Data Structure Engineering For Byte-Addressable Non-Volatile Memory

Ismail Oukid
SAP SE & TU Dresden
Germany
ismail.oukid@sap.com

Wolfgang Lehner
TU Dresden
Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

Storage Class Memory (SCM) is emerging as a viable alternative to traditional DRAM, alleviating its scalability limits, both in terms of capacity and energy consumption, while being non-volatile. Hence, SCM has the potential to become a universal memory, blurring well-known storage hierarchies. However, along with opportunities, SCM brings many challenges. In this tutorial we will dissect SCM challenges and provide an in-depth view of existing programming models that circumvent them, as well as novel data structures that stem from these models. We will also elaborate on fail-safety testing challenges – an often overlooked, yet important topic. Finally, we will discuss SCM emulation techniques for end-to-end testing of SCM-based software components. In contrast to surveys investigating the use of SCM in database systems, this tutorial is designed as a programming guide for researchers and professionals interested in leveraging SCM in database systems.

1. MOTIVATION

The advent of large main-memory capacities has spurred a shift in software design towards main-memory-centric architectures, which yield orders of magnitude faster access characteristics than traditional, disk-centric approaches. The rise of Big Data emphasized the importance of such systems with an ever increasing need for larger main memory capacities. However, DRAM is hitting its scalability limits: (1) it is intrinsically hard to further increase its density [7]; and (2) it constitutes a significant share of the energy consumption in data centers [22], either directly or indirectly (e.g., by the cooling system). Storage Class Memory, also called byte-addressable Non-Volatile Memory, represents a viable alternative and is currently considered an umbrella term for novel memory technologies that exhibit characteristics of both storage and main memory: They combine the non-volatility, density, and economic characteristics of storage (e.g., flash memory) with the byte-addressability and a latency close to that of main memory (e.g., DRAM). SCM technologies will exhibit asymmetric latencies, with writes being noticeably slower than reads, and limited write endurance (although SCM may be significantly more durable than flash memory, e.g., 3D XPoint [1] by 3 orders of magnitude). Moreover, SCM will be denser than DRAM, yielding larger memory capacities. Finally, in contrast to DRAM that constantly consumes energy to refresh its state, idle SCM does not consume energy – only active cells do. Hence, SCM has the potential to lift the scalability issues of DRAM, both in terms of capacity and energy consumption.

In a previous tutorial, Stratis D. Viglas [37] surveyed extensively research works on SCM within the database community, which touched upon optimizing database components, file systems, recovery techniques, and specific database algorithms (e.g., sorts and joins) for the unique characteristics of SCM. In this tutorial, however, we demonstrate how to build software systems that use SCM as a universal memory. This tutorial offers comprehensive coverage of the software development process, starting from understanding the challenges and the opportunities brought by SCM, through designing and implementing SCM-based memory management and data structures, up to testing correctness using suitable frameworks, and evaluating end-to-end performance of SCM-enabled software components using different emulation techniques. Since SCM will be slower than DRAM, we assume in this tutorial hybrid systems that embed both SCM and DRAM.

SCM is emerging as a memory technology with long term prospects. However, potential research efforts may be inhibited by the fact that SCM relies on a novel programming paradigm that has yet to mature, and that hardware is currently not available. With this tutorial we aim at demystifying SCM programming and foster efforts to evolve existing systems to leverage SCM (e.g., Peleton [14]), as well as to explore new system architectures that were not conceivable before (e.g., SOFORT [33], FOEDUS [24]). The tutorial will follow the outline below.

2. SCM PROGRAMMING CHALLENGES

While SCM brings unprecedented opportunities as a potential universal memory, it fulfills the *no free lunch* folklore conjecture and raises unprecedented challenges as well. To store data, software has traditionally used block-addressable devices, managed by a file system and accessed through main memory. The programmer held full control over when data is persisted and the file system took care of handling partial writes, leakage problems, and storage fragmentation. With SCM it becomes possible to access, read, modify, and persist

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4197-4/17/05. . . $15.00
DOI: http://dx.doi.org/10.1145/3035918.3054777

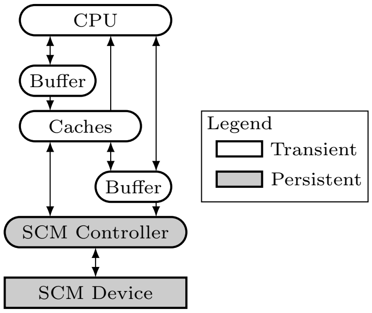

Figure 1: Volatility chain in x86-like processors.

data using load and store instructions at a CPU cache line granularity. The journey from CPU registers to SCM is long and mostly volatile, as illustrated in Figure 1, including store buffers and CPU caches, leaving the programmer with little control over when data is persisted. Worse, compilers and CPUs might speculatively reorder writes; for instance, non-temporal stores are *not* strongly ordered on x86 Intel CPUs and require a memory barrier [6]. Therefore, there is a need to enforce the order and durability of SCM writes sooner than later, often in a synchronous way. In this part of the tutorial, we will elaborate on the architecture specifics of modern processors and pinpoint when the store order and/or durability should be enforced. Thereafter, we will explore in depth the challenges that stem from using SCM as a universal memory, namely, data recovery, partial writes, and persistent memory leaks. Additionally we will elaborate on the challenge of persistent memory fragmentation. While file systems face similar challenges, their solutions do no apply to SCM, since the former benefit from the duality of virtual, memory-resident pages and persistent, disk-resident pages.

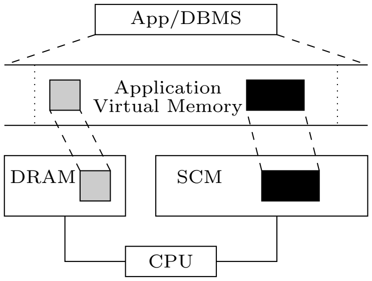

Figure 2: SCM can be mapped directly in the address space of the application.

There seems to be a consensus that SCM will be managed by a special file system that provides zero-copy memory mapping [9], i.e., memory mapping that bypasses the operating system page cache and provides the application layer with direct access to SCM, as illustrated in Figure 2. Several such file systems have already been proposed [21, 40, 23, 41, 29], and recent Linux kernels already embed such a functionality through the new Direct Access (DAX) feature [3] currently supported by the ext4 file system [4]. However, the Linux page table will be a major roadblock for the integration of SCM in large-scale systems for the following reasons:

- The Linux page table currently implements concurrency using a global lock, which makes memory reclamation upon shutdown of a multi-terabyte main memory process very slow, although SCM pages do not need to be reclaimed as they do not correspond to any DRAM memory. Worse, both DRAM and SCM page entries are scanned to reclaim the memory of the DRAM ones. For example, our measurements show that it takes up to 15 min to reclaim 10 TByte of memory upon process termination.
- Memory mapping millions of files will stress the (lack) of scalability of the Linux page table and the file system indexing [11], significantly slowing down startup time of large-scale main memory systems.
- Current Linux kernels support only up to 128 TBytes of address space, which will be insufficient since SCM will enable main memory capacities well beyond 100 TBytes. Moreover, systems approaching that limit will face the issue of *address space fragmentation*, i.e., the lack of sufficient contiguous virtual memory for memory mapping operations.

We will elaborate on how this issue will affect large-scale systems and will comment on potential solutions, such as modifying the operating system or “reverting” to page-based memory management.

3. SCM PROGRAMMING MODELS

To address the challenges discussed in the previous section, state-of-the-art works have followed mainly two approaches. The first one strives to provide *global* solutions in the form of software libraries, mostly following a transactional-memory-like approach, with the goal of making programming against SCM easy and accessible for programmers. Among early works, Volos et al. [39] proposed Mnemosyne, a library collection to program SCM that require kernel modifications and compiler support. Following a similar approach, Coburn et al. [20] proposed NV-Heap, a persistent heap that implements garbage collection through reference counting. Later, Chakrabarti et al. [16] and Chatzistergiou et al. [17] proposed respectively Atlas and REWIND, two log-based user-mode libraries that manage persistent data structures in SCM in a recoverable state using undo/redo logging. Most recently, Avni et al. [15] proposed PHyTM, a persistent hybrid transactional memory system that leverages hardware transactional memory, while assuming hardware support for single-bit atomic commit to SCM.

The second approach, that we denote as the *raw* approach, relies on existing CPU persistence primitives, namely cache line flushing instructions (CLFLUSH, CLFLUSHOPT, and CLWB), memory barriers (MFENCE and SFENCE), and non-temporal stores, to enforce consistency and durability. Moreover, it employs persistent pointers that comprise a base (e.g., file ID) and an offset within that base. An open question, however, is whether to use one large SCM file (pool) or multiple ones. We will elaborate on the advantages and drawbacks of each. For instance, a major advantage of using multiple files is the ability to defragment persistent memory by leveraging the *hole punch* feature of sparse files, while single-pool approaches can abstract away the base part of persistent pointers, as it is unique.

Intel's NVML [8] offers an elaborate collection of libraries that spans both the raw and the transactional approaches. Additionally, it offers a middle-ground approach based on the raw one augmented with useful transaction support, such as atomic execution of object constructors upon persistent memory allocation, which greatly simplifies fail-safe atomicity management.

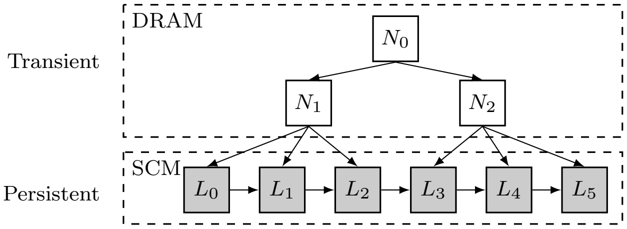

Figure 3: Illustration of a hybrid B-$^+$Tree: inner nodes are kept in DRAM while leaf nodes are kept in SCM [32].

We will discuss both approaches, highlighting their respective advantages and drawbacks. For instance, the transactional approach might hinder performance as it systematically logs modified data, which cost is amplified by the higher latency of SCM, while the raw approach offers great optimization opportunities. Conversely, compared to the raw approach, the transactional one is less prone to programming errors and is accessible to a larger number of programmers, as it abstract away the architectural knowledge and the *manual* fail-safe atomicity management that the raw approach requires. In the rest of the tutorial, we will assume the raw approach as we believe it is the most suitable for implementing high-performance SCM-based software.

4. SCM-BASED DATA STRUCTURES

Data structures are traditionally made durable using undo-redo logging and shadowing techniques. The rise of flash memory led to the emergence of novel optimized data structures, such as the Bw-Tree [26]. However, these remain intrinsically tied to the logging and paging mechanisms, which SCM can do without. The byte-addressability and the asymmetric latency of SCM impose new data structure design goals. For instance, a novel design goal is reducing the number of expensive writes, e.g., by trading writes for more reads [36]. Another design goal is byte-addressable data durability and consistency. In that context, Venkataraman et al. [35] proposed the CDDS B-Tree, a persistent and concurrent B-Tree that relies on versioning to achieve consistency. It recovers from failures by retrieving the version number of the latest consistent version and removing changes that were made past that version. However, its scalability suffers from using a global version number, and it requires garbage collection to remove old versions.

Chen et al. [18] proposed to use unsorted nodes with bitmaps to decrease the number of expensive writes to SCM. They extended their work by proposing the write-atomic B-Tree (wBTree) [19], a persistent tree that relies on the atomic update of the bitmap for consistency, and on undo-redo logs for more complex operations, such as node splits. It employs sorted indirection slot arrays in nodes to enable binary (instead of linear) search. Following another approach, Yang et al. [42] proposed the NV-Tree, a persistent and concurrent B-Tree based on the CSB-Tree [34]. They proposed to enforce the consistency of leaf nodes while relaxing that of inner nodes, and rebuilding them in case of a failure, which greatly simplifies durability management. The NV-Tree keeps inner nodes contiguous in memory and uses unsorted leaves with an append-only strategy. This design implies the need for costly rebuilds when a leaf parent node overflows, and leads to a high memory footprint.

Oukid et al. [32] proposed the FPTree, a hybrid SCM-DRAM persistent and concurrent B-Tree. It extends the relaxed consistency idea of the NV-Tree by placing inner nodes in DRAM, as illustrated in Figure 3, for better performance, and leveraging hash-based fingerprinting in leaf nodes as a means to significantly speedup point queries. Besides, it employs a combination of hardware transactional memory and fine-grained locking to achieve high concurrency. Nevertheless, some of these approaches are oblivious to the problem of persistent memory leaks. Indeed, the wBTree and the NV-Tree do not log the memory reference of a newly allocated or deallocated leaf, which makes these allocations prone to persistent memory leaks. In summary, we will revisit the main proposed techniques for SCM-based data structure design, highlight the pitfalls with regard to SCM challenges, and demonstrate how to avoid them.

5. TESTING OF SCM-BASED SOFTWARE

Although they share the same goals, crash-safety testing for disk-based and SCM-based software are different in that they have to address radically different failure scenarios, such as missing or misplaced persistence primitives. Consistency and recovery testing of SCM-based software did not get much attention so far. Lantz et al. [25] proposed Yat, a hypervisor-based off-line testing framework for SCM-based software. Yat is based on a record-and-replay approach. First, it records all SCM write operations by logging Virtual Machine Manager (VMM) exits that are caused by writes to SCM. Persistence primitive instructions need to be replaced in tested software by illegal instructions to make them traceable by causing a VMM exit. Yat then divides the memory trace into *segments*, delimited by two persistence barriers. It considers that SCM write operations can be reordered arbitrarily within a segment, with the exception of writes to the same cache line which are considered to be of fixed order. Yat replays the trace until a non-tested segment is encountered, then it runs the recovery procedure of the tested software for every possible reordering combination inside that segment. While it can approach comprehensive testing for single-threaded programs, it does so at the cost of a prohibitive testing time.

In contrast to Yat, Oukid et al. [31] propose an on-line automated testing framework, illustrated in Figure 4, that is non-invasive, requiring no software changes in most cases. It employs a suspend-test-resume approach and simulates power failures using data replication, similar to shadow memory testing approaches [27]. The testing framework creates a mirror file for each file that the program creates; the mirror file contains only data that is explicitly flushed by the program. The testing framework triggers randomly simulated crashes in the path of persistence primitives, upon which a test process is forked and the main process is suspended. Then, the test process executes a user-defined program that recovers using the mirror files with copy-on-write access semantics. Upon completion of the user-defined program, the test process is terminated and the changes to the mirror files are discarded. Finally, the main process resumes normal execution. While this approach covers only partially memory-reordering-related errors, this testing framework achieves fast code coverage by leveraging call stack information to limit duplicate testing, and is able to automate crash testing inside the recovery procedure of a program, both of which Yat does

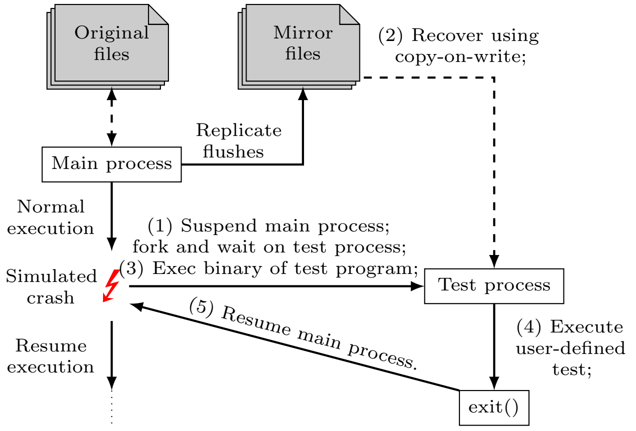
Original files
Mirror files
(2) Recover using copy-on-write;
Replicate flushes
Main process
Normal execution
(1) Suspend main process; fork and wait on test process;
Simulated crash
(3) Exec binary of test program;
Test process
(4) Execute user-defined test;
(5) Resume main process.
Resume execution
exit()

Figure 4: Illustration of automated crash simulation in a suspend-test-resume testing framework [31].

not provide. We will explain how to combine this testing framework with Yat to make up for the limitations of both.

Besides, development efforts to add SCM support to existing debugging frameworks are starting to emerge. For instance, Intel's NVML [8] provides a *Valgrind* extension [13] that validates the correctness of stores made to SCM. This work, however, is in its prototypical phase and its internals are not publicly documented.

We will argue, with supportive examples, that providing comprehensive consistency and recovery testing for large SCM-based software is practically infeasible. Instead, improving the *quality* of such software by covering a wide range of SCM-related errors in a reasonable amount of time is possible. Similarly to multi-threaded software, we argue that providing theoretical correctness guarantees should be a prerequisite for any SCM-based software, and that experimental testing should not (and cannot) make up for the lack of such theoretical guarantees. Hence, design simplicity is key for successful large SCM-based software.

6. SCM EMULATION TECHNIQUES

Researchers are often reluctant to dive into systems research on SCM due to the lack of real hardware. To a certain extent, their concerns are justified as several SCM performance characteristics and their interplay with modern CPU architectures are yet to be seen. Nevertheless, if we assume the DIMM form factor and an access interface similar to that of DRAM, we can emulate SCM latency and bandwidth in different ways, such as:

- Use Non-Uniform Memory Access (NUMA) as a means to emulate a higher latency and a lower bandwidth – the number of QPI hops determines these metrics. This can be achieved using a NUMA-aware allocator (e.g., using *libnuma*) or simply using the Linux *numactl* command [2].
- Use a DRAM-based *tmpfs* mount; this approach can be combined with the NUMA one by configuring *tmpfs* to use the memory of a specific socket [12].
- Reserve a DRAM region upon boot time and mount an SCM-aware file system on it (e.g., ext4 DAX) [5]. This memory region will be invisible to the operating system and will be treated as an SCM device, which contrasts with the *tmpfs* approach where the operating system still manages the memory.

These techniques are easy to use and require no particular effort. On the side of elaborate techniques, Intel has made available an SCM emulation platform to several academic and industry partners, and many publications have already used it [23, 30, 14]. It uses a custom BIOS and microcode to insert stalls on each access to the DRAM-based emulated SCM. A full description of its latency and bandwidth modeling techniques is publicly available [10]. Finally, HP has recently open-sourced Quartz [38], a DRAM-based SCM performance emulator that leverages existing hardware features to emulate different latency and bandwidth characteristics. We will emphasize techniques that the attendees can seamlessly replicate for their own work.

7. CONCLUSION

SCM is emerging as a disruptive memory technology that might have the same level of impact as high core counts and large main-memory capacities, requiring us to rethink current database system architectures. However, research on SCM is currently hindered by the absence of hardware and the lack of maturity of its programming model. In this tutorial, rather than encyclopedic coverage, we strive to provide a comprehensive tool set for designing, implementing, testing the correctness, and evaluating the performance of SCM-based software systems. By sharing our experience in designing and implementing durable and consistent systems fully tailored to SCM, we hope to foster research on novel system architectures that leverage the full potential of SCM.

Looking ahead, we envision that the next evolution of hybrid SCM-DRAM data structures will be morphing data structures that can dynamically adjust their data placement between SCM and DRAM, following performance and memory constraints. Moreover, log-structured data structures, such as the Log-Structured Merge Tree [28], are natural candidates for SCM given their append-only nature and their wide use in key-value databases; we expect to see more research in that field.

Finally, similar to multi-threaded programming, SCM programming will need a set of mature tools and programming models to be adopted by a larger audience. To that end, there has to be a tighter integration of SCM-related functionalities in existing tools (e.g., *Valgrind*). Besides, there is a need for new frameworks that are able to identify complex errors, such as wrong recovery logic, and to identify optimization opportunities, such as duplicate flushes.

Acknowledgments

This work is partially funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (Resilience Path).

Biographies of the Presenters

Ismail Oukid is a fourth year PhD student at TU Dresden, working in close collaboration with SAP SE and Intel. His PhD work consists in conducting pathfinding work for next-generation databases on SCM, which led him to design a novel single-level transactional storage engine, called SOFORT. Through his work he gained in-depth insights into the challenges of SCM and investigated data structure design, memory management techniques, transaction concurrency control, and fail-safety testing techniques for SCM. He received an Engineering degree (eq. MSc) from the Grenoble Institute of Technology (Grenoble INP – Ensimag) in 2013.

Wolfgang Lehner is full professor and head of the database systems group at TU Dresden, Germany. His research is dedicated to database system architecture specifically looking at hardware-related aspects in main-memory centric settings. He is part of TU Dresden's excellence cluster with research topics in energy-aware computing, resilient data structures on unreliable hardware, and orchestration of wildly heterogeneous systems. He is also a principal investigator with the DFG-funded CRC on "Highly Adaptive Energy Efficient Computing" as well as Germany's national "Competence Center for Scalable Data Services and Solutions" (ScaDS). Wolfgang also maintains a close research relationship with the SAP HANA development team in Walldorf, Seoul, and Waterloo. He serves the community in many PCs, is an elected member of the VLDB Endowment, serves on the review board of the German Research Foundation (DFG), and is an appointed member of the Academy of Europe. More info at: http://wwwdb.inf.tu-dresden.de/lehner.

8. REFERENCES

[1] 3D XPoint Technology. https://www.micron.com/about/our-innovation/3d-xpoint-technology.

[2] Control NUMA policy for processes or shared memory. https://linux.die.net/man/8/numactl.

[3] Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[4] Ext4 file system. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[5] How to emulate persistent memory. http://pmem.io/2016/02/22/pm-emulation.html.

[6] Intel®Architecture Instruction Set Extensions Programming Reference. http://software.intel.com/en-us/intel-isa-extensions.

[7] International Technology Roadmap for Semiconductors 2.0, Beyond CMOS, 2015. http://www.semiconductors.org/clientuploads/Research_Technology/ITRS/2015/6_2015%20ITRS%202.0%20Beyond%20CMOS.pdf.

[8] NVML Library. http://pmem.io/nvml/.

[9] SNIA NVM Programming Model V1.1. http://www.snia.org/sites/default/files/NVMProgrammingModel_v1.1.pdf.

[10] SR. Dulloor. *Systems and Applications for Persistent Memory.* PhD Thesis, 2016. https://smartech.gatech.edu/bitstream/handle/1853/54396/DULLOOR-DISSERTATION-2015.pdf.

[11] Supporting file systems in persistent memory. https://lwn.net/Articles/610174/.

[12] Tmpfs file system. https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt.

[13] Valgrind extension for persistent memory. https://github.com/pmem/valgrind.

[14] J. Arulraj, M. Perron, and A. Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.

[15] H. Avni and T. Brown. Persistent hybrid transactional memory for databases. *PVLDB*, 10(4):409–420, 2016.

[16] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.

[17] A. Chatzistergiou, M. Cintra, and S. D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5):497–508, 2015.

[18] S. Chen, P. B. Gibbons, and S. Nath. Rethinking Database Algorithms for Phase Change Memory. In *CIDR*, pages 21–31, 2011.

[19] S. Chen and Q. Jin. Persistent B+-trees in Non-volatile Main Memory. *PVLDB*, 8(7):786–797, 2015.

[20] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Sigplan Notices*, 46(3):105–118, 2011.

[21] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP*, pages 133–146. ACM, 2009.

[22] M. Dayarathna, Y. Wen, and R. Fan. Data Center Energy Consumption Modeling: A Survey. *IEEE Communications Surveys Tutorials*, 18(1):732–794, Firstquarter 2016.

[23] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, page 15. ACM, 2014.

[24] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *SIGMOD*, pages 691–706. ACM, 2015.

[25] P. Lantz, D. S. Rao, S. Kumar, R. Sankaran, and J. Jackson. Yat: A Validation Framework for Persistent Memory Software. In *USENIX Annual Technical Conference*, pages 433–438, 2014.

[26] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*, pages 302–313. IEEE, 2013.

[27] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[28] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[29] J. Ou, J. Shu, and Y. Lu. A high performance file system for non-volatile main memory. In *EuroSys*, page 12. ACM, 2016.

[30] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. In *DaMoN*, page 8. ACM, 2014.

[31] I. Oukid, D. Booss, A. Lespinasse, and W. Lehner. On testing persistent-memory-based software. In *DaMoN*, page 5. ACM, 2016.

[32] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*, pages 371–386. ACM, 2016.

[33] I. Oukid, W. Lehner, T. Kissinger, T. Willhalm, and P. Bumbulis. Instant recovery for main-memory databases. In *CIDR*, 2015.

[34] J. Rao and K. A. Ross. Making B+-trees cache conscious in main memory. In *SIGMOD*, pages 475–486. ACM, 2000.

[35] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST*, volume 11, pages 61–75. USENIX Association, 2011.

[36] S. D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5):413–424, 2014.

[37] S. D. Viglas. Data management in non-volatile memory. In *SIGMOD*, pages 1707–1711. ACM, 2015.

[38] H. Volos, G. Magalhaes, L. Cherkasova, and J. Li. Quartz: A Lightweight Performance Emulator for Persistent Memory Software. In *Proceedings of the 16th Annual Middleware Conference*, pages 37–49. ACM, 2015.

[39] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[40] X. Wu and A. Reddy. SCMFS: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 39. ACM, 2011.

[41] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST*, pages 323–338. USENIX, 2016.

[42] J. Yang, Q. Wei, C. Wang, C. Chen, K. L. Yong, and B. He. NV-Tree: A Consistent and Workload-Adaptive Tree Structure for Non-Volatile Memory. *IEEE Transactions on Computers*, 65(7):2169–2183, 2016.