Beyond processor-centric operating systems

Paolo Faraboschi, Kimberly Keeton, Tim Marsland and Dejan Milojicic
Hewlett-Packard Labs

Abstract

By the end of the decade, computing designs will shift from a processor-centric architecture to a memory-centric architecture. At rack scale, we can expect a large pool of non-volatile memory (NVM) that will be accessed by heterogeneous and decentralized compute resources [3, 17]. Such memory-centric architectures will present challenges that today's processor-centric OSes may not be able to address. In this paper, we describe the characteristics and consequences of memory-centric architectures and propose a memory-centric OS design that moves traditional OS functionality outside of the compute node and closer to memory.

1 Introduction

Data growth is outpacing the compute and storage technologies that have been at the foundation of the IT industry for the last four decades [35]. This divergence causes a deep rethinking of the way in which we build systems, and points towards a much more radical *memory-centric architecture*, where memory is the key resource and everything else, including processing, revolves around it. In effect, *the memory is the computer*.

By the year 2020, we expect memory-centric architectures at the rack scale that implement a *shared something* model, a middle ground between shared everything scale-up systems and shared nothing scale-out systems [3, 17]. In this architecture, the traditional memory hierarchy collapses, replacing traditional block-based storage (hard drives and SSDs) and traditional DRAM with byte-addressable non-volatile memory (NVM) (sometimes called *universal memory*). As memory and storage converge, we expect that the NVM pool will also shift from being directly attached to the main application CPU to becoming a shared resource that is accessible by all compute resources in the rack. Advances in optical networking will make this pooled NVM accessible as memory at almost uniform latency, and manageable as storage for reliability, serviceability and security. In addition to shared NVM, "private" memory, probably volatile and co-packaged with the computing elements, will also provide a lower latency and higher bandwidth "performance tier". Finally, compute resources will become increasingly heterogeneous, and distributed closer to the data.

These memory-centric architectures will present challenges that today's processor-centric OSes may not be able to address, as well as opportunities that they aren't prepared to exploit. Instead, we need a new breed of *memory-centric OS*, which moves traditional OS functionality outside of the compute node and closer to memory, in memory-side accelerators and controllers as well as system-wide services.

In this paper, we will describe future memory-centric architectures and the consequences they have for operating system design. We will also articulate a vision for what future memory-centric operating system services are needed and outline open research challenges towards realizing this vision.

2 Memory-centric architectures

As data becomes the new currency, architectural changes are necessary to overcome the limitations of the traditional compute-centric model. Data-centric designs call for an architectural approach that minimizes data duplication and redundant motion, enables ubiquitous and heterogeneous computing resources, and deals with resilience and security from the ground up.

We are already seeing signs that the traditional monolithic server is disaggregating into a rack-scale architecture [10, 22], where pools of storage, networking and compute resources can be flexibly organized in a software-defined manner to match different workload characteristics and requirements. This is the first step towards an even more radical memory-centric approach.

Take for example HP's Moonshot platform, and the m800 server cartridge [1]. Built around TI's Keystone II SoC, it integrates 4 general purpose ARM cores and 8 accelerated VLIW DSP cores in the same chip. This platform can pack over 20,000 cores, 14 TB DRAM and 100 TB Flash into a single 47" rack.

If we project this trend out to the end of the decade, we can easily expect O(100,000) cores, O(100TB) DRAM, and O(1-10PB) of NVM in a hypothetical 2020 rack. When we deal with this scale of resources, we believe that other fundamental changes will also occur.

Persistent storage will be accessible through the standard load/store path, rather than through the indirect block-oriented I/O path that we use today. As new NVM memory technologies (such as memristor [34] or phase change memory [38]) mature, they will approach the access latencies of DRAM and become first-class citizens in the memory hierarchy. Data-centric applications operating on ever larger (and more sparse) data sets will require increasingly more performance on random access patterns that do not work well with block-oriented I/O. As observed by Bailey, et al. [4], this implies a significant departure from the block-oriented APIs that the OS and other middleware layers between applications and storage devices use to access non-volatile resources.

NVM will become a rack-scale pooled resource. Rather than being directly attached to the CPU, it will become accessible to all of the compute resources in the rack. At the scale of 100,000 cores it is impractical to expect that all cores will have fully cache coherent access to memory [3], and research efforts are already investigating scale-out NUMA extensions that overcome the scalability limitations of single cache-coherent servers [25]. At the PB of memory scale, it will have to be managed to provide the resilience, serviceability and availability properties that storage systems offer today. So, we will inevitably move towards a clustered, *shared something* architecture, which represents a middle ground between pure shared nothing scale out and pure shared everything scale up. Figure 1 illustrates the difference between these models. In a shared nothing architecture, all memory resources are directly connected to one (and only one) compute node and accessing remote memory requires RDMA-style mediated support. In contrast, shared something memory resources are accessible by all compute nodes directly, without the intervention of another node; this model is analogous to the shared disk model of yesterday [14]. Rack-level pooled memory reopens a variety of research questions on non-coherent, software-managed sharing. While we expect the direct-attached shared nothing style to prosper for a while, eventually the benefits of sharing resources at the rack scale will cause the warehouse-scale computing building block to evolve from a single server to a rack. We are already seeing indications of this trend in recent work, where acceleration resources are pooled at the enclosure level and shared by several servers [29].

Optical networking will make most of the network-accessible NVM equidistant. Once one pays the cost to get to the network, high-radix optical switches will enable a low-hop-count topology (such as HyperX [2]) that will make all NVM appear at the approximately the same distance, for all practical purposes. This topology does not mean that locality is not important: to the contrary, we expect node-local memory resources (either stacked, co-packaged, or in close proximity with the computing elements) to be included, representing a far lower latency and higher bandwidth "performance tier" that applications and system software will have to take account of.

Compute resources will become heterogeneous and decentralized, due to the *dark silicon* effect that will eventually become visible [11]. Accelerated functionality will migrate from the main application CPU to everywhere it is required. This trend implies computation much closer to data [30], unless it is strictly required to be centralized. In addition to application offload functionality, we also expect memory to become more "intelligent," including taking an active role in protection, allocation, synchronization, and resilience.

3 Towards a memory-centric OS

Memory-centric architectures lead to a large set of challenges that today's processor-centric OSes may not be able to meet, as well as opportunities that they aren't poised to exploit. In this section, we describe these consequences and the requirements they place on the OS in more detail. Additionally, we describe how shifting OS design from a processor-centric focus to a memory-centric one may help to address these challenges.

3.1 Shared pool of memory

In traditional cache-coherent NUMA machines, memory is "owned" by a particular compute node, which performs allocation and mediates access to that memory segment. Applying this traditional ownership model to the pooled memory in the shared something architecture makes little sense, because the owning compute node and the memory may fail independently. The failure of the owning node would mean that the memory it manages would no longer be accessible, even if the memory itself continued to be operational.

In a world of memory-centric, shared something clusters, we anticipate that traditional memory management functionality will move from the processor-centric node OS into memory side controllers, accelerators and more novel computational elements to form (distributed)

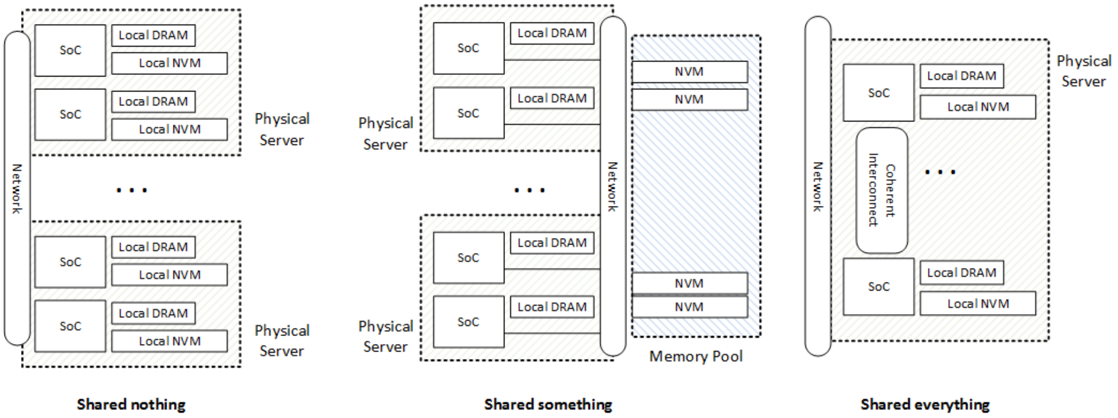

Figure 1: Comparison of *shared nothing*, *shared something* and *shared everything* configurations with NVM

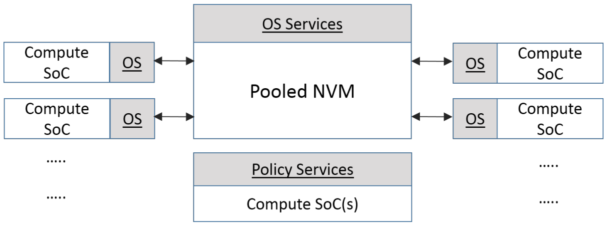

Figure 2: Migration of traditional OS functionality out of single SoC. Functionality, such as NVM management, protection, synchronization, de/encryption, (de)compression, and error handling, will move from the processor-centric SoC node OS into memory side controllers, accelerators and more novel computational elements. Similarly, policy services such as quotas and QoS will be externalized into cluster-wide services.

cluster-wide services. Figure 2 illustrates this migration of traditional OS functionality. To be more specific, we explore a few mechanisms and how they are implemented in a memory-centric, shared something system.

Memory resource management. Functions such as allocation and deallocation of global memory resources must be managed cluster-wide by a combination of memory-side controllers. The application runtime present on each collection of processors (in coordination with the privileged entity that manages memory protection) requests a chunk of universal memory from the memory services, and the OS maps it into the application address space. Some runtimes may wish to manage those resources locally to hand off to other applications. Because the sheer quantity of NVM is potentially very large (petabytes) and it is likely that the distributed services will be implicitly slower, allocation chunks will typically be large. We expect a hierarchical structure where purely local entities may take responsibility for finer-grained allocations. Memory resource management may also need to be aware of the bandwidth and latency attributes of the memory system, as well as the intended use of the memory (e.g., a buffer for inter-node communication vs. a processor-local scratchpad). Lessons learned from earlier research on multi-kernel shared memory systems (e.g., [23]) may be applicable.

Protection and translation. Similarly to resource management, cluster-wide data protection should be enforced locally by the memory elements themselves, in cooperation with the processor-managed virtual memory system. Because we expect a byte-addressable universal memory, the basic unit of access and protection is a cache line. However, OSes today would consider any protection (load) fault at this level to be a fatal memory error! We think that it is likely that such systems will continue to want to use virtual memory to support simple application environments and large (virtually) contiguous memory regions. As the scale of the system increases, we also believe it is important to decouple the concerns of translation and protection. In this context, we want translation to be as efficient as possible via very large pages or even direct mapping (e.g., [5]). As we move to a world where all persistent data lives in memory, we tend to want very fine-grain protection for certain items (e.g., the metadata associated with an object). While many of today's OSes support multiple page sizes and work hard to reduce fragmentation, there is no "goldilocks" page size. Since the goals of efficient translation of large quantities of memory and fine-grain protection are in conflict, we believe that alternative protection mechanisms should be carefully explored (e.g., [36, 37]).

Tradeoffs exist between how much can be accomplished at the hardware ISA level (e.g., [37]) vs. at the software level (e.g., [7]) in terms of generality, performance, and ease of use (e.g., whether recompilation is required). Modern OSes expose considerable functionality to the user space, making it easier to optimize and hide some of the complexities in libraries.

Interconnect-related memory error handling. The application abstraction of a flat virtual address space presents a semantic gap between the apparent uniformity of that space and the fault boundaries at the physical address discontinuities due to the underlying distributed nature of the memory system. In an environment where load/store instructions are effectively mapped by the hardware to network packets that traverse a large memory system, we can encounter new species of memory errors corresponding to possibly transient errors in the communication to memory. Handling these errors will be more subtle, particularly for handling cache eviction due to displacement flushes. For example, rather than treating all errors as fatal (i.e., causing the system to reset), miss-related errors that occur synchronously should probably be treated as transient communication failures to be handled by the application. We believe it will be useful to revisit existing work (e.g., Linux "machine check" recovery [21]) for the rack-scale context.

Reliable Services. Although shared something systems may no longer need replication for performance reasons, a sophisticated memory-centric system may support various forms of redundancy as natural primitives to help improve reliability. A related aspect of building a (more) reliable system or application service on top of a less reliable, distributed infrastructure is the equivalent of cluster membership (i.e., what hardware computational elements are functioning, and what mitigations can be deployed recover from failure?). Detecting errors and managing recovery in a shared something system are likely to be more complex than the equivalent facilities in a shared nothing system [18].

3.2 Memory at large scale

Addressing memory. Even if we assume an uncomplicated physical address space, the sheer size of the memory system of a warehouse-scale shared something cluster may be far larger than any processor thread can address due to the shortage of physical or virtual address bits. We can expect processors to gain address bits over time, but in the meantime we may require additional layers of physical address translation (e.g., [15, 16]) or more sophisticated approaches (e.g., [13]).

We also believe that the various physical and logical address spaces managed by the operating system will become far more dynamic — to deal with the large capacity, failures, or reconfigurations of the memory elements that are carrying load/store traffic. Ultimately this requires memory side accelerators and other components in the memory system to coordinate in transparently managing all levels of the address space of the system as the cluster and the needs of a running application evolve.

Coherency. It is easy for accesses to memory from a small number of CPU cores to be cache-coherent, even for quite large numbers of hardware threads. But ultimately hardware coherence mechanisms do not scale, and therefore we must either abandon cacheability, or carefully manage concurrent access to a given cache line between different groups of cores that are using hardware coherency. As a trivial example, it may be perfectly acceptable for multiple coherence domains to cache a very large, read-mostly data structure, while relying on a heavier weight distributed synchronization mechanism when the data structure needs to be updated.

Synchronization. More generally, multiple hardware threads from different coherence domains need to be able to coordinate access to universal memory as they take part in a computation. This can be provided by memory-atomic synchronization mechanisms directly supported by the memory system that perform cluster-wide equivalents of processor atomics like `ll/sc` or `cas`. Traditional lock-based and lock-free algorithms can then be built on top of these mechanisms with minimal software changes. An additional challenge is to arrange the equivalent of inter-processor messages (c.f. interrupts) needed to implement adaptive locks vs. long-duration spins in the face of lock contention.

Big memory operations. As memory pools increase in size, it will become increasingly time-consuming to perform an operation across all of memory (e.g., zeroing memory, verifying checksums, scanning all data in a single-pass algorithm). Memory-side acceleration may provide assistance for low-level operations (e.g., [27]); increasing accelerator programmability permits more sophisticated operations to be moved closer to the data [20]. This raises interesting questions about software-based control of when and how to use such accelerators [31]. At one extreme, low-level generic functions, such as encryption or compression, may be implemented transparently to the OS and applications; at the other extreme, semantic memory operations, which require understanding of the data (e.g., copying application data structures or semantic checkpointing [19]) require considerable software intervention.

Memory errors at scale. As the size of memory increases, memory errors will not be an exception but common behavior. Traditional mitigation techniques, such as parity, chip kill [9] and memory scrubbing, can detect or correct these problems transparently to the compute nodes [39]. However, because of the load/store access to NVM, the OS and user applications running on the compute nodes must be prepared for the fact that the memory they access may fail. In the past, memory failures encountered by an OS were considered unrecoverable and would typically result in a machine check, while user processes would be killed. Given the high expected rate of large memory failures, we advocate the use of exceptions for error reporting, to permit the OS and applications to recover where possible [24].

Failures need to be contained as much as possible, and the affected resources must be recovered after the failure. In contrast to shared everything and shared nothing systems, where fault domain boundaries (and hence fault containment strategies) are clearly defined, shared something memory errors cross fault domains, and potentially affect multiple compute nodes running independent OS instances. Because multiple nodes may map the same memory regions, when a shared something node fails, additional error containment must be performed (e.g., ”poisoning” memory), so that nodes that subsequently access the shared memory deal with the failure.

3.3 Memory non-volatility

As described by Bailey, et al., the non-volatility of memory presents additional challenges [4].

Abstraction for persistent data. NVM blurs the line between file system and memory access for persistent data (e.g., files vs. memory regions), calling into question whether traditional file systems are the best approach for managing persistent data.

Volatile caches. Some components of the memory system architecture (e.g., store buffers, caches) will likely continue to be volatile, even as main memory becomes fully non-volatile. The OS and applications must carefully control movement of data from volatile processor caches to non-volatile memory. ISA support is only beginning to become available (e.g., Intel's `pcommit` and `clwb`); without sufficient ISA support, the OS and application runtime systems may need to rely on software-controlled mechanisms to manage this data movement in a sensible fashion (e.g., [8, 26]).

Recovery from software and human errors. Because NVM doesn't "forget" its state, it's no longer possible to clear memory state by rebooting the machine (and thus clearing volatile DRAM and processor cache state). The memory-centric OS must provide additional mechanisms such as checkpoints and logging to permit recovery from software and human errors. Since we'll likely want to keep many such recovery points around, the OS must decide how to create and garbage collect these recovery points to provide a reasonable tradeoff between fast (and recent) recovery behavior vs. the cost of generating the recovery points [33].

Encryption is a necessary but insufficient protection mechanism to data held in universal memory [3]. Data at rest can be protected against physical theft of individual devices by keeping the data in the physical media appropriately encrypted, while ensuring that the relevant keys are volatile with respect to the device, but non-volatile with respect to the system. Ideally, data in motion through the memory system should also be encrypted. As usual, key management may present some tricky problems. Note that encryption at rest represents a form of read protection for the underlying data, but does not prevent the data from being overwritten or corrupted.

3.4 Additional issues

Memory-centric I/O. In a system with universal memory, the majority of I/O comes from data networks. A simple model for I/O is to imagine a sophisticated I/O engine in the memory system that handles moving data to/from memory from/to an IP network. A more challenging problem is dealing with very high-speed networks where packet processing by an application needs to happen with very low latency, particularly when the desired performance begins to challenge the latencies of the memory interconnect plus the characteristics of the memory device itself. High speed operations may imply that even cache fill times across a large-scale memory system are too long. In such cases the data must be delivered much closer to the processor (e.g., directly into a cache or locally attached memory). The role of the OS is largely to get out of the way of data plane operations, instead providing control plane functionality such as process isolation and security (e.g., [6, 28, 32]).

Memory-centric application runtimes. Memory-centric systems invite the further exploration of distributed application runtimes that make effective use of distributed persistent memory, which will in turn generate corresponding requirements on a memory-centric OS. For example, earlier work, such as Linda and tuple spaces, and various dataflow programming models should be reevaluated in this context [12].

4 Summary

The future is converging towards a memory-centric rack-scale architecture that adopts a shared something model, where heterogeneous compute resources access a large pool of shared non-volatile memory. Today's processor-centric OSes may fall short of addressing the challenges these architectures present and exploiting the opportunities they provide. In this paper we've articulated a vision for a memory-centric OS that moves traditional OS functionality like memory allocation, protection, synchronization and error handling out of the compute node and closer to the memory system.

Acknowledgments. We thank the anonymous reviewers and our colleagues John Sontag, Brad Morrey, Indrajit Roy, Terence Kelly, Joe Tucek, Keith Packard and Guilherme Magalhaes for their helpful comments.

References

[1] HP ProLiant m800 Server Cartridge (web site). http://bit.ly/moonshot-m800, 2014.

[2] AHN, J. H., BINKERT, N. L., DAVIS, A., MCLAREN, M., AND SCHREIBER, R. S. HyperX: Topology, routing, and packaging of efficient large-scale networks. In *SC* (2009), ACM.

[3] ASANOVIĆ, K. A hardware building block for 2020 warehouse-scale computers. In *FAST 14 keynote* (2014), USENIX.

[4] BAILEY, K., CEZE, L., GRIBBLE, S. D., AND LEVY, H. M. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (2011), HotOS'11.

[5] BASU, A., GANDHI, J., CHANG, J., HILL, M. D., AND SWIFT, M. M. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ISCA '13, pp. 237–248.

[6] 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 Conference on Operating Systems Design and Implementation* (2014), OSDI'14, pp. 49–65.

[7] CASTRO, M., COSTA, M., MARTIN, J.-P., PEINADO, M., AKRITIDIS, P., DONNELLY, A., BARHAM, P., AND BLACK, R. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09, pp. 45–58.

[8] CHAKRABARTI, D. R., BOEHM, H.-J., AND BHANDARI, K. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (2014), OOPSLA '14, pp. 433–452.

[9] CHEN, C.-L., AND HSIAO, M. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development 28*, 2 (1984), 124–134.

[10] COSTA, P. Towards rack-scale computing: Challenges and opportunities. In *First International Workshop on Rack-scale Computing* (2014), ACM.

[11] ESMAEILZADEH, H., BLEM, E. R., AMANT, R. S., SANKARALINGAM, K., AND BURGER, D. Dark silicon and the end of multicore scaling. *IEEE Micro 32*, 3 (2012), 122–134.

[12] GELERNTER, D. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan. 1985), 80–112.

[13] GERBER, S., ZELLWEGER, G., ACHERMANN, R., KOURTIS, K., ROSCOE, T., AND MILOJICIC, D. Not your parents' physical address space. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (2015), HotOS'15.

[14] GRAY, J., AND REUTER, A. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[15] HOWARD, J. Rock Creek system address look up table and configuration registers. External Architecture Specification (EAS) Revision 0.1, Intel Microprocessor Technology Laboratories, January 2010.

[16] HOWARD, J., DIGHE, S., HOSKOTE, Y., VANGAL, S., FINAN, D., RUHL, G., JENKINS, D., WILSON, H., BORKAR, N., SCHROM, G., PAILET, F., JAIN, S., JACOB, T., YADA, S., MARELLA, S., SALIHUNDAM, P., ERRAGUNTLA, V., KONOW, M., RIEPEN, M., DROEGE, G., LINDEMANN, J., GRIES, M., APEL, T., HENRISS, K., LUND-LARSEN, T., STEIBL, S., BORKAR, S., DE, V., VAN DER WIJNGAART, R., AND MATTSON, T. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)* (2010), ISSCC, pp. 108–109.

[17] HP LABS. The Machine. http://www.hpl.hp.com/research/systems-research/themachine/, January 2015.

[18] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (2010), USENIX ATC'10.

[19] KANNAN, S., GAVRILOVSKA, A., SCHWAN, K., AND MILOJICIC, D. Optimizing checkpoints using NVM as virtual memory. In *Proceedings of the 27th IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (May 2013), pp. 29–40.

[20] KELLY, T., KUNO, H., PICKETT, M., BOEHM, H., DAVIS, A., GOLAB, W., GRAEFE, G., HARIZOPOULOS, S., JOISHA, P., KARP, A., MURALIMANOHAR, N., PERNER, F., MEDEIROS-RIBEIRO, G., SEROUSSI, G., SIMITSIS, A., TARJAN, R., AND WILLIAMS, S. Sidestep: Co-designed shiftable memory and software. Tech. Rep. Technical Report HPL-2012-235, Hewlett-Packard Laboratories, 2012.

[21] KLEEN, A. Machine check handling on Linux. *SUSE Labs* (2004).

[22] KYATHSANDRA, J., AND ZHOU, X. Rack Scale Architecture: Designing the Data Center of the Future. http://bit.ly/idf14-rsa, Intel IDF14 Shenzen, 2014.

[23] MILOJICIC, D., HOYLE, S., MESSER, A., MUNOZ, A., RUSSELL, L., WYLEGALA, T., VELLANKI, V., AND CHILDS, S. Global Memory Management for a Multi Computer System. In *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4* (2000), WSS'00.

[24] MILOJICIC, D., MESSER, A., SHAU, J., FU, G., AND MUNOZ, A. Increasing relevance of memory hardware errors: A case for recoverable programming models. In *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System* (2000), EW 9, pp. 97–102.

[25] NOVAKOVIC, S., DAGLIS, A., BUGNION, E., FALSAFI, B., AND GROT, B. Scale-out numa. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ACM, pp. 3–18.

[26] PARK, S., KELLY, T., AND SHEN, K. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13, pp. 225–238.

[27] PATTERSON, D., ANDERSON, T., CARDWELL, N., FROMM, R., KEETON, K., KOZYRAKIS, C., THOMAS, R., AND YELICK, K. A case for intelligent RAM. *IEEE Micro 17*, 2 (Mar. 1997), 34–44.

[28] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14, pp. 1–16.

[29] PUTNAM, A., CAULFIELD, A. M., CHUNG, E. S., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., PRASHANTH, G., JAN, G., MICHAEL, G., HAUCK, H. S., HEIL, S., HORMATI, A., KIM, J.-Y., LANKA, S., LARUS, J., PETERSON, E., POPE, S., SMITH, A., THONG, J., YI, P., AND BURGER, X. D. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014), ISCA '14, pp. 13–24.

[30] RANGANATHAN, P. From microprocessors to nanostores: Rethinking data-centric systems. *IEEE Computer 44*, 1 (2011), 39–48.

[31] ROSSBACH, C., CURREY, J., AND WITCHEL, E. Operating systems must support GPU abstractions. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems* (2011), HotOS'11.

[32] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's time for low latency. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS XIII* (2011).

[33] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C. V., CARTON, R. W., AND OFIR, J. Deciding when to forget in the elephant file system. In *Proceedings of the 17th Symposium on Operating Systems Principles (SOSP* (1999), SOSP'99.

[34] STRUKOV, D. B., SNIDER, G. S., STEWART, D. R., AND WILLIAMS, R. S. The missing memristor found. *Nature 453* (May 2008), 80–83.

[35] THE ECONOMIST. The data deluge: Businesses, governments and society are only starting to tap its vast potential. http://www.economist.com/node/15579717, Feb. 2010.

[36] WITCHEL, E., CATES, J., AND ASANOVIĆ, K. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (2002), ASPLOS X, pp. 304–316.

[37] WOODRUFF, J., WATSON, R. N., CHISNALL, D., MOORE, S. W., ANDERSON, J., DAVIS, B., LAURIE, B., NEUMANN, P. G., NORTON, R., AND ROE, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st Annual International Symposium on Computer Architecture* (2014), ISCA '14, pp. 457–468.

[38] XIE, Y. Modeling, architecture, and applications for emerging memory technologies. *IEEE Design & Test of Computers 28*, 1 (2011), 44–51.

[39] YOON, D. H., MURALIMANOHAR, N., CHANG, J., RANGANATHAN, P., JOUPPI, N. P., AND EREZ, M. Free-p: Protecting non-volatile memory against both hard and soft errors. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (2011), IEEE, pp. 466–477.