Consensus for Non-Volatile Main Memory

Huynh Tu Dang*, Jaco Hofmann†, Yang Liu‡, Marjan Radi‡, Dejan Vucinic‡, Fernando Pedone*, Robert Soulé*
*Università della Svizzera italiana
{danghu, fernando.pedone, robert.soule}@usi.ch
†Technische Universität Darmstadt
hofmann@esa.tu-darmstadt.de
‡Western Digital Research
{yang.liu0, marjan.radi, dejan.vucinic}@wdc.com

***Abstract*—Traditionally, computer storage has been separated into a hierarchy based on response time, volatility, and cost of media. This tiering is undergoing a significant upheaval as a new breed of memory technologies, termed Storage Class Memories (SCM), now make it feasible to replace several tiers of the hierarchy with a single, cost-effective, uniform type of memory/storage. To make large-scale SCM deployments practical, however, memory system designers will first need to solve the problem of how to guard against unavoidable storage wear-out and failures—problems traditionally absent from "main memory" and handled by software at leisurely timescales in the domain of storage.**

In this paper, we propose a novel approach to providing fault tolerance in SCM-based main memory. Our key insight is to treat memory as a distributed storage system and rely on data replication and a consensus protocol to keep the replicas consistent. Separate memory instances store replicated copies of the data, and we use a programmable network interconnect to provide fast consensus between the memory instances. Our initial experiments using software memory controller emulation demonstrate reasonable overhead over local memory reads and show great promise as scalable main memory.

***Index Terms*—storage class memory, consensus, in-network computing**

I. Introduction

Computer memory and storage are organized in a hierarchy with tiers distinguished by response time, volatility, and cost. At the top of the hierarchy are SRAM caches and DRAM main memory, which have low latency, unlimited write endurance, and fine granularity of access. They are, however, power-hungry, expensive and volatile, necessitating further tiering to solid-state NAND flash storage (SSD), and finally to spinning disk or tape magnetic storage at the bottom of the hierarchy. These terminal tiers of non-volatile and durable bit storage have much higher access latency and granularity than volatile memory and, in the case of NAND flash, finite write endurance limiting the total amount of data that can be written before device replacement is required.

This traditional organization is being shaken up by the advent of Storage Class Memory (SCM): several emerging memory technologies, such as Phase-Change Memory (PCM) [1], Resistive RAM (ReRAM) [2], and Spin-Torque Magnetic RAM (STT-MRAM) [3] are non-volatile, offer byte-addressability, and response times not much slower than DRAM, but could cost significantly less on account of simpler memory cell architecture resulting in denser packing. Recent breakthroughs in selector element physics [4] enabled larger memory cell arrays which result in better utilization of die area, leading to further cost advantages over DRAM. At the time of writing one such product based on cross-point PCM (Intel Optane® M.2) is about 6.7x cheaper per gigabyte at retail as a result of acute DRAM shortage. Consequently, in some scenarios it is feasible to replace several tiers of the traditional memory/storage hierarchy with a single, cost-effective, uniform type of memory that also serves as the terminal tier of non-volatile and durable data storage.

While this sounds appealing from the architectural simplicity and elegance standpoint, there is a fly in the ointment. All known SCM technologies involve movement of atoms, and so have unavoidable wear-out mechanisms resulting in limited write (and sometimes read) endurance of devices. This places severe practical limits on scale-out size of storage systems built on top of these technologies, and even the practicality of single systems where DRAM is replaced with cheaper SCM main memory that is, alas, guaranteed to fail after brief use.

What this means in practice is that to enable significant displacement of DRAM in prevailing systems architectures, we must translate a variety of techniques traditionally used for slow durable storage (e.g., RAID for disks or SSDs) to work at timescales suitable for main memory. This strategy would enable us to satisfy data replication and consistency requirements that are taken for granted in the current many-tiered architectures.

Memory faults are not unique to SCM, even though the details of how the errors occur differ depending on the storage medium. A wide spectrum of approaches are used to solve the problem. For main memory, many systems simply ignore the issue, and treat the memory as if there were no errors. This can result in crashes—any error detected in main memory or caches is handled by simply shutting down the entire system. In fact, memory errors are one of the largest causes of machine failures [5]. Clearly, this approach doesn't scale to larger main memories for obvious reasons.

Contemporary supercomputers, where memory Mean Time Between Failures (MTBF) is measured in minutes on account of the sheer number of independent components that can fail, deal with main memory faults by "checkpointing", i.e. periodically storing a copy of all memory on disks. Sophisticated management is required to keep the overhead of checkpointing reasonable [6], and cost is not to be spoken of.

Disk and NAND flash SSD storage often use RAID. Unfortunately, RAID does not work well at scale since it depends on a centralized controller, the failure of which renders the data unavailable and possibly corrupted.

In this paper we propose a new approach to providing fault-tolerance in non-volatile main memory based on SCM. Our key insight is to treat memory as a distributed storage system, and rely on data replication with a consensus protocol to keep the replicas consistent through failures. Although consensus protocols have been historically considered a performance bottleneck, several recent projects have demonstrated a promising new approach to achieving high-performance consensus [7]–[12]. These systems leverage programmable network hardware [13]–[15] to execute consensus logic directly in the network forwarding plane, achieving tremendous reductions in latency and increases in throughput.

Our approach uses replicated SCM instances that are kept consistent by a programmable interconnect running a generalization of a protocol by Attiya, Bar-Noy, and Dolev [16]. We refer to this as the ABD protocol. The ABD protocol is well suited to the task for several reasons. First, the protocol ensures linearizable read/write access to memory, while tolerating failures. Second, ABD is simpler than more general protocols, such as Paxos [8], allowing for an efficient, in-network implementation. Third, ABD only requires that the switch keeps soft state during the protocol exchange, reducing the reliance on scarce switch resources (e.g., SRAM, TCAM).

Overall, this paper makes the following contributions:

- We identify constraints and requirements for an in-network implementation of a consensus protocol for storage class memory.
- We describe an implementation of a memory controller and network adaptation of the ABD protocol.
- We provide initial evidence of the feasibility and benefits of using in-network consensus to keep replicated PCM consistent.

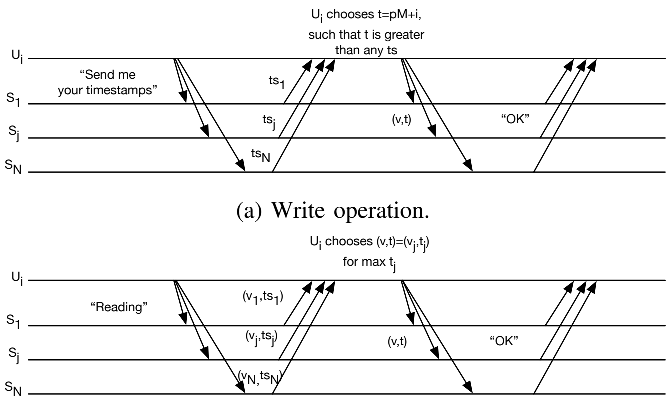
The rest of this paper is organized as follows. We first provide background on phase-change memory and the ABD protocol (§II). We then summarize out focusing on how the protocol maps to network hardware abstractions (§III). Next, we present an evaluation on programmable ASICs and FPGAs (§V). Related work is discussed in (§VI). Finally, we conclude in (§VII).

II. Background

Before discussing the details of our system design, we first briefly present background on both a leading SCM technology and the ABD protocol.

A. Phase-Change Memory

Of the many SCM technologies explored in research laboratories, Phase-Change Memory [17] has been the most successful in the marketplace to date, at first in power-constrained mobile devices [18] and more recently in enterprise storage [19]. The memory element relies on the peculiar phase diagram of so-called amorphous semiconductors, most commonly alloys of Germanium, Antimony and Telluride (GST), which exhibit two distinct solid phases. If the material is heated and cooled quickly, it stays in an amorphous solid state with high resistivity and good optical transparency. If instead the material is heated just below the critical melting temperature, it crystallizes into an opaque solid state of low resistivity.



(a) Write operation.

(b) Read operation.

Fig. 1: The ABD protocol.

GST materials were first explored in late 1960s [20] and found widespread use in optical storage media (e.g., Blu-Ray®) but the technology to make them commercially viable as solid-state memories has only recently matured (e.g., Optane® and 3D XPoint® from Intel and Micron) [21]. The breakthrough involved the development of a suitable selector device [4], [22], [23] permitting larger arrays of memory cells and better die utilization, which resulted in reduced cost and increased profitability of the technology.

PCM has several attractive qualities as a memory technology. First, it has very fast response time, practically on the order of a hundred nanoseconds but reaching even below one nanosecond under laboratory conditions [24], well into DRAM's domain. To put that in context, read latency of modern high-capacity NAND flash is on the order of 50-70 microseconds, making SSD response times in the vicinity of 100 microseconds after error correction and protocol overhead. Second, unlike NAND flash, PCM is byte-addressable on both reads and writes, so requires no erase block management and garbage collection which cause poor latency tails [25]. Third, PCM is naturally non-volatile due to the properties of the GST material. Other forms of non-volatile memory with comparable response times, such as battery-backed DRAM, require constant power with its associated cost and logistical complexity. Fourth, PCM has high write endurance of more than a million cycles and long retention time of many years. Finally, PCM is inherently less expensive to produce than

DRAM at lithographic parity as a result of its denser packing and simpler memory cell structure.

B. ABD Protocol

Attiya, Bar-Noy, and Dolev described a protocol for implementing an atomic register in an asynchronous message-passing system [16]. This protocol is well-suited as a building block for providing fault-tolerance for storage class memory, because the protocol is optimized for read and write requests—i.e., the operations that we would expect from memory. It is more efficient in terms of communication steps than alternative protocols, such as Paxos [26] and Chain Replication [27], which allow for arbitrary operations (e.g., increment).

The protocol assumes that there are user processes that have access to message channels and would like to execute read and write operations as if they had some shared memory at their disposal (i.e., emulating shared memory with message passing). Although the original paper assumes a single writer, the protocol can be easily generalized for multiple writers and multiple readers. We refer to the generalized protocol, which we describe below, as the ABD protocol.

We first describe the general formulation of the protocol, before discussing the modifications that we need to make for a switch-based deployment in Section III.

The ABD protocol assumes there are M user processes, and N server processes. Every user process can send a message to every server process, and vice-versa. Each user process $U_i \in \{U_1, \ldots, U_M\}$ chooses a unique timestamp of the form $t = pM + i$, where p is a positive integer. For example, if $M = 32$, U_1 chooses timestamps from the set $\{1, 33, 65, \ldots\}$. This naming convention allows us to easily identify which user process issued a request. Both read and write requests require two phases.

To write a value, v, the user process, U_i, sends a message to all server processes, requesting their timestamp. Each server process, $S_j \in \{S_1, \ldots, S_N\}$ responds with their current timestamp, ts_j. Upon receiving a majority of responses, U_i chooses a new timestamp, t, of the form $t = pM+i$, such that t is greater than its previous t and any ts_j it received. U_i sends the pair (v, t) to all server processes. The server processes compare t to their local timestamp, ts_k. If t is greater than ts_k, the server processes update their value and timestamp to v and t, and return an acknowledgement to U_i.

To perform a read, the user process, U_i, sends a read message to all server processes. Each server process, $S_j \in \{S_1 \ldots S_N\}$ responds with their current value and timestamp, (v_j, ts_j). Upon receiving a majority of responses, U_i chooses $(v, j) = (v_j, ts_j)$ for the maximum value of ts_j. Then, like the write operation, U_i then sends the pair (v, t) to all server processes. The server processes compare t to their local timestamp, ts_k. If t is greater than ts_k, the server processes update their value and timestamp to v and t, and return an acknowledgement to U_i.

III. Design

Figure 2 illustrates the high-level design of our system. Overall, there are three main components. Clients, using a custom memory controller, issue read and write requests. A set of memory instances service those requests. The stored data is replicated across several memory instances. A programmable switch running a modified version of the ABD protocol interposes on all requests, and ensures that the replicas stay consistent.

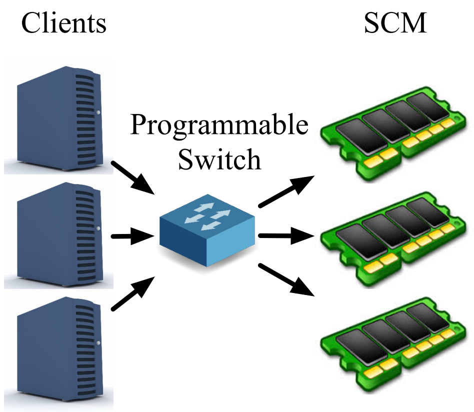


Fig. 2: Clients issue memory read/write requests to off-device storage-class memory instances. A programmable switch runs ABD protocol to keep replicas consistent.

Implementing the ABD functionality as part of the switching fabric allows multiple replicas of data to be kept consistent, while satisfying the stringent performance demands on memory accesses. However, implementing this logic on any ASIC (including reconfigurable ones) imposes constraints due to the physical nature of the hardware. These constraints include:

- *Memory*. The amount of memory available in each stage for stateful operations or match actions is limited [28].
- *Primitives*. Each stage of the pipeline can only execute a single ALU instruction per field.
- *State between stages*. There is a limited amount of Packet Header Vector (PHV) that pass state between stages.
- *Depth of pipeline*. There is a fixed number of match units.

The goal of our design is to provide efficient implementation of the ABD algorithm that respects the physical limitations of the hardware. In creating our design, we necessarily make some assumptions about the deployment. To make these assumptions explicit, we list them below:

- We do not want to extend memory controller with logic for replication. It should only be aware of read/write requests. This is to simplify integration with existing coherence buses and CPU cache controllers and avoid re-engineering everything starting from the CPU pipelines.
- We assume that cache lines are 64 bytes. Since the values in the ABD protocol are cache lines, the size of the values in the protocol are also 64 bytes.
- We assume that switches do not fail. In reality, any device may fail, and a true fault-tolerant system would account for those failures. However, accounting for switch failure would make the protocol significantly more complex. Because the mean time to failure for memory is significantly

shorter than the mean time to failure for a switch, we start with the simplified version.

- We assume that clients are directly connected to the switch, with one client per port. This constrains the deployment topology, and this constraint may not hold in practice. This assumption could be relaxed given an appropriate tunneling protocols between clients and the switch. However, again, as a first step, we make this assumption to simplify the protocol.
- We assume that the system will need to support $\sim$1000 CPUs, each issuing about 10 concurrent requests. So, the load that the switch needs to support will be about 10K concurrent requests at a time.

Below, we describe the design of the memory controller and switch logic in more detail.

A. Memory Controller

The system needs to issue ABD reads and writes transparently without modifying user applications. To achieve this, we provide a pair of special device drivers (client and server) to handle page faults. When an application on the client calls `malloc`, instead of going to the standard system call implementation of the library, our system intercepts the library call, and invokes `mmap` on the character device we create. The client device driver then allocates the requested size of memory from the kernel driver on the remote server, and returns the address back to the client driver.

The client device driver maintains a local buffer with configurable size (set to page size of 4KB by default) to serve the page faults in the first place. When there is a miss in the local buffer, the driver will issue ABD accesses to fetch the page remotely. If the local buffer is dirty at the miss, the content of the buffer will be written to the remote server first, before the requested content can be retrieved from the server and updated to the local buffer.

The servers and the clients communicate with a remote procedure call (RPC) mechanism inside the drivers, so that the remote servers know how to handle `malloc`, `free`, and ordinary reads and writes issued by the clients.

B. Switch Logic

Our deployment model and assumptions necessitate that we modify the original protocol described in Section II-B. The original protocol is designed to access a single register. We need to generalize the protocol to support multiple registers, each corresponding to a different cache line. Moreover, because we don't want the memory controller to be aware of the replication (i.e., it should simply issue read and write requests), the switch needs to maintain the timestamps that are stored at the client in the original protocol.

The amount of state that needs to be stored on the switch is dependent on a few different variables. First, the size of the address space and the size of the cache lines determine the number of cache lines that need to be stored:

$$\#\text{ of cache lines} = \frac{\text{size of address space}}{\text{size of cache line}} \tag{1}$$

Our implementation used a cache line of 64 bytes, and an address space for 4GB of data.

P4 offers a programming abstraction of “registers”, which are an array of cells. The size of each cell is bound by the width of the ALU on the underlying hardware. Since the size of the cell is less than the size of the cache line, the cache line needs to be split across multiple register entries. The number of register cells per cache line is determined by the following equation:

$$\text{cells per cache line} = \frac{\text{size of cache line}}{\text{size of cell}} \tag{2}$$

Ideally, we would store one timestamp per cache line per port. However, if the address space is too big, then one can keep a timestamp per block of cache lines. Overall, the number of cache lines and number of timestamps must be less than the total memory available:

$$\begin{aligned} &((\#\text{ of cache lines} \times \text{entries per cache line}) \\ &\quad + \#\text{ of timestamps}) \times (\text{size of cell}) \\ &\leq (\text{memory per stage}) \times (\#\text{ of stages}) \end{aligned} \tag{3}$$

Moreover, the switch code uses an additional 4 registers, each with (# of timestamps) cells of size 8-bits for quorum checking in each phase of read/write requests; including timestamp and write quorums in a write request; and read and write-back quorums in a read request.

The switch also has a table for forwarding packets. Forwarding is done at layer 2. ABD packets should not need an IP header, although our prototype implementation still uses them, as they are required by the server NICs. To send messages to a set of memory replicas, our implementation uses Ethernet multicast. We assign one multicast group to each set of replicas; when sending messages to the replicas, the switch code sets the destination MAC address to be the multicast group identifier.

C. Failure Assumptions

In contrast to Paxos [26], which depends on the elecion of a non-faulty leader for progress, the ABD protocol only depends on the availability of a majority of participants. The ABD protocol assumes that the failure of a participant does not prevent connectivity between other participants, which can be violated in the event of a switch failure. To cope with switch failures, there would need to be a redundant component, and the protocol would need to be extended to include a notion of sending to the primary or the backup. For now, our prototype assumes that switches do not fail. Packet reordering is handled naturally by the ABD protocol, which ensures atomicity (i.e., serializability). To cope with packet loss, we rely on time-outs. If a client does not receive a response after the time limit, it must resend the request.

IV. IMPLEMENTATION

The switch logic for the client side of the ABD protocol was implemented with 858 lines of $P4_{14}$ code, and compiled using Barefoot Capilano to run on Barefoot Network's

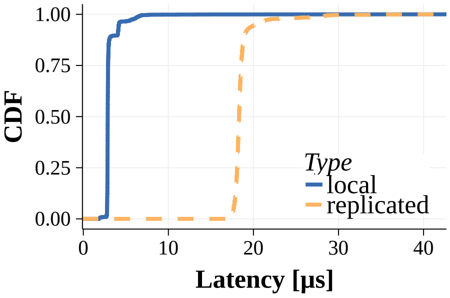


Fig. 3: Latency CDF reading a cache line from local memory and from the replicated remote memories.

Tofino ASIC [14]. To simulate the memory endpoints, we used NetFPGA SUME FPGAs. The server side code of the ABD protocol was written with 208 lines of $P4_{16}$ code, and compiled using P4-NetFPGA [29] to run on the NetFPGA SUME boards.

The memory controller emulator is implemented as a Linux character device driver. It maintains the necessary data structures, and handles page faults by sending and receiving packets to and from the servers. When incoming packets arrive, the driver handles the actual memory management and content updates, and returns the requested content back to the clients (i.e., applications). The driver is written with 1157 lines of C code.

V. Evaluation

Our evaluation quantifies the overhead for page fault handling via calls to remote replicated memory versus local memory.

For the experimental setup, we used a 32-port ToR switch with Barefoot Networks Tofino ASIC [14]. The switch, which ran the ABD protocol, was configured to run at 10G per port. To simulate the memory endpoints, we used three NetFPGA SUME FPGAs. To issue read and write requests, we used the kernel client running on a commodity server. The testbed included three NetFPGA SUME hosted in three Supermicro 6018U-TRTP+ servers. One server was also the client. The servers have dual-socket Intel Xeon E5-2603 CPUs, with a total of 12 cores running at 1.6GHz, 16GB of 1600MHz DDR4 memory and one Intel 82599 10Gbps NIC. All connections used 10G SFP+ copper cables. The servers were running Ubuntu 16.04 with Linux kernel version 4.10.0.

For our preliminary experiments, we did not yet implement a true memory controller in hardware. Instead, we emulate the behavior using an application that calls `mmap` to map a file into memory, and then issues write requests to addresses at different pages. We recorded the time before and after the write requests to measure the latency for each request. We repeated the measurement 100K times under two different configurations: one with unmodified Linux network page handler servicing the requests locally, and one with the calls to remote, replicated memories.

The results are shown in Figure 3. The median latency for fetching a cache line from local memory is 3 μs while it takes 18 μs for fetching a page from the remote replicated memories. The latency is pretty stable around 18 μs. We note that these measurements include full L2 parsing. A custom protocol could further reduce the latency. These results are encouraging. The performance is significantly faster than traditional replicated storage systems and shows great promise for use with scalable main memory.

VI. Related Work

This paper describes an in-network implementation of the ABD protocol to provide consensus for Storage Class Memories. As such, there is related work on different consensus protocols, in-network computing, and low-latency network hardware used for storage.

Consensus Protocols: The ABD protocol differs from both Paxos [26], and Chain Replication [27] in several respects. First, the ABD protocol is optimized for read and write requests, as opposed to arbitrary operations (e.g., increment). As a result, it is more efficient that Paxos and Chain Replication in terms of communication steps. Chain Replication assumes that failures are detected reliably, while Paxos and ABD do not. Paxos depends on the elecion of a non-faulty leader, or otherwise the protocol does not make progress. Chain Replication relies on a trusted entity outside of the chain to manage group membership, and that entity is usually implemented as a replicated service using Paxos. Thus, Chain Replication also depends on a non-faulty leader. In contrast, with ABD, as long as there is a majority of non-faulty processes, an operation can finish.

In-network Computing: Several recent projects [9]–[12] have leveraged the emerging trend of programmable networks hardware [13]–[15] to optimize consensus. From a high level, there are two approaches to network-accelerated consensus. One group of systems, such as Speculative Paxos [9] and NoPaxos [11], use programmable networks to enforce a particular behavior (e.g., increased likelihood of in-order delivery), which means that the consensus protocol makes strong assumptions about the network. If these assumptions are violated, then the systems must fall back to traditional consensus. The second group, including Paxos Made Switch-y [8], Consensus in a Box [10], and NetChain [12] implement consensus protocol logic directly in the programmable hardware.

In a separate, but related line of research, Eris [30] and NOCC [31] use programmable switches to accelerate transaction processing. However, they have very different execution models. The Eris model is based on prior work on independent transactions [32], [33], in which transactions are ordered first, and then executed. In contrast, with the NOCC model, clients pre-execute transactions locally, and then submit the result for validation (i.e., ordering).

Low-latency Network Hardware: A variety of global shared memory systems have been built on alternative low-latency network architectures which are more amenable to embedded use. For instance, the FaRM system [34] reports 31 μs latency for a distributed key-value store using RDMA protocol over Inifiniband transport. Such systems differ from the one reported here in two crucial aspects: first, they do not use a consensus system for hardware replication of remote distributed memory; and second, they rely on RDMA for software management of local copies. The essence of our solution is to pave the path to hardware-only management of local working copies of data in DRAM or SRAM caches, and so erase the "local vs. remote" dichotomy imposed on the system designer by the RDMA paradigm. Our current choice of Ethernet transport was dictated by the commercial availability of programmable dataplane switches, and does not preclude using programmable Infiniband switches in the future should they become available, with the commensurate performance gains from reliable in-order transport.

VII. Conclusion

Overall, Storage Class Memory has incredible potential to disrupt the traditional memory hierarchy. Using in-network implementations of consensus helps solve a critical challenge for adopting this new technology. Our initial experiments using software memory controller emulation already operate at time scales orders of magnitude faster than traditional replicated storage systems and show great promise as scalable main memory.

Acknowledgments

We wish to thank our shepherd, Fernando Ramos, and the anonymous reviewers for the constructive comments. This work was supported in part by SNF grants number 200021_166132 and 407540_167173, and an award from Western Digital Corporation.

References

[1] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec 2010.

[2] H. Akinaga and H. Shima, "Resistive random access memory (reram) based on metal oxides," *IEEE*, vol. 98, no. 12, pp. 2237–2251, Dec 2010.

[3] J. A. Katine, F. J. Albert, R. A. Buhrman, E. B. Myers, and D. C. Ralph, "Current-driven magnetization reversal and spin-wave excitations in co /cu /co pillars," *Physical Review Letters*, vol. 84, pp. 3149–3152, Apr 2000.

[4] G. W. Burr, R. S. Shenoy, K. Virwani, P. Narayanan, A. Padilla, B. Kurdi, and H. Hwang, "Access devices for 3d crosspoint memory," *Journal of Vacuum Science & Technology B*, vol. 32, no. 4, p. 040802, Jul 2014.

[5] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: A large-scale field study," *SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 193–204, Jun. 2009.

[6] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–11.

[7] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM, Jun. 2015, pp. 5:1–5:7.

[8] H. T. Dang, M. Canini, F. Pedone, and R. Soulé, "Paxos made switch-y," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 2, pp. 18–24, May 2016.

[9] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing distributed systems using approximate synchrony in data center networks," in *USENIX Symposium on Networked Systems Design and Implementation*, May 2015, pp. 43–57.

[10] Z. István, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a box: Inexpensive coordination in hardware," in *USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2016, pp. 425–438.

[11] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say no to paxos overhead: Replacing consensus with network ordering," in *USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2016, pp. 467–483.

[12] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2018, pp. 35–49.

[13] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *USENIX Symposium on Networked Systems Design and Implementation*, May 2015, pp. 103–115.

[14] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, Aug. 2013.

[15] "XPliant Ethernet Switch Product Family," www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html, 2014.

[16] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995.

[17] M. Wuttig and N. Yamada, "Phase-change materials for rewriteable data storage," *Nature materials*, vol. 6, no. 11, p. 824, 2007.

[18] G. Servalli, "A 45nm generation phase change memory technology," in *IEEE International Electron Devices Meeting*, Dec. 2009, pp. 1–4.

[19] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, "Reducing dram footprint with nvm in facebook," in *European Conference on Computer Systems*, Apr. 2018, pp. 42:1–42:13.

[20] S. R. Ovshinsky, "Reversible electrical switching phenomena in disordered structures," *Physical Review Letters*, vol. 21, no. 20, p. 1450, Nov. 1968.

[21] D. Kau, S. Tang, I. V. Karpov, R. Dodge, B. Klehn, J. A. Kalb, J. Strand, A. Diaz, N. Leung, J. Wu *et al.*, "A stackable cross point phase change memory," in *IEEE International Electron Devices Meeting*, Dec. 2009, pp. 1–4.

[22] F. Pellizzer and A. Pirovano, "Phase change memory with ovonic threshold switch," U.S Patent 7 677 830, Mar. 30, 2010.

[23] R. R. Shanks, "Ovonic threshold switching characteristics," *Non-Crystalline Solids*, vol. 2, pp. 504–514, Jan. 1970.

[24] D. Loke, T. Lee, W. Wang, L. Shi, R. Zhao, Y. Yeo, T. Chong, and S. Elliott, "Breaking the speed limits of phase-change memory," *Science*, vol. 336, no. 6088, pp. 1566–1569, Nov. 2012.

[25] C. Sun, D. Le Moal, Q. Wang, R. Mateescu, F. Blagojevic, M. Lueker-Boden, C. Guyot, Z. Bandic, and D. Vucinic, "Latency tails of byte-addressable non-volatile memories in systems," in *Memory Workshop (IMW), 2017 IEEE International*. IEEE, May 2017, pp. 1–4.

[26] L. Lamport, "The part-time parliament," *ACM TOCS*, vol. 16, no. 2, pp. 133–169, May 1998.

[27] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2004, pp. 7–7.

[28] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation." in *USENIX Symposium on Networked Systems Design and Implementation*, Mar. 2017, pp. 67–82.

[29] "P4-NetFPGA," https://github.com/NetFPGA/P4-NetFPGA-public, 2017.

[30] J. Li, E. Michael, and D. R. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *ACM Symposium on Operating Systems Principles*, Oct. 2017, pp. 104–120.

[31] T. Jepsen, L. P. de Sousa, M. Moshref, F. Pedone, and R. Soulé, "Infinite resources for optimistic concurrency control," in *ACM SIGCOMM Workshop on In-Network Computing*, Aug. 2018.

[32] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *Very large data bases*, Sep. 2007, pp. 1150–1160.

[33] J. Cowling and B. Liskov, "Granola: Low-overhead distributed transaction coordination," in *USENIX Annual Technical Conference*, Jun. 2012, pp. 223–235.

[34] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, "FaRM: Fast Remote Memory," in *USENIX Symposium on Networked Systems Design and Implementation*, Apr. 2014, pp. 401–414.