# Hardware Trends: Challenges and Opportunities in Distributed Computing

Tim Harris
timothy.l.harris@oracle.com
Oracle Labs, Cambridge, UK

This article is about three trends in computer hardware, and some of the challenges and opportunities that I think they provide for the distributed computing community. A common theme in all of these trends is that hardware is moving away from assumptions that have often been made about the relative performance of different operations (e.g., computation versus network communication), the reliability of operations (e.g., that memory accesses are reliable, but network communication is not), and even some of the basic properties of the system (e.g., that the contents of main memory are lost on power failure).

Section 1 introduces "rack-scale" systems and the kinds of properties likely in their interconnect networks. Section 2 describes challenges in systems with shared physical memory but without hardware cache coherence. Section 3 discusses non-volatile byte-addressable memory. The article is based in part on my talk at the ACM PODC 2014 event in celebration of Maurice Herlihy's sixtieth birthday.

## 1 Rack-Scale Systems

Rack-scale computing is an emerging research area concerned with how we design and program the machines used in data centers. Typically, these data centers are built from racks of equipment, with each rack containing dozens of discrete machines. Over the last few years researchers have started to weaken the boundaries between these individual machines, leading to new "rack-scale" systems. These architectures are being driven by the need to increase density and connectivity between servers, while lowering cost and power consumption.

Different researchers mean somewhat different things by "rack-scale" systems. Some systems are built from existing components. These are packaged together for a particular workload, providing appropriate hardware, and pre-installed software. Other researchers mean systems with internal disaggregation of components: rather than having a rack of machines each with its own network interface

and disk, there might be a pool of processor nodes, disk nodes, and networking nodes, all connected over an internal intra-machine interconnect. The interconnect can be configured to connect sets of these resources together in different ways.

Initial commercial systems provide high-density processor nodes connected through an in-machine interconnect to storage devices or to external network interfaces. Two examples are the HP MoonShot [12] and AMD SeaMicro [22] single-box cluster computers. Many further ideas are now being explored in research projects—for instance, the use of custom system-on-chip (SoC) processors in place of commodity chips.

These systems should not just be seen as a way to build a faster data center. Communicating over a modern interconnect is different from communicating over a traditional packet-switched network. Some differences are purely trends in performance—a round-trip latency for over InfiniBand is around 1$\mu$s, not much longer than the time it takes to access data stored in DRAM on a large shared-memory multiprocessor. The Scale-Out NUMA architecture provides one example of how latencies may be reduced even further: it exposes the interconnect via a specialized "remote memory controller" (RMC) on a multi-core SoC [18]. Threads in one SoC can instruct the RMC to transfer data to or from memory attached to other processors in the system. Threads communicate with their RMC over memory-mapped queues (held in the SoC's local caches). These operations have much lower latency than accessing a traditional network interface over PCI-express. If network latencies continue to fall, while memory access latencies remain constant, then this will change the optimization goals when designing a protocol.

Other differences are qualitative: as with the Scale-Out NUMA RMC, the main programming interface in many rack-scale systems is RDMA (remote direct memory access). To software, RDMA appears as a transfer from a region of a sender's address space into a region in the receiver's address space. Various forms of control message and notification can be used—e.g., for a receiver to know when data has arrived, or for a sender to know when transmission is complete. Flow control is handled in hardware to prevent packet loss.

Some network devices provide low-latency hardware distribution of data to multiple machines at once (for instance, the ExaLINK matrix switches advertise 5ns latency multicasting data from an input port to any number of output ports [1]). Researchers are exploring how to use this kind of hardware as part of an atomic broadcast mechanism [7].

**Research questions:** What are the correct communication primitives to let applications benefit from low-latency communication within the system? What are the likely failure modes and how do we achieve fault tolerance? What is the ap-

propriate way to model the guarantees provided by the interconnect fabric in a rack-scale system? How should the interconnect fabric be organized, and how should CPUs, DRAM, and storage be placed in it?

# 2 Shared Memory Without Cache Coherence

The second trend I will highlight is toward systems with limited support for cache coherence in hardware: Some systems provide shared physical memory, but rely on threads to explicitly flush and invalidate their local caches if they want to communicate through them. Some researchers argue that cache coherence will be provided within a chip, but not between chips [15].

This kind of model is not entirely new. For instance, the Cray T3D system distributed its memory across a set of processor nodes, providing each node with fast access to its local memory, and slower access to uncacheable remote memory [6]. This kind of model makes it important to keep remote memory accesses rare because they will be slow even in the absence of contention (for instance, lock implementations with local spinning are well suited in this setting [16]).

One motivation for revisiting this kind of model is to accommodate specialized processors or accelerators. The accelerator can transfer data to and from memory (and sometimes to and from the caches of the traditional processors) but does not need to participate in a full coherence protocol. A recent commercial example of this kind of system is the Intel Xeon Phi co-processor accessed over PCI-express [13].

A separate motivation for distributing memory is to provide closer coupling between storage and computation. The IRAM project explored an extreme version of this with the processor on the same chip as its associated DRAM [19]. Close coupling between memory and storage can improve the latency and energy efficiency of memory accesses, and permit the aggregate bandwidth to memory to grow by scaling the number of memory-compute modules.

Some research systems eschew the direct use of shared memory and instead focus on programming models based on message passing. Shared memory buffers can be used to provide a high-performance implementation of message passing (for instance, by using a block of memory as a circular buffer to carry messages). This approach means that only the message passing infrastructure needs to be aware of the details of the memory system. Also, it means that software written for a genuinely distributed environment is able to run correctly (and hopefully more quickly) in an environment where messages stay within a machine.

Systems such as K2 [14] and Popcorn [4] provide abstractions to run existing shared-memory code in systems without hardware cache coherence, using ideas from distributed shared memory systems.

Conversely, the Barrelfish [5] and FOS [23] projects have been examining the use of distributed computing techniques within an OS. Barrelfish is an example of a *multikernel* in which each core runs a separate OS kernel, even when the cores operate in a single cache-coherent machine. All interactions between these kernels occur via message-passing. This design avoids the need for shared-memory data structures to be managed between cores, enabling a single system to operate across coherence boundaries. While it is elegant to rely solely on message passing, this approach seems better suited to some workloads than to others—particularly when multiple hardware threads share a cache, and could benefit from spatial and temporal locality in the data they are accessing.

**Research questions:** What programming models and algorithms are appropriate for systems which combine message passing with shared memory? To what extent should systems with shared physical memory (without cache coherence) be treated differently from systems without any shared memory at all?

# 3 Non-Volatile Byte-Addressable Memory

There are many emerging technologies that provide non-volatile byte-addressable memory (NV-RAM). Unlike ordinary DRAM, memory contents are preserved on power loss. Unlike traditional disks, locations can be read or written at a fine granularity—nominally individual bytes, although in practice hardware will transfer complete cache lines. Furthermore, unlike a disk, these reads and writes may be performed by ordinary memory access instructions (rather than using RDMA, or needing the OS to orchestrate block-sized transfers to or from a storage device).

This kind of hardware provides the possibility of an application keeping all of its data structures accessible in main memory. Researchers are starting to explore how to model NV-RAM [20]. Techniques from non-blocking data structures provide one starting point for building on NV-RAM. A power loss can be viewed as a failure of all of the threads accessing a persistent object. However, there are several challenges which complicate matters:

First, the memory state seen by the threads before the power loss is not necessarily the same as the state seen after recovery. This is because, although the NV-RAM is persistent, the remainder of the memory system may hold data in ordinary volatile buffers such as processor caches and memory controllers. When power is lost, some data will transiently be in these volatile buffers. Aggressively flushing every update to NV-RAM may harm performance. Some researchers have explored flushing updates upon power-loss, but that approach requires careful analysis to ensure that there is enough residual power to do so [17].

The second problem is that applications often need to access several structures—for instance, removing an item from one persistent collection object, processing it, and adding it to another persistent collection. If there is a power loss during the processing step, then we do not want to lose the item.

Transactions provide one approach for addressing these two problems. It may be possible to optimize the use of cache flush/invalidate operations to ensure that data is genuinely persistent before a transaction commits, while avoiding many individual flushes while the transaction executes. As with transactional memory systems, transactions against NV-RAM would provide a mechanism for composing operations across multiple data structures [10]. What is less clear is whether transactions are appropriate for long-running series of operations (such as the example of processing an object when moving it between persistent collections).

Having an application's data structures in NV-RAM could be a double-edged sword. It avoids the need to define translations between on-disk and in-memory formats, and it avoids the time taken to load data into DRAM for processing. This time saving is significant in "big data" applications, not least when restarting a machine after a crash. However, explicit loading and saving has benefits as well as costs: It allows in-memory formats to change without changing the external representation of data. It allows external data to be processed by tools in a generic way without understanding its internal formats (backup, copying, de-duplication, etc.). It provides some robustness against transient corruption of in-memory formats by restarting an application and re-loading data.

It is difficult to quantify how significant these concerns will be. Earlier experience with persistent programming languages explored many of these issues [3]. Recent work on dynamic software updates is also relevant (for instance, Arnold and Kaashoek in an OS kernel [2], and Pina *et al.* in applications written in Java [21]).

**Research questions:** How should software manage data held in NV-RAM, and what kinds of correctness properties are appropriate for a data structure that is persistent across power loss?

# 4 Discussion

This article has touched on three areas where developments in computer hardware are changing some of the traditional assumptions about the performance and behavior of the systems we build on.

Processor clock rates are not getting significantly faster (and, many argue, core counts are unlikely to increase much further [9]). Nevertheless, there are other

ways in which system performance can improve such as by integrating specialized cores in place of general-purpose ones, or by providing more direct access to the interconnect, or by removing the need to go through traditional storage abstractions to access persistent memory.

I think many of these trends reflect a continued blurring of the boundaries between what constitutes a "single machine" versus what constitutes a "distributed system". Reliable interconnects are providing hardware guarantees for message delivery, and in some cases this extends to guarantees about message ordering as well even in the presence of broadcast and multicast messages. Conversely, the move away from hardware cache coherence within systems means that distributed algorithms become used in systems which look like single machines—e.g., in the Hare filesystem for non-cache-coherent multicores [8].

Many of these hardware developments have been proceeding ahead of the advancement of formal models of the abstractions being built. Although the use of verification is widespread at low levels of the system – especially in hardware – I think there are important opportunities to develop new models of the abstractions exposed to programmers. There are also opportunities to influence the direction of future hardware evolution—perhaps as with how the identification of the consensus hierarchy pointed to the use of atomic compare and swap in today's multiprocessor systems [11].

# References


[1] EXALINK Fusion (web page). Apr. 2015. `https://exablaze.com/exalink-fusion`.

[2] J. Arnold and M. F. Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proc. 4th European Conference on Computer Systems (EuroSys)*, pages 187–198, 2009.

[3] M. Atkinson and M. Jordan. A review of the rationale and architectures of PJama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical report, University of Glasgow, Department of Computing Science, 2000.

[4] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: bridging the programmability gap in heterogeneous-ISA platforms. In *EuroSys '15: Proc. 10th European Conference on Computer Systems (EuroSys)*, page 29, 2015.

[5] 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 *SOSP '09: Proc. 22nd Symposium on Operating Systems Principles*, pages 29–44, 2009.

[6] Cray Research Inc. *CRAY T3D System Architecture Overview Manual*. 1993. ftp://ftp.cray.com/product-info/mpp/T3D_Architecture_Over/T3D.overview.html.

[7] M. P. Grosvenor, M. Fayed, and A. W. Moore. Exo: atomic broadcast for the rack-scale computer. 2015. http://www.cl.cam.ac.uk/~mpg39/pubs/workshops/wrsc15-exo-abstract.pdf.

[8] C. Gruenwald III, F. Sironi, M. F. Kaashoek, and N. Zeldovich. Hare: a file system for non-cache-coherent multicores. In *EuroSys '15: Proc. 10th European Conference on Computer Systems*, page 30, 2015.

[9] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, 2011.

[10] T. Harris, M. Herlihy, S. Marlow, and S. Peyton Jones. Composable memory transactions. In *PPoPP '05: Proc. 10th Symposium on Principles and Practice of Parallel Programming*, June 2005.

[11] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[12] HP Moonshot system: a new class of server. http://www.hp.com/go/moonshot, Accessed 9 July 2014.

[13] Intel Corporation. Intel Xeon Phi coprocessor system software developers guide. 2012. IBL Doc ID 488596.

[14] F. X. Lin, Z. Wang, and L. Zhong. K2: a mobile operating system for heterogeneous coherence domains. In *ASPLOS '14: Proc. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 285–300, 2014.

[15] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, 2012.

[16] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, Feb. 1991.

[17] D. Narayanan and O. Hodson. Whole-system persistence. In *ASPLOS '12: Proc. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–410, 2012.

[18] S. Novaković, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-Out NUMA. In *ASPLOS '14: Proc. 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[19] D. A. Patterson, K. Asanovic, A. B. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. E. Kozyrakis, D. B. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. A. Yelick. Intelligent RAM (IRAM): the industrial setting, applications and architectures. In *Proceedings 1997 International Conference on Computer Design: VLSI in Computers & Processors, ICCD '97, Austin, Texas, USA, October 12-15, 1997*, pages 2–7, 1997.

[20] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[21] L. Pina, L. Veiga, and M. Hicks. Rubah: DSU for Java on a stock JVM. In *OOPSLA '14: Proc. Conference on Object-Oriented Programming Languages, Systems, and Applications*, Oct. 2014.

[22] A. Rao. SeaMicro SM10000 system overview, June 2010. `http://www.seamicro.com/sites/default/files/SM10000SystemOverview.pdf`.

[23] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, Apr. 2009.