Data Management in Non-Volatile Memory

Stratis D. Viglas
School of Informatics
University of Edinburgh, UK
sviglas@inf.ed.ac.uk

ABSTRACT

Non-volatile memory promises to bridge the gap between main memory and secondary storage by offering a universal storage device. Its performance profile is unique in that its latency is close to main memory and it is byte addressable, but it exhibits asymmetric I/O in that writes are more expensive than reads. These properties imply that it cannot act as a drop-in replacement for either main-memory or disk. Therefore, we must revisit the salient aspects of data management in light of this new technology. In what follows we present the current work in the area with a view towards identifying the open problems and exposing the research opportunities. In particular, we address issues like: (*a*) incorporating non-volatile memory into the data management stack, (*b*) supporting transactions and ensuring persistence and recovery, and (*c*) query processing.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems; D.4.2 [**Storage Management**]: Secondary storage

Keywords

Non-volatile memory; Persistence; Performance; Recovery; Query processing

1. INTRODUCTION

Non-volatile memory (NVM), also referred to as persistent memory, is a new type of storage medium that aims to bring forth, for the first time, a universal storage device. That is, a device that bridges the gap between volatile main memory and non-volatile, block-based secondary storage. Its performance characteristics are such that it is close to main memory (DRAM) in terms of access latency—albeit slower—but much faster than flash memory and magnetic disks (see also Figure 1 for a comparison). The key properties of non-volatile memory, apart from its performance and persistence, are byte-addressability and write-read cost asymmetry. Byte-addressability means that NVM is not a block-storage device and it is certainly feasible, depending on how vendors choose to expose it to the system, for it to be accessible through memory load/store operations. Likewise, we can perform byte-level I/O as opposed to block-level I/O without having to amortize the I/O cost by increasing the block size. At the same time, NVM exhibits some of the pathologies of flash memory: writes are more expensive than reads both in terms of response time (*i.e.*, they simply need longer to be completed) and in terms of how they affect the lifetime of the medium. Note, however, that the physical mechanisms used in contemporary NVM designs are not the same as those in flash memory [2, 22]—see also Section 2.1. As such, NVM requires new techniques for masking this asymmetry from the users.

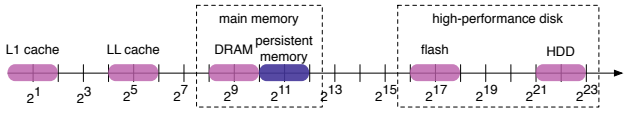

Figure 1: Typical access latency in processor cycles on a 4GHz processor (figure adapted from [33])

Given the above, the question is how do we expose NVM to developers and users? Treating persistent memory as persistent storage means that we will be downplaying its performance merits and using algorithms and techniques that have been designed for a different medium. Treating persistent memory as volatile memory means that we will need to rethink our data structures in light of the new available capacity and assume that everything can be memory-resident. In both alternatives, by treating persistent memory as yet another persistent storage device or yet another additional volatile memory device, we are effectively reusing a prior abstraction that has been originally built for different performance characteristics. In doing that, we are missing an opportunity to bring out the best of each abstraction and drop all the bloat that it may be carrying with it.

Objectives In this work, our aims are: (*a*) to present this new type of storage technology in more detail and focus on what makes it different and why it requires a study of its own; (*b*) to give insight into how this technology can be seamlessly integrated into the data management stack; (*c*) to give an overview of the recent work in the area and explain the key techniques in more detail; (*d*) to identify both the immediate research opportunities and the longer-term fundamental problems that require attention; and (*e*) to jumpstart researchers and practitioners in the area.

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 ACM 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'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2731082.

2. BACKGROUND

NVM is a class of memory, rather than a specific type of storage medium. There are many competing implementations that are classified as NVM even though they do not necessarily share the same underlying technology. We will first present an overview of the different technologies that have been proposed in the literature and have picked up significant traction to be considered as the forerunners when NVM becomes mainstream. After addressing the technological side of NVM we will look at how it can be introduced into the system stack, *i.e.*, we will present ways that existing systems can immediately start using NVM as a drop-in replacement for either DRAM or for secondary storage. We will next turn our attention to the use-cases that are most relevant to data management. We will start with the most basic guarantee that NVM provides: persistence. Persistence is guaranteed by the medium, but only at the level of ensuring that data is not lost if power is lost. There are various aspects that need to be addressed when higher-level workflows like database systems are taken into account. Immediately following persistence is recovery. Not only do we want to ensure long-term access to the data, we also need to have sufficient mechanisms in place to allow us to recover portions of the database in the event of a power failure. This is a departure from the typical database architecture where all processing takes place in a DRAM-backed bufferpool and I/O is performed in blocks. The runtime must now guarantee the recoverability of each individual byte change, as NVM is byte-addressable. A repercussion of different persistent and recovery mechanisms is that we need to develop new mechanisms for transactional support as well. Finally, we will address query processing. We will discuss how the fundamentals of query processing need to be revisited in the context of NVM. In the last part of the presentation we will focus on the open challenges in the area with an eye towards identifying future research directions. A more detailed discussion of the various topics follows.

2.1 The inner workings

NVM is an umbrella term that encompasses both read-only memories and persistent random-access memories. The most widely used underlying technology used in NVMs is the floating-gate transistor, which acts as a single memory cell. It works by having the device controller apply device-specific operating voltages to the gate to trap or release electrical charge. The recorded value can then be read by sensing the charge (no charge equals 0, charge equals 1). This technology forms the basis of various implementations like the (Electrical) Erasable Programmable Read Only-Memory (EEPROM), or the most widely used flash memory. In flash memory, multiple cells are blocked and are not individually set. Operations take place at the granularity of an entire block, which is erased and reprogrammed (also referred to as the the erase-before-write property of flash memory). Floating-gate transistors can only go through a limited number of erase/program cycles, thus reducing the lifetime of the device. Limited endurance and block-level I/O, combined with the physical property of setting voltage taking longer than sensing, result in writing to flash memory being much slower than reading. The key difference between flash memory and the wider class of NVMs is the method by which information is stored and sensed: rather than storing and sensing charge, NVM technologies alter some physical property of the medium thereby changing its electrical resistance, which they then sense to read back the information.

Phase-Change Memory, or PCM, is the most widely known NVM technology. Information is recorded using a mechanism similar to that of optical media. A heating element heats the silicone of each memory cell, either changing it to an amorphous state, or crystallizing it. The two different states, or phases, have different electrical resistance, thus recording one bit. An electrical current is applied to measure the electrical resistance and sense the state of each cell. There are various other phases in between an amorphous and a completely crystallized state. Thus, a single cell can record more than one bit of information. The main merit of PCM is its high density. Its main disadvantage is that it is not a purely electrical device as it requires a heating element for programming and thus it is more power-hungry and dissipates more heat during operation. Additionally, there is considerable voltage drift due to the resistance of the amorphous state increasing over time according to a power law. So, as times goes by, it becomes increasingly difficult for the device to store multiple bits in a single cell, thus losing capacity. Therefore, even though the technology is capable of high density in the offset, this capability is lost over time.

Resistive Random Access Memory, or ReRAM, works in a similar way to PCM. The main difference is that instead of affecting the crystallization of the medium, a heating element physically opens a conduction path on the material for the current to pass. That physical path can be closed on demand. The existence (or not) of the physical path affects the electrical resistance of the material, which can thus be used to record information. As electrical current passing through the material, in addition to sensing the electrical resistance, is used to read the stored information.

Magnetoresistive Random Access Memory, or MRAM, employs the magnetic properties of the medium in addition to its electrical ones. The memory cells are magnetic storage elements formed by two magnetic plates separated by a thin insulating layer, each capable of holding a magnetic field. One of the plates is a permanent magnet, whereas the other plate's field can be set at will. The two fields generate a magnetic tunnel, whose electrical resistance changes depending on the orientation of the two fields. The electrical resistance can be measured if current is supplied to the two plates. The polarity of the fields of the plates results in two levels of resistance that can record one bit of information.

All these technologies support byte-addressable fine-grained control of the medium and alleviate some of the performance problems associated with the large-scale erase-to-program operations of flash memory. However, setting the physical properties takes as long a time and requires a heating element (for PCM and ReRAM) or polarization of the medium (for MRAM); in all cases, energy consumption is greater. Thus, the problem has only been alleviated and writes still hurt performance more than reads, whether the performance metric is response time, energy consumption, or device lifetime. All vendors aim to produce NVM at high densities, and thus capacities, but at a much lower energy consumption point, thereby making NVM cost-effective.

2.2 Integrating NVM into the stack

NVM presents a new level in the memory hierarchy so a salient decision is how to best integrate it with the rest of the system stack. There are two main options: the first

option is to treat NVM as secondary storage and use file I/O operations to access it. If that is the case, we need to optimize the I/O substrate of the system for persistent memory, as opposed to optimizing the memory subsystem for persistence. The second option is to treat NVM as byte-addressable DRAM with persistence guarantees through persistent memory regions. We then need to extend the system to support NVM through either: (*a*) changes to the memory controller so that it can guarantee persistence through its firmware; or (*b*) extending the system's instruction set architecture to expose persistent memory to the developer through specialized memory load/store operations; or (*c*) exposing a higher-level API to the operating system and then relying on software to guarantee persistence and to decide on data placement. These alternatives can be refined further into four main designs that have been used in practice.

RAM disk The first approach is to employ a memory-mounted filesystem over NVM. RAM disks are lightweight filesystems that bypass disk-related overheads. They do not incur disk I/O, though they support persistence. RAM disks bypass the filesystem cache with writes and reads being synchronous to the portion of main memory allocated to the RAM disk. Therefore, RAM disks bridge the mismatch between block devices and byte-addressable NVM by employing filesystem practices to manage NVM.

Byte-addressable filesystem The second option is to use a filesystem optimized for persistent memory such as PMFS [13] or BPFS [9]. These are typically implemented as kernel extensions and thus reduce the overhead of system calls and employ low-level fine-grained persistence primitives to implement file-level access through CPU load/store instructions, thereby minimizing overhead.

Heap-based approaches The third option is to substitute the runtime's memory allocator (*e.g.*, `malloc()`) with one that uses the non-volatile memory for allocations, as opposed to the system's heap (see, *e.g.*, [8, 40] for two approaches). This affects the memory allocator, but not the way by which data structures allocate memory. In applying this approach we use main memory data structures implemented for volatile memory. Thus, we may be using patterns that are not optimized for NVM. Consider, for instance, a dynamic array like a C++ `vector`. C++ `vector`s have an initial capacity; when that capacity is reached they allocate a memory chunk twice as big as their current capacity; copy the elements over; and release the memory they had previously occupied. The doubling of allocated memory and, more importantly, the copying of elements over are far from ideal for NVM as they incur a large number of writes.

Optimized blocked memory Finally, we can use a combination of the previous options. We can keep the interface of a heap-based allocator, but change the implementation of our data structures so they use lightweight logging approaches to record changes to NVM and export a byte addressable interface to the user. Such an approach has been proven to decouple the persistence guarantees from the implementation of data structures and the memory allocator and incurs a minimal overhead [39].

2.3 Persistence

Persistent regions have been proposed to support persistent virtual memory and user-accessible heaps [17, 34, 44]. These attempts either provide persistence guarantees for heap-allocated data, or employ a mapping to block-level I/O devices and file abstractions to implement persistence. Recoverability relies on staging persistence and logging through combining volatile main memory and persistent disk storage. In [25], the authors use battery-backed DRAM for persisting the file cache [5]. They also invoke disk-based I/O and uses a coarse-grained region approach to log undo information.

Two recent proposals [8, 40] provide implementations of heaps for persistent memory to user applications. Such support can act as a drop-in replacement for `malloc()` and enables the developers to seamlessly integrate persistent memory regions into their applications. Moreover, heaps in NVM can be used as building blocks for programmers to create and manage their own recovery protocols. Fang *et al.* [14] propose an NVM-based log manager for DBMSs, which relies on a client-server design and uses epoch barriers to guarantee persistence. Giles *et al.* [15] address embedded transaction management in user code. They introduce custom hardware to force parts of the log to NVM before committing, while keeping user updates in a dedicated buffer before persisting the log. Other recent work [16] explicitly addresses redo logging without in-place updates. Similarly, [47] embeds transaction management in user code by assuming the existence of a non-volatile cache that is used as a drop-in replacement for a log. Finally, [3] studies the update semantics of NVM data in lock-based code (as opposed to transactional code) and the mechanisms used for logging and recovery in NVM.

2.4 Recovery

Prior to NVM, researchers proposed battery-backed DRAM and buffer manager extensions to support recoverability [28]. For instance, [10] uses battery-backed DRAM with an ARIES-like protocol, and assumes page-level I/O for data and log updates. DBMSs optimized for volatile memory [11, 19] can be used as starting points for integrating persistent memory, even though they are sometimes subject to the inefficiencies of a block-based design towards durability.

Pelley *et al.* [32] address the problem of supporting durability for OLTP workloads executed over NVM. They first show that NVM can act as a drop-in replacement for disk drives and gives almost instantaneous recovery in the event of failure. However, this comes at the price of a reduction in throughput during transactional processing. They identify the barrier latency in NVM as one of the main culprits for this discrepancy, and introduce distributed logging and group commits for mitigating the impact of suboptimal barrier implementations. Similarly, [41] examines more closely the impact of NVM and non-uniform memory access on multicore and multi-socket architectures. The authors argue that distributed logging is not elected but enforced by the environment and propose passive group commit as an implementation mechanism for distributed logging. Passive group commit exhibits a minimal overhead and enables distributed commit to scale well in logging-intensive workloads. Along the same lines, Oukid *et al.* [30] proposed a framework by which NVM is combined with DRAM for recovery. The insight is that updates should happen in-place on NVM but in small batches to better take advantage of the performance profile of NVM. Doing so results in almost instantaneous restart in the event of failure.

Logging and recovery for NVM is also addressed in [4] where the authors present REWIND: a user-mode library approach to managing transactional updates directly from user code written in an imperative general-purpose language.

REWIND starts from write-ahead logging and adapts it for NVM. It relies on a custom persistent in-memory data structure for the log that supports recoverable operations on itself. The scheme also employs a combination of batching in the memory layout of the log data, non-temporal updates, persistent memory fences, and lightweight logging to optimize performance. The authors show that such an optimized logging scheme can be orders of magnitude faster than disk-based approaches ported to persistent memory, or than data-structure-specific recoverability extensions.

Other work has considered more general data management tasks. For instance, [35] compares both DBMSs and file systems to custom alternatives; while [18] quantifies the overhead of several DBMS functionalities. There has also been recent interest in extending file systems for NVM. For example, [31] presents an extended transactional and recoverable I/O interface for multiple, non-consecutive blocks.

2.5 Query processing

There has been a large body of work dealing with query processing but in the context of flash memory. Research has focused on flash-specific bufferpool management schemes [20, 21, 29, 45], query evaluation techniques [12, 27, 36], or indexing [23, 24, 38, 42, 43, 46]. Though this host of work addresses the write-read asymmetry of the medium, it does not cater for byte addressability. The differences in block- *vs.* byte-level access suggest that considerable effort will be necessary to port these approaches to persistent memory.

Chen *et al.* [6] explored how database algorithms need to be changed in the presence of phase-change memory. The authors argued for a radical reimplementation of algorithms by eliminating data copying and using pointers to data in order to reduce memory stores. This resulted in substantial benefits both in terms of performance and in terms of lifetime and power consumption of the device.

A strand of work tackles algorithms for query processing, when these algorithms work over data stored in NVM. For example, [37] describes the necessary optimizations if sorting is to be applied over data in PCM. The authors propose and evaluate mechanisms for using a small DRAM buffer ahead of PCM to optimize performance. Similarly, [39] introduces write-limited algorithms for fundamental query processing operations. These are families of algorithms for sorting and join processing that trade expensive writes for cheaper reads with the goal of achieving the same performance as traditional query processing algorithms, but at a fraction of the write cost. Write-limited algorithms are accompanied by a runtime that dynamically tracks reads and writes and decides whether intermediate results should be materialized, or deferred and regenerated on demand from their primary sources. Further, different ways of incorporating NVM into the stack are also explored, along the lines of Section 2.2. The choice of algorithm in addition to the choice of how to best access NVM from query processing code is a complicated problem that requires a carefully crafted cost model and runtime, if informed decisions are to be made.

Data structures have also been tackled in the context of NVM. In [7], the authors present three schemes to optimize B^+-trees for phase-change memory: keeping node entries unsorted, allowing underflow in nodes, or allowing overflow chains. All three methods aim to optimize write performance. The authors show that the three proposed schemes can reduce the response time and improve device lifetime and energy consumption. Similarly, [38] proposes the introduction of controlled imbalance in the B^+-tree to cope with asymmetric I/O. The observation is that if we know the write-to-read cost ratio of the device we can afford to introduce imbalance as that will manifest as extra reads, which are cheaper than writes. Once the savings from those extra reads are spent, we can rebalance the tree. The unbalanced B^+-tree exhibited performance improvements over the traditional B^+-tree in a variety of settings.

Finally, query optimization has been addressed in [1]. The authors argue that even though the optimization algorithm is robust, its cost models are suboptimal when dealing with asymmetric storage. The reason is that the big differentiating factor in performance so far has been random *vs.* sequential I/O. In asymmetric media, however, it is the number of write *vs.* the number of read operations that should be used as the basis for cost functions. The authors implement an asymmetry-aware cost model for PostgreSQL and show how it can lead to more efficient execution plans. In addition, they propose mechanisms by which the system can automatically tune the cost model to the host hardware. It would be interesting to extend and refine this work to account for the byte addressability of NVM. This can be further improved with main memory execution models that take the CPU's cache hierarchy into account [26].

3. OUTLOOK

Non-volatile memory has the potential to become a universal storage device that bridges the gap between main memory and secondary storage. Even though non-volatile memory is a general term encompassing a multitude of technologies, regardless of the specific type of technology the performance profile of non-volatile memory is unique. Non-volatile memory is asymmetric with writes being more expensive than reads, and it is also byte-addressable. The combination of these factors requires that all aspects of data management are revisited: from transactional processing, to persistence and recovery, to query processing. A related, but salient question, apart from the actual use of non-volatile memory for data management, is how to best integrate non-volatile memory in the data management stack at the system level. There has been some preliminary work on non-volatile memory, but the area is certainly fresh with plenty of opportunities for original and high-impact work.

4. REFERENCES

[1] D. Bausch, I. Petrov, and A. Buchmann. Making cost-based query optimization asymmetry-aware. In *DaMoN*, 2012.

[2] P. Bonnet, L. Bouganim, I. Koltsidas, and S. D. Viglas. System co-design and data managment for flash devices. *PVLDB*, 4, 2011.

[3] D. Chakrabarti and H.-J. Boehm. Durability semantics for lock-based multithreaded programs. In *HOTPAR*, June 2013.

[4] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8, 2015.

[5] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. In *ASPLOS*, 1996.

[6] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *CIDR*, 2011.

[7] P. Chi, W.-C. Lee, and Y. Xie. Making b+-tree efficient in pcm-based main memory. In *ISLPED*, 2014.

[8] 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. In *ASPLOS*, 2011.

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

[10] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In *VLDB*, 1989.

[11] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*, 2013.

[12] J. Do and J. M. Patel. Join processing for flash SSDs: remembering past lessons. In *DaMoN*, 2009.

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

[14] R. Fang, H.-I. Hsiao, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *ICDE*, 2011.

[15] E. Giles, K. Doshi, and P. Varman. Bridging the programming gap between persistent and volatile memory using WrAP. In *CF*, 2013.

[16] E. Giles, K. Doshi, and P. Varman. Software support for atomicity and persistence in non-volatile memory. In *MEAOW*, October 2013.

[17] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *ATC*, 2012.

[18] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2), 2008.

[20] H. Kim and S. Ahn. BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST*, 2008.

[21] I. Koltsidas and S. D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1(1), 2008.

[22] I. Koltsidas and S. D. Viglas. Data management over flash memory. In *SIGMOD*, 2011.

[23] X. Li *et al.* A new dynamic hash index for flash-based storage. In *WAIM*, 2008.

[24] Y. Li *et al.* Tree indexing on flash disks. In *ICDE*, 2009.

[25] D. E. Lowell and P. M. Chen. Free transactions with Rio vista. In *SOSP*, 1997.

[26] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB*, 2002.

[27] D. Myers. On the use of NAND flash memory in high-performance relational databases. MSc Thesis, MIT, 2007.

[28] W. T. Ng and P. M. Chen. Integrating reliable memory in databases. In *VLDB*, 1997.

[29] Y. Ou *et al.* CFDC: a flash-aware replacement policy for database buffer management. In *DAMON*, 2009.

[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*, 2014.

[31] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *HPCA*, 2011.

[32] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge. Storage management in the NVRAM era. *PVLDB*, 7(2), 2014.

[33] M. K. Qureshi, S. Gurumurthi, and B. Rajendran. *Phase Change Memory: from devices to systems.* Morgan & Claypool, 2012.

[34] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *SOSP*, 1993.

[35] R. Sears and E. Brewer. Stasis: Flexible transactional storage. In *OSDI*, 2006.

[36] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD*, 2009.

[37] M. Vamsikrishna, Z. Su, and K.-L. Tan. A write efficient pcm-aware sort. In *DEXA*, 2012.

[38] S. D. Viglas. Adapting the B+-tree for asymmetric I/O. In *ADBIS*, 2012.

[39] S. D. Viglas. Write-limited sorts and joins for persistent memory. *PVLDB*, 7(5), 2014.

[40] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[41] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10), 2014.

[42] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient R-tree implementation over flash-memory storage systems. In *GIS*, 2003.

[43] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An Efficient B-tree Layer Implementation for Flash-Memory Storage Systems. *Trans. on Embedded Computing Sys.*, 6(3), 2007.

[44] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *ASPLOS*, 1994.

[45] S. yeong Park, D. Jung, J. uk Kang, J. soo Kim, J. Lee, S. yeong Park, D. Jung, J. uk Kang, J. soo Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *CASES*, 2006.

[46] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, Dimitrios, and W. A. Najjar. Microhash: an efficient index structure for flash-based sensor devices. In *FAST*, 2005.

[47] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *MICRO*, 2013.