An NVM Carol

Margo Seltzer
John A. Paulson School of
Engineering and Applied Sciences
Harvard University
Cambridge, MA
Email: margo@eecs.harvard.edu

Virendra Marathe
Oracle Labs
Burlington, MA
Email: virendra.marathe@oracle.com

Steve Byan
Oracle Labs
Burlington, MA
Email: steve.byan@oracle.com

***Abstract*—Around 2010, we observed significant research activity around the development of non-volatile memory technologies. Shortly thereafter, other research communities began considering the implications of non-volatile memory on system design, from storage systems to data management solutions to entire systems. Finally, in July 2015, Intel and Micron Technology announced 3D XPoint. It's now 2018; Intel is shipping its technology in SSD packages, but we've not yet seen the widespread availability of byte-addressable non-volatile memory that resides on the memory bus.**

We can view non-volatile memory technology and its impact on systems through an historical lens revealing it as the convergence of several past research trends starting with the concept of single-level store, encompassing the 1980s excitement around bubble memory, building upon persistent object systems, and leveraging recent work in transactional memory. We present this historical context, recalling past ideas that seem particularly relevant and potentially applicable and highlighting aspects that are novel.

I. Introduction

In the 1960s we had bubble memory; today we are contemplating the emergence of a new form of non-volatile memory (NVM). Sixty years ago, some claimed that bubble memory was going to fundamentally change how we build systems; we hear the same predictions today. This suggests two fundamental questions: First, is this technology here to stay, or will it too go the way of bubble memory? Second, if things will turn out differently this time, what have we learned in the past 60 years that better prepares us for this brave new world?

In July 2015, Intel announced 3D XPoint (prounounced Crosspoint), a non-volatile memory technology with latency approaching that of DRAM and persistence characteristics similar to flash [31]. Touted as "game-changing," this was merely one in a long line of *storage class memories* (SCM), emerging technologies that combine the latency of main memory with the capacity and price of persistent storage.

Unlike prior persistent media, these new technologies offer *byte addressability*. Rather than requiring that data be marshalled into relatively large blocks, NVM permits accesses directly to individual bytes and typically transfers data to/from media in cache line size chunks, using normal memory system interfaces and mechanisms (e.g., loads and stores in the ISA, caches, and memory controllers). This byte addressability creates the opportunity to use identical data representations in both transient and persistent media, avoiding the overhead of translating back and forth between "on-disk" data structures and memory resident ones. However, it also introduces challenges: data can become persistent at any time – that is, while ISAs support instructions that cause data to move from on chip caches to the persistent media, there are no instructions that prevent data from moving to the persistent media. Therefore, updates that touch multiple memory locations can persist at any time, in any order, unless care is taken to prevent this. This combination of opportunity and challenge motivates much of the research in software for NVM.

A. The Technology

Intel and Micron have been downright secretive about the precise technology underlying 3D XPoint [56], [79]. However, with the release of 3D XPoint SSDs (i.e., Optane^{TM}), it seems that the cat is out of the bag, and the winner is PCM [17]. In addition to phase change memory (PCM) [69], there were speculations about other technologies, such as Resistive Ram (ReRAM) [18] and spin torque transfer magnetic RAM (STT-MRAM) [38], [39]. We can trace the roots of PCM back to fundamental research in the 1960s, which first demonstrated the switching effects of certain "disordered structures," containing tellurium, arsenic, silicon and germanium [67]. The salient property observed was, "After switching from a highly resistive state, structural changes result in the preservation of a conductive state even when the current is totally removed." In other words, applying the right current to the material induced a phase change that persisted even after the current was removed. Shortly thereafter, Charles Shi demonstrated fabrication of a device based on these principles [70]. It wasn't until 1999, almost thirty years later, PCM commercialization efforts began. Over the next decade, activity and excitement around PCM grew; by 2014, it seemed that PCM was on the verge of widespread availability and adoption.

Relative to PCM, Resistive RAM (ReRAM) is a much newer technology. The underlying technology relies on finding a material that is not prone to the irreversible dielectric breakdown when subjected to voltage high enough to allow it to conduct. In this case, the material sits between two electrodes and its resistivity changes as different volatages are applied to the electrodes. Perhaps the most famous of the

ReRAM technologies was HP Lab's memristor from 2006[1].

Spin torque transfer magnetic RAM (STT-MRAM) leverages the spin polarization found in ferromagnetic materials. Electrons possess both a charge and a spin. Traditional memory takes advantage of the former, by leveraging electron behavior in the the presence of an electrical field; spintronics leverages the latter, behavior in the presence of a magnetic field, in a manner similar to hard drives. Writing bits essentially maps to passing a spin-polarized current through a magnetic layer to change the spin of the electrons in the magnetic layer that stores the bits. Early versions of magnetic ram (MRAM) placed the magnetic tunnel junction (MTJ) inline with the silicon substrate. STT-MRAM places the MJT perpendicular to the substrate, resulting in denser and lower current circuits. Multiple companies made STT-MRAM based product announcements in early 2018 [59], [60], [61].

Carbon nanotubes (CNTs) are the most recent entry to the NVM market. They are exactly what they sound like, tiny cylinders of graphene on the order of a few nanometers in diameter, and are stronger than steel, half the density of aluminum, and capable of exhibiting useful electronic properties [25]. When laid out in a thin film, bit values are determined by the internal connectivity of an area of the film. If most of the tubes touch, the bit is one; else if most are separated, the bit is 0. Claims of DRAM performance and a 5nm size coupled with the technical simplicity, could make this technology an industry darling.

Table I provides a high level overview of the competing technologies.

Technology	**Read Latency**	**Write Latency**	**Density**	**Cost**
DRAM	15 ns	15 ns	Low	$$$$
PCM	50 ns	500 ns	Medium	$$
ReRAM	10 ns	50 ns	High	$$$$
STT-MRAM	10 ns	50 ns	Low	$$$
CNT	$< 50ns$	$< 50\ ns$	High	$$$

TABLE I
COMPARISON OF DIFFERENT NVM TECHNOLOGIES [55], [62], [78].

B. What's the Hype?

A technology that seamlessly blurs the line between main memory and persistent storage offers tantalizing possibilities, from closing the IO gap [45] to unifying the representation of dynamic and persistent objects to eliminating or significantly reducing system recovery time. Fundamentally, it makes us rethink both our hardware and software stacks. Is NVM a larger main memory (i.e., DRAM replacement), a faster persistent storage (i.e., a replacement for SSDs and HDDs), or a new level in the memory hierarchy? Does NVM obviate the need for file systems? What about the storage layer of database systems? Does caching go away? If not, are caches now persistent?

In addition to introducing exciting opportunities, NVM also creates new problems. How do we handle error conditions traditionally "fixed" by a reboot? How do we keep data secure – if there is no distinction between main and persistent memory, when do we encrypt data? What steps do we need to take to securely delete data? How do we make persistent memory reliable in the face of device failure?

With the steady stream of product announcements, particularly, the Intel/Micron 3D XPoint, there has been enormous research activity addressing these questions and more. We take an historical perspective and focus on the fundamental technologies and techniques that researchers are bringing to bear on these problems. In particular, we compare the research questions that have been asked in the past with those being asked today to determine what we've learned in the past sixty years and where there are unique and unmined areas of exploration.

The rest of this paper is structured as follows. In Section II, we discuss the core technologies that are fundamental to our ability to fully leverage NVM. We then review the different areas of current NVM research in Section III. In Section IV, we speculate about the most promising avenues of future research and then conclude in Section V.

II. NVM PAST

Building systems that exploit NVM introduces a host of new challenges, but fortunately, many of the problems that need to be solved can build upon prior work. In this section, we introduce some of those relevant technologies, highlight the research undertaken in these areas in the past, and identify the results that apply and those that remain unaddressed in modern NVM systems. We'll begin by revisiting bubble memory [7], perhaps the first instance of NVM. Next, we will move to a discussion of the single level store [46], which initially integrated main memory with backing store to produce virtual memory and later, in Multics, extended to seamless integration between address spaces and persistent storage [10]. This model of integrating storage into process address spaces is fundamental to how NVM use is envisioned. After that, we'll visit the interconnected areas of persistent object systems and object oriented databases, both of which bear close resemblance to research being undertaken today in persistent data structures. Finally, we'll touch upon transactional memory, which provides the mechanisms most frequently used to ensure that updates to persistent objects accessible in main memory do not produce persistent inconsistencies.

A. Bubble Memory

We all know the saying (or its derivatives), ”Those who cannot remember the past are condemned to repeat it [72]." but we may not all be as aware of how eerily familiar the excitement over NVM is. Let's play a little game. Below are eight quotes – pick the decade from which each quote came (no fair checking the references before playing). We've replaced each instance of NVM/bubble memory with "new

[1] A memristor was a hypothetical circuit, first proposed by Leon Chua in 1971. There is some argument about whether today's NVM technologies, such as ReRAM technically qualify as memristors.

technology" and instances of mechanical disk, drum storage, hard drive, flash, SSD with "persistent storage."

- New technology has the potential of replacing persistent storage [8].
- The new technology market is finally ready to start to fulfill some of the expectations its enthusiasts have been predicting for a number of years [8].
- There are several aspects of new technology that make existing database architectures inappropriate for them [1].
- However, favorable factors are now emerging that will propel the emerging new technology business onto a rapid growth trajectory [35].
- By exploiting this new technology for relational database management systems, we introduce efficient support for permutation, sorting and searching for data [91].
- We present four alternative implementations to incorporate new technology into the processing stack of a query processor [81].
- Some of the researchers in this field look forward to the ultimate replacement of persistent storage by new technology [64].
- Since the new technology is intrinsically similar to the data model, and adapted to the access requirements, we believe the overall system is simpler both in operation and in programming. [14].

Let's assume you now agree that there are commonalities between then and now. So, what is bubble memory? Bubble memory arose from fundamental research on semiconductor materials in the 1960s [67]. The bubbles themselves are cylindrical crystals of an appropriate semiconductor material. They are oriented vertically in a thin magnetized film, such that a charge applied in the perpendicular direction causes the crystals to bubble. In this way, the presence of the bubble indicates a 1 and the absence a 0 [7].

Early reaction to bubble memory devices sounds a lot like what we hear today about NVM, "Some of the researchers in this field look forward to the ultimate replacement of disks by the magnetic bubbles of the future" [64]. Several companies, including National Semiconductor, Rockwell, and Texas Instruments entered the market [8]. However, as the manufacturing process became more mature, the message became more nuanced, "Bubble systems are not expected to replace disk storage in many commercial applications because they will be too expensive, but for areas where the environment is severe, or 'dusty', or remote, bubbles will prove more reliable and cheaper to maintain" [7].

Bubble memory saw some successful deployment in embedded systems; perhaps the most successful of which was in the Konami Bubble System arcade video game system (1984) [13]. However, by 1981, all the major vendors had left the market. Why? Ultimately, both RAM and hard drives became cheaper and more dense (a trend that we've seen continue for the past three decades), while bubble fabrication was challenging and required a complex controller (similar to that of a hard disk) [30].

Much of the activity around bubble memory was in companies, more so than the research community, so the kind of system research activity today around NVM is relatively scarce in the bubble memory era. However, some of the problems tackled will seem familiar.

Some considered development of file system access structures that would minimize access time, for example, designing record sizes to match the physical device characteristics, eschewing marshalling multiple items into large blocks, and allowing for tree structures persisted natively [86]. We might ask how this differs in any fundamental way from more recent research in designing data structures designed to exploit cache locality [52]. Much of the work around using bubble memory to improve relational database performance focused on special purpose hardware approaches [14], [28], [92], rather than the software approaches popular today.

So, what lessons can we learn from this brief historical jaunt? One lesson may be caution in the presences of euphoria. Recall that the major vendors of the time (e.g., National Semiconductor, Texas Instruments) invested heavily in new technology that ultimately disappeared. Price/performance curves matter a great deal, so while NVM does provide enormous opportunities for fundamental changes in our systems, the technology will survive only if it can deliver on those opportunities in the face of economic reality. If NVM can genuinely approach DRAM performance at Flash prices, it is likely to be a game-changer. Anything short of that will leave its future in jeopardy.

B. Single Level Store

The idea of seamlessly integrating volatile main memory with persistent storage began with the advent of virtual memory. The University of Manchester's Atlas computer [32], [46] was the first system to implement what we take for granted today: that an address could refer to data stored in main memory or to one in persistent memory (in the case of Atlas, on a magnetic drum). While we most frequently associate this concept with virtual memory, researchers of the day recognized its broad applicability, "...the paper describes an automatic system which in principle can be applied to any combination of two storage systems so that the combination can be regarded by the machine user as a single level" [46]. In fact, this is precisely what the Multics virtual memory system did: "the Multics user no longer uses files; instead he references all information as segments, which are directly accessible to his programs" [10].

Research in this area in the 60s and 70s focused on three primary topics: designing the hardware to support virtual memory [23], [46], [54], the software structure and algorithms for managing virtual memory [26], [27], [34], [46], and the introduction of the hierarchical name space as a means for organizing persistent data [10].

It is instructive to compare the historical single level store concept and implementation to what we observe in today's research. Persistence combines two fundamental concepts: naming and storage. The historical work experimented with both, since there were not prevailing standards at the time. Today, the notion of a file system namespace and protection is fundamentally tied to persistence, leading to a two-part solution: use of traditional file system naming and protection mechanisms coupled with direct access via the process address space [71]. Rather than suggest lessons learned, this restrospective suggests that one ask whether file system naming and protection mechanisms are the correct approach to management of persistent memory. We'll come back to this question in Section IV.

C. Persistent Objects

Once we begin to blur the line between main memory objects and persistent objects, we find ourselves in the domain of persistent object systems and then object oriented databases. The early history of persistent objects and object oriented databases will seem quaint with the wisdom of hindsight. These systems arose, in many cases, in response to limitations in the relational model, as originally conceived by Codd [20] and realized in systems such as Ingres [36] and SystemR [3]. Relational databases were limited in the types they provided, their ability to construct complex types (i.e., objects), limitations on data type representations (e.g., maximum field widths), their modeling power, and the fundamental mismatch between data as it is used in programs and as it is represented in a database [5], [22]. While we know how this played out (e.g., the object relational model [76], [77]), there is a rich history of both persistence in object oriented languages and object oriented databases. In a world where persistent objects reside in a process's address space, we are well-advised to refresh our memory with the techniques and approaches developed in the past.

We limit our discussion to systems that allow direct manipulation of persistent objects, such as persistent Smalltalk (OPAL) [22], PS-Algol [5], and ObjectStore's persistent C++ (pC++) [49]. All of these systems provide database-style transactions and query mechanisms.

PS-Algol is perhaps most similar to today's work in NVM, stemming from the hypothesis that, "it should be possible to add persistence to an existing language with minimal change to the language" [5]. In fact, their design was agnostic to the host language to which persistence was added. Today's NVM researchers would find PS-Algol eerily familiar. A programmer opens a database and obtains a root pointer from which all persistent objects can be retrieved. In today's work, such as NV-Heaps [19] and the SNIA NVM programming model PM modes [75], these operations are parallel to `mmap`ping a file and then accessing persistent objects relative to the `mmap`ped region.

In contrast, OPAL and pC++ call themselves database systems and are designed around specific data models. OPAL implements the Gemstone Data Model, derived from a pure set theoretic data model, which was formally specified before the host language was selected. The data model for pC++ is somewhat less formally defined, but is something akin to an object-relational model, providing C++ objects, classes, and collections in addition to RDBMS relationships and queries. Eventually, different groups combined modeling efforts, forming The Object Data Management Group (ODMG) (http://www.odbms.org/odmg-standard).

No discussion of object oriented languages would be complete without introducing Java. Early work around persistence in Java took the form of *orthogonal persistence*, also called persistence by reachability [4], [6]. The fundamental idea was quite attractive: Programmers can simply designate objects of any type as "persistent roots," and objects of any type transitively reachable from such roots inherit the property of persistence. Moreover, the code that accesses such objects is exactly the code that accesses nonpersistent objects of the same types. While appealing, the idea did not catch on in mainstream Java, largely because it was perceived as a far too ambitious undertaking. Instead, the Java Data Objects (JDO) [42] and Java Persistence API (JPA) [44], which were similar in spirit to the ODMG standards, gained traction. However, they both relied on *dual* instantiation of persistent data (one in storage media and one in DRAM).

These systems all faced the implementation challenge of moving objects between volatile and persistent (block-based) storage. Today's NVM systems need not migrate data between different media, but, depending on how the NVM is accessed (block mode versus byte addressable mode), may still need to address the issues of allocating objects to blocks. However, the more interesting issue is the emphasis in the older systems on database functionality, such as transactions, versioning, replication, and naming. In modern systems, little NVM research addresses versioning, replication or naming, and in many cases considers transactions only as a mechanism to ensure consistency of NVM data structures, not to provide higher level data semantics. The next section explores this transaction issue in more detail; we'll return to these database issues in Section IV.

D. Transactional Memory

The key difference between persistent object systems of the past and today's NVM lies in the hardware reality that any write can become persistent at any time. Maintaining data structure consistency in the absence of control over when data becomes persistent requires either careful and meticulous application of barrier and flush instructions or a higher level transaction or transaction-like mechanism, similar to software transactional memory, which we now review.

The idea of transactional memory started out as a hardware solution for achieving synchronization without explicit locking. Hardward transactional memory (HTM) is essentially a form of optimistic concurrency control [48] where the validate, commit, and abort operations are handled via extensions to the cache architecture [37]. Hardware transactional memory has some inherent limitations, such as the length of transactions and precisely what operations can occur inside a transaction. Unsurprisingly, researchers took up the challenge these constraints presented with software (STM) [74] and hybrid [24] implementations.

While many of the transaction mechanisms designed for NVM reference the prior work on STM, there is rarely a technical discussion about why prior approaches do not apply. We speculate that there is a strong tendency towards NIH (not-invented-here) and suggest that current researchers might save time and effort by leveraging the prior work in STM. In our own work, we find that the trade-off space in transaction implementations is complicated. The different approaches (e.g., undo-logging, redo-logging, copy-on-write) require significantly different numbers of barrier instructions and have dramatically different effects on cache performance. The best performing implementation depends on the size and number of objects being accessed, the read/write ratio, and the degree of contention on the objects. As practically all of these techniques have been used in the STM literature, it would seem prudent to take advantage of that work.

III. NVM Present

Given the long history of work in related fields, what's new about today's NVM? First, as discussed in Section I, activity around NVM is not concentrated on a single technology. There are multiple viable technologies poised to become commercially relevant. Second, there is significant industry collaboration around the emerging technologies. The Storage Networking Industry Association (SNIA) has a working group developing reference libraries and programming models. Both Linux and Windows support the Direct Access (DAX) feature, which allows loads and stores to persistent memory after the memory has been mmapped into an address space. This could either be a side effect of the maturity of the industry, relative to 1980, or it could be a sign that things are truly different. Third, The types of companies active in the space are different from those active in the 1980s. While the Micron/Intel 3D XPoint product might be similar in spirit to the National Semiconductor's and Texas Instruments' products of the 1980s, we also see engagement today from systems' companies: HP, Microsoft, IBM, Oracle, etc. Perhaps the most audacious project in this space was HP's The Machine [80], which was canceled as a project or product, but is moving the key technologies, such as NVM, into other product lines [57].

The biggest observable difference, however, is the engagement of the research community. It's difficult to assess whether the engagement is greater because the research community itself is significantly larger than it was in the 1980s, because new technology is always good fodder for research, or because something is fundamentally different from prior decades. Nonetheless, although NVM systems are not yet shipping, there are hundreds of prototype systems poised to exploit it.

Today's research falls into approximately five categories: NVM-aware storage allocators [19], [65], NVM-consistent data structures [15], [16], [41], [50], [87], file systems for NVM [82], [88], [89], NVM-based logging systems [40], [84], and database engine architectures [2], [47]. In addition to this research activity, standards, e.g., SNIA's NVM Programming Model [75] and ISA changes to support NVM are active areas.

Rather than visiting each area, we focus on the themes that cut across multiple areas. Then, in Section IV, we connect some of the remaining open questions and challenges to the topics discussed in Section II.

A. Synchronization, Consistency and Programming Models

Historically, in-memory data structures use a combination of synchronization primitives and cache consistency protocols to achieve consistency, while persistent data leverages the rigid interface to IO (e.g., the file system or database) and ordered large grain operations to ensure consistency and/or recoverability. In an NVM system, neither approach is sufficient. As discussed in Section II-D, there is no way to prevent data stored into NVM from becoming persistent. This leads to a programming model with fine grain ordering, requiring careful placement of barriers and flushes. That is, rather than using higher level synchronization primitives, such as locks or semaphores, the programmer works at the level of assembly instructions, a style similar to that used when working with lockless data structures [9].

Intel originally extended its ISA with a PCOMMIT instruction to ensure that data in the memory controller write pending queues was flushed to persistent storage. However, they later specified that Asynchronous DRAM Refresh (ADR) is required to support persistent memory. With ADR support, data in the controller write pending queues is guaranteed to make it to persistent memory, so PCOMMIT became unnecessary and was deprecated. A side effect of this change is reducing (from two to one) the number of SFENCE instructions required to reliably write data to persistent memory. However, even without PCOMMIT, ensuring write ordering to NVM requires one cache flush instruction per involved cache line plus an SFENCE instruction. While the flushes can be asynchronous (e.g., by using CLWB and CLFLUSHOPT), the SFENCE adds significant latency to operations. One theme that emerges across most of the work is designing approaches that minimize the number of SFENCEs.

Some of the early work (BPFS) in the NVM era proposed hardware extensions to enforce ordering [21]. The Byte-Addressable Persistent File System (BPFS) develops a tree-structured file system that primarily uses shadow paging for updates, but propagates shadows up the tree until arriving at a page that can be modified with a single, atomic write. At that point, BPFS performs the update in place with the atomic write, thus *short-circuiting* the copy-on-write mechanism of shadow-paging. Subsequent work eschews the invention of new hardware and works within the confines of the existng Intel ISA with NVM extensions. Several early systems explored the use of traditional logging approaches [66], [83], with their accompanying SFENCE instructions. The fundamental difference between these logging solutions and those used in the STM arena are that the STM logs are not persistent, so cache consistency algorithms neatly provide ordering constraints, while the NVM solutions require stricter ordering requirements. As a result, it is unsurprising, the subsequent research examined approaches that remove such ordering constraints.

Storing tree-nodes as unordered entries transforms updates in carefully designed systems into atomic operations, removing the need for the expensive barrier instructions [90]. The NV-Tree takes this one step further, observing that only leaf nodes need to be persistent; internal nodes can be reconstructed after failure [16].

Another approach to mitigating the cost of barrier instructions is removing the barriers from the critical path, typically through a combination of volatile caching on top of an NVM persistent structure [43], [51], [58], [87]. This class of solutions is particularly interesting, as it subtly changes the architectural integration of NVM. Earlier solutions treat NVM as both main memory and a persistent store, but these decoupled approaches retain the more traditional model of DRAM as a cache for persistence, thereby treating NVM like a traditional flash or hard drive, just with a different interface.

A common theme observed throughout all these systems is careful attention to the number and placement of flush and fence instruction. Designs using this approach are brittle and difficult to get correct – a single missing flush or fence, can render persistent data structures unrecoverable. This makes the issue of programming model crucial. One of the earliest complete implementations of a programming model is the NVM Direct library [12], which proposes extensions to C along with a C pre-compiler. The goal of the library is catch subtle problems at compile time that, if left undetected, could produce persistent corruption. The library is a comprehensive collection of concepts and techniques from both the past and present in addition to some new constructs. It has support for STM-like transactions, but with support for open nesting [63], persistent heaps [19], `mmap`ped regions, USIDS or persistent identifiers that map to executable functions (similar to application-specific logging and recovery functions [11], [73]), deferred operations that take place during commit, abort, or unlock, and explicit syntax to differentiate access to persistent and volatile data.

The newer NVM Programming Model from SNIA (NPM) takes a more hands-off approach [75]. Rather than specify an API, it defines different modes of access (ifile, block, volume, pm-file), the behavior of modules comprising a system, and the interactions among those modules. The standard requires that when NVM is used as volatile memory, it must behave identically to volatile memory with respect to load/store instructions, atomic read-modify-write instructions, cache coherency, etc. When used as non-volatile memory, it must respect `msync` calls, preserve the modification order of stores to the same location, and preserve the atomicity of thread-ordered stores (e.g., C11 and C++11 atomic stores and Java and C# volatile stores). However, subsequent stores to different locations, still require flush and fence operations.

The sufficiency of these different programming models depends on precisely who will be writing code that interacts with NVM. Will NVM be hidden behind standard data structure APIs or will applications manipulate NVM directly? Based on today's developments, we believe that direct manipulation of NVM will be hidden inside higher level libraries: data structure packages, collections, databases, and filesystems. If so, the more laissez faire SNIA approach is likely sufficient.

B. Naming

The saying, "There are only two hard things in Computer Science: cache invalidation and naming things," [2] is attributed to Phil Karlton. We have nothing to say about cache invalidation (although it is probably an important problem if caches reside in NVM), but naming remains central to the management of persistent data. Ever since Multics introduced the hierarchical path name to access persistent data [10], such pathnames have been the default answer. Early database systems explicitly managed persistent storage (i.e., disks), implementing their own naming system. Although this capability is still available today, it is more common for database systems to use the naming services of the underlying file system, which allows the use of standard utilities, i.e., `cp` and `mv`, on databases. Even in the persistent language systems discussed in Section II-C, programs must open databases by pathname. Thus, it comes as no surprise that the NVM Programming Model also specifies use of file system naming to locate and identify NVM regions. Ultimately though, this could create a bit of a chicken and egg problem: if NVM is your persistent storage, does it not make sense to implement the file system namespace in NVM? If so, then we need to find the part of the NVM that contains the namespace, so we need a name for it. Alternately, we could support two different ways of using NVM – first as a file system to implement naming and then as a direct access store for better performance? Surely, there must be a better (simpler) way.

C. Replacing Persistent Storage Components

This brings us to the last cross-cutting theme: transforming systems designed for hard drives into systems leveraging NVM: logging, file systems, and databases. In all of these cases and those discussed in Section II-C, persistent storage is block-based, while today's NVM is byte-addressable. Byte-addressability introduces two important considerations: there is no need to marshall small data structures onto blocks and the persistent media is available via direct loads and stores, greatly reducing the depth of the software stack, which traditionally requires a mode switch and an IO protocol. There seems to be an absence of research that revisits these 1980 era persistent object systems in the presence of byte-addressability, which is unfortunate, as it seems a natural fit.

Different systems take more or less radical approaches to byte-addressability, ranging from using blocks in conjunction with finer-grained writes [21], [29] to revising existing designs [2], [53], [85] to building entirely new systems [88]. One topic that is regularly ignored is the fine-grained intermingling of persistent and volatile state that the NVM Direct library

[2] There is still skepticism about the source of the quote. See https://skeptics.stackexchange.com/questions/19836/has-phil-karlton-ever-said-there-are-only-two-hard-things-in-computer-science/39178#39178

carefully addresses [12]. Projects that do address this find it is a serious problem [53].

IV. NVM Future

If NVM is to become the de facto persistent storage of the future (and this is still a big if), there remain important questions to answer and problems to be addressed. There are two areas of work that we've left out: whole system persistence and embedded systems. If the history of bubble memory is an exemplar, perhaps the embedded systems arena is, in fact, the one likely to feel the greatest effect. In terms of whole system persistence, this is an area that warrants a full discussion on its own right; we'll leave that for another time.

The biggest open secret in this space is the need for redundancy or replication. Fundamentally, persistent storage sitting on the memory bus is rather different from a dual ported, shared disk or RAID device. Today, losing a CPU board rarely results in data loss. However, when persistent data is on the memory bus, what happens when you lose the board? Since we do not anticipate dual ported memory in the near future, ensuring data availability requires replicating the data. Compared to traditional persistent media (i.e., hard drives), networking overheads are tolerable. However, compared to NVM latency, the traditional TCP stack adds orders of magnitude to the raw latency, which makes traditional replication techniques non-starters. RDMA networks, and more recent industry trends toward high performance network fabrics that tend toward a distributed shared memory style interface to remote memory [33] would seem to be essential in the NVM setting. We observe some industry movement in this direction with support for remote persistence over RDMA networks [68]. This area seems crucial to address before we can trust critical data to NVM systems.

Turning to traditional systems issues, the disconnect between the relatively feature-rich persistent object models of the 1980s and the no-model systems we see today is interesting. Have we all drunk the NoSQL koolaid to the extent that we no longer believe data models are important? The continued presence of relational and object-relational systems suggests that this might be a serious oversight. There must be at least one good Ph.D. dissertation to be written on data models for NVM data management systems.

Versioning is another topic on which the current work is silent. That seems an odd omission in the presence of concerns over data tampering and the prevalence of systems supporting MVCC.

Data security raises another set of questions. Best practices for data management include encrypting data "at rest". Historically, there was a clear distinction between data at rest (i.e., on disk) and data in volatile memory. On a system where all memory is persistent, best practice would suggest that data in NVM should be encrypted. But if we manipulate such data directly where it resides, how, when and where do we decrypt it? Intel provides Total Memory Encryption (TME) and MultiKey TME (MKTME), both of which provide a mechanism to keep all memory encrypted except when it is physically being manipulated in the CPU. TME uses a single key, while MKTME uses a bounded set of keys, and data is encrypted on a page basis. However, MKTME assigns keys on a block basis, which probably reduces the value of byte addressability. Is this kind of hardware solution sufficient? If we use NVM for a database server, how do we multiplex the potentially large set of principles across a limited set of memory keys?

And then there is naming. Is the combination of file system naming and direct access correct? It seems awkward and leads to the chicken and egg problem discussed in Section III. We are entirely comfortable mounting removable media (e.g., flash and hard drives) with symbolic names. Why has none of the work taken this approach with NVM? Is it just too obvious and therefore uninteresting? If so, we might argue that sometimes obvious and uninteresting approaches can also be necessary.

Finally, reboot has been the ultimate debugging tool forever. What happens to our ability to fix systems when rebooting returns a system to precisely the state it was in before a reboot? How do we preserve our persistent data, without preserving our persistent bugs as well? If NVM becomes pervasive, we're going to have to figure that out.

V. Conclusions

NVM is an exciting and potentially game changing technology. However, its future is far from clear. On one hand, we encourage a certain degree of skepticism. The future of NVM will be determined by a combination of the new functionality it permits, the improvement to existing systems it enables, the cost of NVM, and the future cost of existing volatile memory and persistent storage. On the other hand, we should prepare to embrace this brave new world – it offers promises of vastly improved performance and ultimately simpler systems if we don't have to rely on different in-memory and persistent representations.

In Section IV, we suggested some gaps in today's research. In particular, ensuring high availability and fault tolerance of persistent data stored on NVM is essential. Similarly, history provides valuable lessons – features such as versioning, encryption and naming have received insufficient attention. We're convinced that the innovation in this space will come from, not yet another NVM-aware data structure, but from leveraging past work effectively to build exciting new systems with new capabilities. Let's revisit this discussion in a decade and see what we've accomplished.

References

[1] J. Arulraj and A. Pavlo. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1753–1758, New York, NY, USA, 2017. ACM.

[2] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

[3] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.

[4] M. Atkinson and M. Jordan. A review of the rationale and architectures of pjama: A durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical report, Mountain View, CA, USA, 2000.

[5] M. P. Atkinson, P. J. Bailey, K. Chisholm, W. P. Cockshott, and R. Morrison. An approach to persistent programming. *Comput. J.*, 26:360–365, 1983.

[6] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *SIGMOD Rec.*, 25(4):68–75, Dec. 1996.

[7] K. F. Baker. A review of magnetic bubble memories and their applications. *Radio and Electronic Engineer*, 51(3):105–115, March 1981.

[8] H. Banks. The computer bubble that burst, September 1981.

[9] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 261–270, New York, NY, USA, 1993. ACM.

[10] A. Bensoussan, C. T. Clingen, and R. C. Daley. The multics virtual memory: Concepts and design. *Commun. ACM*, 15(5):308–318, May 1972.

[11] Application specific logging and recovery. *Berkeley DB Programmer Reference Manual*, june 2016.

[12] B. Bridge. Nvm direct api (version 0.8), February 2016.

[13] Magnetic bubble memory.

[14] H. Chang. On bubble memories and relational data base. In *Proceedings of the Fourth International Conference on Very Large Data Bases - Volume 4*, VLDB '78, pages 207–229. VLDB Endowment, 1978.

[15] A. Chatzistergiou, M. Cintra, and S. D. Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proc. VLDB Endow.*, 8(5):497–508, Jan. 2015.

[16] S. Chen and Q. Jin. Persistent b+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, Feb. 2015.

[17] J. Choe. Intel 3d xpoint memory die removed from intel optane$^{\text{TM}}$ pcm (phase change memory, May 2017.

[18] L. Chua. Resistance switching memories are memristors. *Applied Physics A*, 102(4):765–783, Mar 2011.

[19] 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 *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118, 2011.

[20] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, Jan. 1983.

[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 *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[22] G. Copeland and D. Maier. Making smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 316–325, New York, NY, USA, 1984. ACM.

[23] R. C. Daley and J. B. Dennis. Virtual memory, processes, and sharing in multics. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 12.1–12.8, New York, NY, USA, 1967. ACM.

[24] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, Oct. 2006.

[25] M. F. L. De Volder, S. H. Tawfick, R. H. Baughman, and A. J. Hart. Carbon nanotubes: Present and future commercial applications. *Science*, 339(6119):535–539, 2013.

[26] P. J. Denning. The working set model for program behavior. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 15.1–15.12, New York, NY, USA, 1967. ACM.

[27] P. J. Denning. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 915–922, New York, NY, USA, 1968. ACM.

[28] K. L. Doty, J. D. Greenblatt, and S. Y. W. Su. Magnetic bubble memory architectures for supporting associative searching of relational databases. *IEEE Transactions on Computers*, C-29(11):957–970, Nov 1980.

[29] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[30] J. Dvorak. Whatever happened to bubble memory?

[31] J. Evangelho. Intel and micron jointly unveil disruptive, game-changing 3d xpoint memory, 100x faster than nand, July 2015.

[32] J. Fotheringham. Dynamic storage allocation in the atlas computer, including an automatic use of a backing store. *Commun. ACM*, 4(10):435–436, Oct. 1961.

[33] Genz consortium core specification 1.0, February 2018.

[34] R. M. Graham. Protection in an information processing utility. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 1.1–1.5, New York, NY, USA, 1967. ACM.

[35] J. Happich. Time is ripe for emerging non-volatile memory, say analysts yole, June 2017.

[36] G. D. Held, M. R. Stonebraker, and E. Wong. Ingres: A relational data base system. In *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, AFIPS '75, pages 409–416, New York, NY, USA, 1975. ACM.

[37] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[38] M. Hosomi, H. Yamagishi, T. Yamamoto, K. Bessho, Y. Higo, K. Yamane, H. Yamada, M. Shoji, H. Hachino, C. Fukumoto, H. Nagao, and H. Kano. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. *International Electron Devices Meeting*, pages 459–462, 2005.

[39] Y. Huai. Spin-Transfer Torque MRAM (STT-MRAM): Challenges and Prospects. *AAPPS Bulletin*, 18(6):33–40, 2008.

[40] J. Huang, K. Schwan, and M. K. Qureshi. Nvram-aware logging in transaction systems. *Proc. VLDB Endow.*, 8(4):389–400, Dec. 2014.

[41] D. Hwang, W.-H. Kim, Y. Won, and B. Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 187–200, Oakland, CA, 2018. USENIX Association.

[42] Jsr 12: Javatm data objects (jdo) specification, 2003.

[43] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, Feb 2017.

[44] Java persistence api.

[45] R. H. Katz, G. A. Gibson, and D. A. Patterson. Disk system architectures for high performance computing. *Proceedings of the IEEE*, 77(12):1842–1858, Dec 1989.

[46] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, April 1962.

[47] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 691–706, New York, NY, USA, 2015. ACM.

[48] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[49] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The objectstore database system. *Commun. ACM*, 34(10):50–63, Oct. 1991.

[50] S. K. Lee, K. H. Lim, H. Song, B. Nam, and S. H. Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 257–270, Santa Clara, CA, 2017. USENIX Association.

[51] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. ACM.

[52] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 183–196, New York, NY, USA, 2012. ACM.

[53] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[54] A. J. W. Mayer. The architecture of the burroughs b5000: 20 years later and still ahead of the times? *SIGARCH Comput. Archit. News*, 10(4):3–10, June 1982.

[55] C. Mellor. Deep inside nantero's non-volatile carbon nanotube ram tech, August 2016.

[56] C. Mellor. Intel and micron's xpoint: Is it pcm? we think it is, January 2016.

[57] C. Mellor. Rip hpe's the machine product, 2014-2016: We hardly knew ye. *The Register*, november 2016.

[58] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[59] R. Mertens. Crocus nano electronics successfully tests its 90 nm pmtj stt-mram tech. *www.mram-info.com*, January 2018.

[60] R. Mertens. evaderis demonstrates an innovative mram-based, memory-centric mcu. *www.mram-info.com*, January 2018.

[61] R. Mertens. Everspin strats to produce commercial 40nm 256mb stt-mram chips. *www.mram-info.com*, January 2018.

[62] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distrib. Syst.*, 27(5):1537–1550, May 2016.

[63] J. E. B. Moss. Open nested transactions: Semantics and support. *Workshop on Memory Performance Issues (WMPI '06)*, February 2006.

[64] W. Myers. Key developments in computer technology: A survey. *Computer*, 9(11):48–77, Nov. 1976.

[65] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow.*, 10(11):1166–1177, Aug. 2017.

[66] 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 *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 371–386, New York, NY, USA, 2016. ACM.

[67] S. R. Ovshinsky. Reversible electrical switching phenomena in disordered structures. *Phys. Rev. Lett.*, 21:1450–1453, Nov 1968.

[68] Persistent memory over fabrics.

[69] A. Pohm, C. Sie, R. Uttecht, V. Kao, and O. Agrawal. Chalcogenide glass bistable resistivity (ovonic) memories. *IEEE Transactions on Magnetics*, 6(3):592–592, Sep 1970.

[70] A. Pohm, C. Sie, R. Uttecht, V. Kao, and O. Agrawal. Chalcogenide glass bistable resistivity (ovonic) memories. *IEEE Transactions on Magnetics*, 6(3):592–592, Sep 1970.

[71] A. Rudoff. Persistent memory programming. *;login*, 42, 2017.

[72] G. Santayana. *The Life of Reason: Reason in Common Sense*. Charles Scribner's Sons, 1905.

[73] M. Seltzer and K. Bostic. Berkeley db. *The Architecture of Open Source Applications*, 1, march 2012.

[74] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[75] Nvm programming model (npm), june 2017.

[76] Iso/iec 9075:1999, 1999.

[77] M. Stonebraker and L. A. Rowe. The design of postgres. *SIGMOD Rec.*, 15(2):340–355, June 1986.

[78] M. Swift and H. Volos. Programming and usage models for non-volatile memory, March 2015.

[79] B. Tallis. Techinsights publishes preliminary analysis of 3d xpoint memory, May 2017.

[80] A. Vance. With 'the machine,' hp may have invented a new kind of computer. *Bloomberg LP*, june 2014.

[81] S. D. Viglas. Write-limited sorts and joins for persistent memory. *Proc. VLDB Endow.*, 7(5):413–424, Jan. 2014.

[82] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[83] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[84] T. Wang and R. Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[85] M. Wilcox. Add support for nv-dimms to ext4.

[86] W. E. Wright. Some file structure considerations pertaining to magnetic bubble memory. *The Computer Journal*, 26(1):43–51, 1983.

[87] F. Xia, D. Jiang, J. Xiong, and N. Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, 2017. USENIX Association.

[88] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[89] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 478–496, New York, NY, USA, 2017. ACM.

[90] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, Santa Clara, CA, 2015. USENIX Association.

[91] M. Zaki. Magnetic bubble memory structures for relational database management systems. *International Journal of Computer & Information Sciences*, 10:341–358, 1981.

[92] M. Zaki. Magnetic bubble memory structures for relational database management systems. *International Journal of Computer and Information Sciences*, 10:341–358, october 1981.