of abstraction (NICs, disks, consoles, sockets, etc.) that all work in a consistent way.

2.2 System Bus

We propose the use of a new system bus specifically for the purpose of inter-device communication, similar to TMNT [2]. The system bus (as seen in Figure1) acts as the control plane that enables devices to control each other but does not carry data. The system bus only provides a mechanism for device communication and contains no policies. Unlike LegoOS [36] and Barrelfish [6], no entity sees the entire system and there is no global state replication. The bus enables devices to communicate their resource needs in a standard way and enables devices to broadcast their capabilities so that other devices may discover them. This is accomplished by devices sending messages on the bus to request services such as allocation of memory or opening a file. It operates as a privileged device and is the mechanism for maintaining virtualization.

System Initialization. When the system starts, all hardware devices in the system undergo a period of initialization in which they can perform a self-test. When the device determines that it is functioning normally, it will send a message to the system bus, which will record that it is alive. Afterwards, the device will load its applications, of which there can be many. Applications can require one or more services that are provided by other devices. For example, a NIC might need to read data stored in a file on an SSD. Before a device can use a resource, it must first discover which devices in the system can provide access to that resource. The device discovery mechanism is similar to the SSDP (simple service discovery protocol) from the UPnP suite [15], or USB device attachment messages.

Address Translation. An application can be distributed across many devices, but what uniquely identifies it is its virtual address space. As in currently deployed systems, address translation remains the cornerstone of data isolation in shared memory. From a security standpoint, it is not a good idea for a device to be responsible for its own mappings because a compromised device could potentially gain access to resources that it is not authorized for. Therefore, it is the responsibility of the privileged system bus to create virtual-to-physical mappings by updating IOMMU page tables, as instructed by the resource controller (i.e. the memory controller). Similarly, the resource controller cannot be allowed to access the IOMMU of another device directly, since this will lead to security vulnerabilities. Instead, the system bus updates the page tables of a device only when it is instructed to do so by the controller of that particular resource.

Memory management. Virtual memory management is necessary to share memory among different components of the same application, while protecting that memory from other applications. This is largely accomplished through the IOMMU, which gates access to the physical memory from each device, as is commonly done today. When allocating memory, the system bus provides the mechanism for updating virtual to physical mappings, but does not provide the policy. The mappings are set by the memory controller, which manages its own allocation tables internally for each application, similarly to how the *mComponent* – the hardware memory component that is implemented in the LegoOS system [36]. These mappings are sent to the system bus which programs the appropriate page tables for the IOMMU of the requesting device.

Protocol Support. Devices that coordinate via the system bus will be required to adhere to the bus protocol. This is not unlike compliance with existing bus protocols, such as

PCIe. Every device will need to conform to a minimum behaviour to interoperate and share services with other devices. Today, operating systems communicate with devices through controller-specific interfaces such as AHCI (SATA controllers) and EHCI (USB controllers). Devices and controllers already participate in these control protocols with hardware and firmware on the device. That communication would be replaced with higher level protocols to request services directly from a device. The system bus protocol is not expected to be more computationally intensive or to have more complicated logic than many existing control protocols such as those previously mentioned (and may possibly be simpler). Therefore, we expect most devices and controllers that exist today will not require significant changes to their hardware requirements to support the system bus.

2.3 Dataplane

We require two different functions from our interconnects: memory access – *data plane*, and device configuration – *control plane*. We believe these functions should be separate, from a system design and performance perspective. In traditional systems, the CPU is responsible for setting up address spaces during initialization. Since we cannot rely on the CPU, there must be an independent method of addressing devices before virtual address spaces are set up. PCIe partially conflates these two functions by providing both memory access and a certain degree of device configuration through the standard config space and BAR regions by addressing devices by physical address (bus, device, function). Sinces most devices will support multiple virtual address spaces (one per application), they must have the ability to select which virtual address space is in use for each memory operation (like a PASID [33]). The memory bus must have high throughput and low latency, while the system management bus need not. On the other hand, the system management bus must be able to process messages, so it can update the management tables on behalf of applications. While it is not impossible to design a bus that incorporates both functions (high speed memory access and message decoding) we do not see a compelling reason to combine them. There are many existing system interconnects that appear to be good candidates such as PCIe, CCIX [9], GenZ [28], openCAPI [32] and CXL [13].

Notifications. Notifications are a method for a device to signal that it requires some attention. This can be caused by normal operation such as notifying that some requested data is ready. It can also be used to signal an error condition, such as a failed DMA transaction due to an invalid virtual address. Notifications to the CPUs are often sent using interrupts but can be sent over the interconnect as a memory write to a special address. This is similar to the method used to implement MSI (message signalled interrupts) of the PCI standard. Some protocols such as RDMA call this a "doorbell" [24].

Coherency. Cache coherence takes on a different meaning in a system that has no CPU. The purpose of the cache is to improve performance of the CPU by avoiding expensive trips to the main memory. It is convenient to think of a cache as belonging to the memory hierarchy, which obscures the fact that most caches reside in the same physical package (and most often on the same die) as the CPU. Therefore in a system with no CPU, we must carefully reconsider the placement and purpose of cache and cache coherency in the system. Since the cache is private to a device, if a device uses memory only to share data with other devices, caching will not provide much benefit. Devices and applications will certainly continue to use huge amounts of RAM and will benefit from a cache hierarchy in the device (as exemplified by GPUs today). Cache coherency however, is only required in programming models that rely on implicit memory sharing between different processing units. Many distributed systems (such as IX [7] and LegoOS [36]) rely instead on explicit message passing and discard coherency completely. Most of the interconnects mentioned in section 2 support cache coherency messages, but do not require them. In short, each device can choose whether or not to participate in cache coherency of the system, based on its hardware capabilities and the needs of the application.

2.4 What we can build today

Some devices already exist that may be programmed to be used in our system. Certain smart NICs and smart SSDs could be augmented with monitor software relatively easily. What prevents us from removing the CPU completely is that there is no existing hardware component that can act as our system bus. To complete the system, we need a discrete memory controller and interconnect controller that are separate from the CPU package (similar to Intel's Memory Controller Hub[21] or IBM's MXT [39] which not longer exist, as far as we know).

We can emulate the operation of the system bus in software that runs on a CPU. Each device (assuming the devices are really self-managed) would behave as usual – sending and receiving messages from the system bus – but these would be tunneled over shared memory to our emulator. The emulator would still intercept any memory allocation messages and reprogram the IOMMUs accordingly. The emulator would also need to play the role of any resource monitor that cannot yet be embedded in a device: for example, the memory controller. Building an emulated CPU-less system, which is the next step of our research, will permit answering research questions about viability, security and performance of such a system.

3 PUTTING IT ALL TOGETHER

To show a complete example of how the system works, we describe how a hypothetical key-value store application (KVS) would work on a system without a CPU. The data (keys and values) are stored in a file hosted by a smart SSD, while the operations (get, insert, update, etc.) are processed in a smart-NIC. The NIC exposes a KVS interface to other machines over the network by listening on a socket [16, 38] or RDMA connection [29].

Figure 2 shows the initialization sequence, as the KVS application running in the NIC connects to the SSD to access its data file. ① The NIC sends a broadcast message (containing the file name) via the system bus to discover which storage service owns the file. ② The SSD responds that it can offer a service for that file. ③ The NIC sends a request to open the service (including an authorization token) to gain access to the file. ④ The SSD responds with the connection details and the amount of shared memory required. ⑤ The NIC sends a request to the memory controller (including the virtual address), asking it to allocate the shared memory. ⑥ Upon seeing the response from the memory, the system bus programs the IOMMU belonging to the NIC, giving it access to the shared memory at the specified virtual address. ⑦ The NIC sends another message to the system bus to grant access to the shared memory to the SSD. The NIC may then establish the connection by programming the VIRTIO queues in the SSD using virtual addresses.

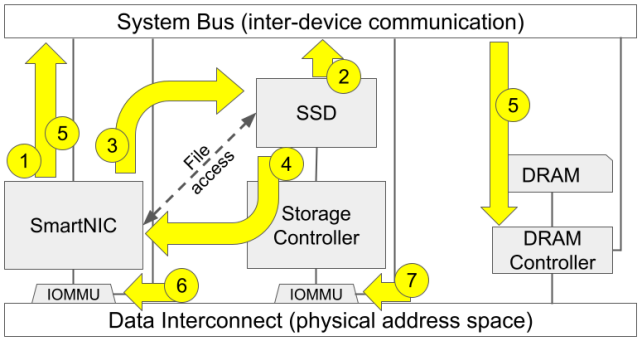

Figure 2: KV-Store application initialization sequence

The IOMMUs protect the memory by translating all memory accesses by the devices to a virtual memory space. To create a shared memory area, the IOMMUs must be programmed to map virtual addresses used by two devices to the same physical addresses. In a system with CPUs, the operating system kernel is responsible for memory management, because it contains a mapping of each processes' address space and is able to perform privileged tasks. An application would invoke a system call (such as *mmap* on Linux) for the kernel to create a shared memory mapping. Without a CPU, the responsibilities are split between the memory controller, which keeps track of physical memory allocations for each device, and the privileged system bus that can update mappings. Instead of a system call, a device sends a control message. Once the action is authorized by the memory controller, the system bus performs the operation. Access to a memory region may be granted by the device that owns the region to another device, but must be first authorized by the memory controller.

4 OPEN QUESTIONS

Access Control. If fine-grained access control is needed, an access control service can be provided by a smart storage controller, such as a smart SSD. This would be roughly equivalent to the 'login' program and 'passwd' file on Linux. Access control to an individual file is implemented by the file system service, on the device that provides that service (likely a smart SSD). A user that wishes to open a file would enter commands through a console app which would use that user's identifier when requesting files from the file system service. Similarly, loader services (to load new microcode, firmware or application code to a device) can also use the same authentication service before replacing sensitive data.

Error Handling. We are not aware of any work on offloading applications to accelerators that addresses how to handle programming bugs. Existing systems have the privilege of relying on the CPU and operating system to handle a variety of unspecified errors. We must be more precise in defining the types of errors that will occur, and how they are to be handled. Recoverable errors are those that do not require a reset of the device. Page faults are caused when the translation hardware (MMU or IOMMU) fails to find a mapping, or if an access is attempted to memory without the correct permissions. In a system with no CPU, the IOMMU would deliver any faults to its attached device. Each device would be responsible to handle its own faults appropriately (i.e. reset the service or stop the application). The failure model is not worse than in a system with a centralized CPU. The major difference is the responsibility of error handling has shifted to the device itself rather than an external entity.

Similarly, if a resource suffers a fatal error but the device survives, the device is responsible for handling the error itself. It must send a message to any consumer using that resource and then reset the resource. It is the responsibility of the application logic running on the consumer to recover from this scenario. If the entire device fails, the resource bus must send messages to all other devices in the system that may be using a resource of the failed device. The bus can also send a reset signal to the failed device in an attempt to restart it.

System Maintenance. We have described in section 3 how a key/value store application would work and assume that such an application can write to a log file during normal operation. A system operator may periodically wish to view these logs to gather statistics or tune some parameters. We did not describe how an operator may view the logs. It is likely that such a machine will be deployed as a server in a data center and will not have a local console. Remote operation would be the best option if many such systems are deployed in a data center, because one remote console can be used to manage many CPU-less systems. The logs could be accessed remotely by another machine over the network through a remote access service. User authentication can be performed by an authentication service running on any device.

Programmability. One of the biggest questions that arises for such an unfamiliar system is how to program it. The main point to keep in mind is that you are writing an application to run on a programmable device, that could use services from one or more other devices. The applications are developed on a machine that has a development environment (i.e. toolchain) for the target hardware. Since each device can have a different instruction set or implementation language, multiple toolchains would be required. In many cases however, the development process would target only one device. To use our KV-store application as an example, all application logic would be compiled to run on the smartNIC. The development environment for the smartNIC would include a library that encapsulates the functionality of the system bus, and provide functions for service discovery, resource allocation, etc. This depends on the other devices in the system (SSD, memory controller, etc.) being able to expose the required resources in an appropriate manner. Developing the app on the target machine without a CPU would require one of the accelerators to run the compiler, which does not have any apparent benefit.

Security. We rely on virtual memory to prevent unauthorized access to the memory of another application. We rely on the implementation of the devices to provide isolation between applications running on the same device. There are certainly other security issues that will arise when designing commonly used services such as a file system. This is non unlike designing a security model for an NFS service, which also exposes an abstraction of a file to a remote device[35].

5 CONCLUSION

Demand for higher performance is pushing system design towards specialized hardware. Individual devices are rapidly developing to manage themselves, relaxing the dependency on the CPU and traditional operating system. While there are many further developments that need to be made, we have already seen huge progress towards self-managing hardware with high levels of abstractions. These higher level interfaces replace traditional software implementations that run on a CPU with more efficient implementations that run on the device.

We have taken an extreme position as a thought experiment. Anything less than contemplating the complete removal of the CPU from a system allows us to fall back to the existing way of viewing systems. Such a drastic change forced us (and hopefully you) to think about system design in a new way and the impact it will have on how we manage such a system for our applications. We, of course, realize that not all systems require accelerators and some problems are just easier to solve on a CPU. However, we now have a new question to ask - what would it look like if we reintroduced a CPU to such a system with self-managing hardware devices? Would it fundamentally change how software is written on a CPU? The CPU is no longer a central component in many of our existing systems and it may not be long before we see systems that are built without a CPU at all. We must consider how such changes will impact system design and what shape operating systems will take in a completely decentralized system.

REFERENCES

[1] K. Akdemir, M. Dixon, W. Feghali, P. Fay, V. Gopal, J. Guilford, E. Ozturk, G. Wolrich, and R. Zohar. Breakthrough AES performance with Intel® AES new instructions. In *Intel Whitepaper*. Intel, 2010. URL https://software.intel.com/sites/default/files/m/d/4/1/d/8/10TB24_Breakthrough_AES_Performance_with_Intel_AES_New_Instructions.final.secure.pdf.

[2] A. Akshintala, V. Miller, D. E. Porter, and C. J. Rossbach. Talk to my neighbors transport: Decentralized data transfer and scheduling among accelerators. *Proceedings of the 9th Workshop on Systems for Multi-core and Heterogeneous Architectures*, 2018. URL https://par.nsf.gov/biblio/10060951.

[3] ARM Inc. Arm cryptocell-300 family. *ARM Developer*, 2020. URL https://developer.arm.com/ip-products/security-ip/cryptocell-300-family.

[4] N. Asmussen, M. Roitzsch, and H. Härtig. M^3x: Autonomous accelerators via context-enabled fast-path communication. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 617–632, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/asmussen.

[5] L. S. Barbosa. Power8 in-core cryptography. *IBM DeveloperWorks*, 2015. URL https://www.ibm.com/developerworks/library/se-power8-in-core-cryptography/index.html.

[6] 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 22nd Symposium on Operating Systems Principles*, SOSP '09, page 29âĂŞ44, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629579. URL https://doi.org/10.1145/1629575.1629579.

[7] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 14), pages 49–65, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay.

[8] A. Caulfield, P. Costa, and M. Ghobadi. Beyond smartnics: Towards a fully programmable cloud: Invited paper. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6, 2018. doi: 10.1109/HPSR.2018.8850757.

[9] CCIX Consortium. CCIX cache coherency interface. *online*, 2019. URL https://www.ccixconsortium.com/.

[10] B. Y. Cho, W. Seob Jeong, D. Oh, and W. W. Ro. Xsd: Accelerating mapreduce by harnessing the gpu inside an ssd. In *WoNDP: 1st Workshop on Near-Data Processing in Conjunction with the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, Davis, California, USA, Dec. 2013.

[11] C. Chung, J. Koo, J. Im, Arvind, and S. Lee. Lightstore: Software-defined network-attached key-value drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 939âĂŞ953, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304022. URL https://doi.org/10.1145/3297858.3304022.

[12] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. Serving dnns in real time at datacenter scale with project brainwave. *IEEE Micro*, 38(2):8–20, 2018. doi: 10.1109/MM.2018.022071131.

[13] CXL Consortium. Compute express link™: The breakthrough cpu-to-device interconnect. *online*, 2020. URL https://www.computeexpresslink.org/.

[14] A. De, M. Gokhale, R. Gupta, and S. Swanson. Minerva: Accelerating data analysis in next-generation ssds. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16, 2013. doi: 10.1109/FCCM.2013.46.

[15] A. Donoho, B. Roe, M. Bodlaender, J. Gildred, A. Messer, Y. Kim, B. Fairman, and J. Tourzan. Upnp device architecture 2.0. *online*, 2020.

[16] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. Nica: An infrastructure for inline acceleration of network applications. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 345âĂŞ361, USA, 2019. USENIX Association. ISBN 9781939133038.

[17] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 39(3):365âĂŞ376, June 2011. ISSN 0163-5964. doi: 10.1145/2024723.2000108. URL https://doi.org/10.1145/2024723.2000108.

[18] W. Fang, B. He, Q. Luo, and N. K. Govindaraju. Mars: Accelerating mapreduce with graphics processors. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):608–620, 2011. doi: 10.1109/TPDS.2010.158.

[19] S. Hajnoczi and M. Tsirkin. Virtio without the "virt". *online*, Nov. 2019. URL https://lwn.net/Articles/805235/.

[20] T. J. Ham, J. L. AragÃşn, and M. Martonosi. Desc: Decoupled supply-compute communication management for heterogeneous architectures. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 191–203, 2015. doi: 10.1145/2830772.2830800.

[21] Intel®. 5100 memory controller hub. *online*, 2009. URL https://www.intel.com/content/dam/doc/datasheet/5100-memory-controller-hub-chipset-datasheet.pdf.

[22] S. Jiang, D. He, C. Yang, C. Xu, G. Luo, Y. Chen, Y. Liu, and J. Jiang. Accelerating mobile applications at the network edge with software-programmable fpgas. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 55–62, 2018. doi: 10.1109/INFOCOM.2018.8485850.

[23] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1âĂŞ12, June 2017. ISSN 0163-5964. doi: 10.1145/3140659.3080246. URL https://doi.org/10.1145/3140659.3080246.

[24] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 437–450, Denver, CO, June 2016. USENIX Association. ISBN 978-1-931971-30-0. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia.

[25] A. Kannan, N. E. Jerger, and G. H. Loh. Enabling interposer-based disintegration of multi-core processors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 546âĂŞ558, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340342. doi: 10.1145/2830772.2830808. URL https://doi.org/10.1145/2830772.2830808.

[26] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42âĂŞ52, Sept. 1998. ISSN 0163-5808. doi: 10.1145/290593.290602. URL https://doi.org/10.1145/290593.290602.

[27] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach. Sharing, protection, and compatibility for reconfigurable fabric with amorphos. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 107âĂŞ127, USA, 2018. USENIX Association. ISBN 9781931971478.

[28] M. Krause and M. Witkowski. Gen-Z DRAM and persistent memory theory of operation. *online*, 2019. URL https://genzconsortium.org/.

[29] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 137âĂŞ152, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132756. URL https://doi.org/10.1145/3132747.3132756.

[30] Mellanox Inc. Bluefield dpu sw manual v3.1.0.11424. *online*, 2020. URL https://docs.mellanox.com/display/BlueFieldSWv31011424/VirtIO-net%20Emulated%20Devices.

[31] S. K. Moore. Chiplets are the future of processors: Three advances boost performance, cut costs, and save power. *IEEE Spectrum*, 57(5):11–12, 2020. doi: 10.1109/MSPEC.2020.9078405.

[32] OpenCAPI consortium. OpenCAPI consortium. *online*, 2019. URL https://opencapi.org/.

[33] PCI-SIG. Process address space id (pasid). *online*, 2011. URL https://members.pcisig.com/wg/PCI-SIG/document/12366.

[34] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter.

[35] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. *Design and Implementation of the Sun Network Filesystem*, page 379âĂŞ390. Artech House, Inc., USA, 1988. ISBN 0890063370.

[36] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, Carlsbad, CA, Oct. 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/shan.

[37] M. Silberstein. Omnix: An accelerator-centric os for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 69âĂŞ75, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350686. doi: 10.1145/3102980.3102992. URL https://doi.org/10.1145/3102980.3102992.

[38] G. Siracusano and R. Bifulco. Is it a smartnic or a key-value store? both! In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos '17, page 138âĂŞ140, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350570. doi: 10.1145/3123878.3132014. URL https://doi.org/10.1145/3123878.3132014.

[39] R. B. Tremaine, T. B. Smith, M. Wazlowski, D. Har, K.-K. Mak, and S. Arramreddy. Pinnacle: Ibm mxt in a memory controller chip. *IEEE Micro*, 21(2):56âĂŞ68, Mar. 2001. ISSN 0272-1732. doi: 10.1109/40.918003. URL https://doi.org/10.1109/40.918003.

[40] M. Tsirkin and C. Huck. Virtual i/o device (virtio) version 1.1. *online*, 2019. URL https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html.

[41] M. Yoshimi, R. Kudo, Y. Oge, Y. Terada, H. Irie, and T. Yoshinaga. An fpga-based tightly coupled accelerator for data-intensive applications. In *2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs*, pages 289–296, 2014. doi: 10.1109/MCSoC.2014.47.

[42] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 161âĂŞ170, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333153. doi: 10.1145/2684746.2689060. URL https://doi.org/10.1145/2684746.2689060.