Implications of Non-Volatile Memory as Primary Storage for Database Management Systems

Naveed Ul Mustafa*, Adrià Armejach†, Ozcan Ozturk*, Adrián Cristal† and Osman S. Unsal†
*Computer Engineering Department, Bilkent University, Ankara, Turkey.
Email: naveed.mustafa@bilkent.edu.tr, ozturk@cs.bilkent.edu.tr
† Barcelona Supercomputing Center, Barcelona, Spain.
Email: adria.armejach@bsc.es, adrian.cristal@bsc.es, osman.unsal@bsc.es

***Abstract*—Traditional Database Management System (DBMS) software relies on hard disks for storing relational data. Hard disks are cheap, persistent, and offer huge storage capacities. However, data retrieval latency for hard disks is extremely high. To hide this latency, DRAM is used as an intermediate storage. DRAM is significantly faster than disk, but deployed in smaller capacities due to cost and power constraints, and without the necessary persistency feature that disks have. Non-Volatile Memory (NVM) is an emerging storage class technology which promises the best of both worlds. It can offer large storage capacities, due to better scaling and cost metrics than DRAM, and is non-volatile (persistent) like hard disks. At the same time, its data retrieval time is much lower than that of hard disks and it is also byte-addressable like DRAM.**

In this paper, we explore the implications of employing NVM as primary storage for DBMS. In other words, we investigate the modifications necessary to be applied on a traditional relational DBMS to take advantage of NVM features. As a case study, we have modified the storage engine (SE) of PostgreSQL enabling efficient use of NVM hardware. We detail the necessary changes and challenges such modifications entail and evaluate them using a comprehensive emulation platform. Results indicate that our modified SE reduces query execution time by up to 40% and 14.4% when compared to disk and NVM storage, with average reductions of 20.5% and 4.5%, respectively.

I. Introduction

Traditional design of Database Management Systems (DBMS) assumes a memory hierarchy where datasets are stored in disks. Disks are a cheap and non-volatile storage medium suitable for storing large datasets. However, they are extremely slow for data retrieval. To hide their high data-access latency, DRAM is used as an intermediate storage between disks and the processing units. DRAM is orders of magnitude faster than a disk. In addition, with increasing DRAM chip densities and decreasing memory prices, systems with large pools of main memory are common.

For these reasons, relational in-memory DBMSs have become increasingly popular [1, 2, 3, 4]. Significant components of in-memory DBMSs, like index structures [5], recovery mechanisms from system failure [6], and commit processing [7] are tailored towards the usage of main memory as primary storage. However, in-memory DBMSs dealing with critical or non redundant data still need to provide a form of persistent storage, typically a large pool of disks [1, 8, 9, 10].

DRAM is a major factor affecting the power-efficiency of in-memory database servers. In a typical query execution for an in-memory database, 59% of the overall energy is spent in main memory [11]. Furthermore, there are inherent physical limitations related to leakage current and voltage scaling that prevent DRAM from further scaling [12, 13]. As a result, DRAM is unlikely to keep up with current and future dataset growth trends as a primary storage medium.

NVM is an emerging storage class technology which provides a good combination of features from disk and DRAM. Prominent NVM technologies are PC-RAM [1] [14], STT-RAM [2] [13], and R-RAM [3] [15]. Since NVM provides persistency at the device level, it does not need a refresh cycle like DRAM to maintain data states, as a consequence NVM technologies consume less energy per bit compared to DRAM [16]. In addition, NVM features significantly better access latencies than hard disks - with read latencies being almost as good as those of DRAM, byte-addressability, and higher density than DRAM [17].

To benefit from these features, a DBMS design should take into account the characteristics of NVM. Simple ports of a traditional DBMS - designed to use disks as primary storage medium - to NVM will show improvement due to the lower access latencies of NVM. However, adapting a DBMS to fit NVM characteristics can offer a number of benefits beyond lower access latencies.

In this paper, we study the implications of employing NVM in the design of a DBMS. We first discuss and provide insights on the different available options of including NVM into the memory hierarchy of current systems. We then investigate the required modifications in the DBMS's storage engine (SE) to leverage NVM features using a well-known relational disk-optimized DBMS - PostgreSQL. We explain in detail the necessary steps to modify PostgreSQL, and explain how the modifications impact the internals of the DBMS. Our modifications aim at providing fast access to data by bypassing the slow disk interfaces while maintaining all the functionalities of a robust DBMS such as PostgreSQL.

We evaluate two modified SEs of PostgreSQL using a comprehensive emulation platform and the TPC-H [18] benchmark. In addition, we also evaluate an unmodified version of

[1]PC-RAM: Phase Change Random Access Memory
[2]STT-RAM: Spin Transfer Torque Random Access Memory
[3]R-RAM: Resistive Random Access Memory

PostgreSQL using a high-end solid state disk and the emulated NVM hardware. We show that our modified SEs are able to reduce the kernel execution time, where file I/O operations take place, from around 10% to 2.6% on average. In terms of wall-clock query execution time, our modifications improve performance by 20.5% and 4.5% on average when compared to disk and NVM storage, respectively. We also demonstrate that the performance of our modified SE is limited by the fact that, since data is directly accessed from the NVM hardware, it is not close to the processing units when it is needed for query processing. This leads to long latency user-level cache misses that eat up the improvements achieved by avoiding expensive data movement operations.

II. Background

In this section, we first describe in detail the properties of NVM technologies, highlighting the implications these might have in the design of a DBMS. We then describe currently available system software to manage NVM.

A. Characteristics of NVM

Data access latency: Read latency of NVM technologies is significantly lower than that of a disk. However, since NVM devices are still under development, sources quote varying read latencies. For example, the read latency for STT-RAM ranges from 1 to 20ns [16, 19, 20]. Nonetheless, there is a general consensus that read latencies will be similar to those of DRAM [16, 19].

PC-RAM and R-RAM are reported to have a higher write latency compared to DRAM, but STT-RAM also outperforms DRAM in this regard [16, 19]. However, the write latency is typically not on the critical path, since it can be tolerated by using buffers [17].

Density: NVM technologies provide higher densities than DRAM, which makes them a good candidate to be used as main memory as well as primary storage, particularly in embedded systems [21]. For example, PC-RAM provides 2 to 4 times higher density as compared to DRAM [17], and it is expected to scale to lower technology nodes as opposed to DRAM.

Endurance: Endurance is defined as the maximum number of writes for each memory cell [17]. The most promising contestants are PC-RAM and STT-RAM. Both memories offer an endurance close to that of DRAM. More specifically, endurance for NVMs is 10^{15} whereas for DRAM it is 10^{16} [22]. On the other hand, NVMs exhibit higher endurance than flash memory technologies [19].

Energy consumption: Since NVMs do not need a refresh cycle to maintain data states in memory cells like DRAM, they are more energy efficient. A main memory designed by using PC-RAM technology consumes significantly lower per access write energy as compared to DRAM [22]. Other NVM technologies also have similar lower energy consumption per bit when compared to DRAM [16, 20].

In addition to the features listed above, NVM technologies also provide byte-addressability like DRAM and persistency like disks. Due to these features, NVMs are starting to appear in embedded and energy-critical devices and are expected to play a major role in future computing systems. Companies like Intel and Micron have launched the 3D XPoint memory technology, which features non-volatility [23]. Intel has also introduced new instructions to support the usage of persistent memory at the instruction level [24].

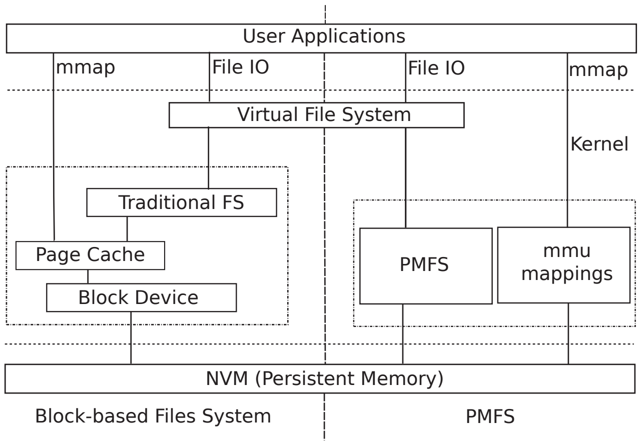


Fig. 1. Comparison of traditional FS and PMFS. "mmap" refers to the system call for memory mapped I/O operation. "mmu" is the memory management unit responsible for address mappings.

B. System software for NVM

Using NVM as primary storage necessitates modifications not only in application software but also in system software in order to take advantage of NVM features. A traditional file system (FS) accesses the storage through a block layer. If a disk is replaced by NVM without any modifications in the FS, the NVM storage will still be accessed at block level granularity. Hence, we will not be able to take advantage of the byte-addressability feature of NVM.

For this reason, there have been developments in file system support for persistent memory. PMFS is an open-source POSIX compliant FS developed by Intel Research [25, 26]. It offers two key features in order to facilitate usage of NVM.

First, PMFS does not maintain a separate address space for NVM. In other words, both main memory and NVM use the same address space. This implies that there is no need to copy data from NVM to DRAM to make it accessible to an application. A process can directly access file-system protected data stored in NVM at byte level granularity.

Second, in a traditional FS stored blocks can be accessed in two ways: (i) file I/O and (ii) memory mapped I/O. PMFS implements file I/O in a similar way to a traditional FS. However, the implementation of memory mapped I/O differs. In a traditional FS, memory mapped I/O would first copy pages to DRAM [25] from where application can examine those pages. PMFS avoids this copy overhead by mapping NVM pages directly into the address space of a process. Figure 1 from [25] compares a traditional FS with PMFS.

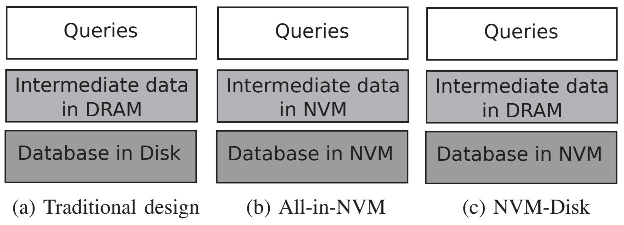


Fig. 2. NVM placement in the memory hierarchy of a computing system.

III. Design Choices

In this section, we discuss the possible memory hierarchy designs when including NVM in a system. We also discuss the high-level modifications necessary in a traditional disk-optimized DBMS in order to take full advantage of NVM hardware.

A. *Memory Hierarchy Designs for an NVM-Based DBMS*

There are various ways to place NVM in the memory hierarchy of a current DBMS computing system. Figure 2 shows different options that might be considered when including NVM into the system. Figure 2a depicts a traditional approach, where the intermediate state - including logs, data buffers, and partial query state - is stored in DRAM to hide disk latencies for data that is currently in use; while the bulk of the relational data is stored in disk.

Given the favorable characteristics of NVM over the other technologies, an option might be to replace both DRAM and disk storage using NVM (Figure 2b). However, such a drastic change would require a complete redesign of current operating system and application software. In addition, NVM technology is still not mature enough in terms of endurance to be used as a DRAM replacement. Hence, we advocate for a platform that still has a layer of DRAM memory, where the disk is completely or partially replaced using NVM, as shown in Figure 2c (NVM-Disk).

Using this approach, we can retain the programmability of current systems by still having a layer of DRAM, thereby exploiting DRAM's fast read and write access latencies for temporary data structures and application code. In addition, it allows the possibility to directly access the bulk of the database relational data by using a file system such as PMFS, taking full advantage of NVM technology, which allows the system to leverage NVM's byte-addressability and to avoid API overheads [27] present in current FSs. Such a setup does not need large pools of DRAM since temporary data is orders of magnitude smaller than the actual relational data stored in NVM. We believe this is a realistic scenario for future systems integrating NVM, with room for small variations such as NVM alongside DRAM to store persistent temporary data structures, or having traditional disks to store cold data.

B. *List of Modifications for a traditional DBMS*

Using a traditional disk-based database with NVM storage will not take full advantage of NVM's features. Some important components of the DBMS need to be modified or removed when using NVM as a primary storage.

Avoid the block level access: Traditional design of DBMS uses disk as a primary storage. Since disks favor sequential accesses, database systems hide disk latencies by issuing fewer but larger disk accesses in the form of a data block [28].

Unfortunately, block level I/O costs extra data movement. For example, if a transaction updates a single byte of a tuple, it still needs to write the whole block of data to the disk. On the other hand, block level access provides good data locality.

Since NVM is byte-addressable, we can read and write only the required byte(s). However, reducing the data retrieval granularity down to a byte level eliminates the advantage of data locality altogether. A good compromise is to reduce the block size in such a way that the overhead of the block I/O is reduced to an acceptable level, while at the same time the application benefits from some degree of data locality.

Remove internal buffer cache of DBMS: DBMSs usually maintain an internal buffer cache. Whenever a tuple is to be accessed, first its disk address has to be calculated. If the corresponding block of data is not found in the internal buffer cache, then it is read from disk and stored in the internal buffer cache [29].

This approach is unnecessary in an NVM-based database design. If the NVM address space is made visible to a process, then there is no need to copy data blocks. It is more efficient to refer to the tuple directly by its address. However, we need an NVM-aware FS, such as PMFS, to enable direct access to the NVM address space by a process.

Remove the redo logging: To ensure the atomicity, consistency, isolation and durability (ACID) properties of a database, a DBMS maintains two types of logs: the undo and redo logs. The undo log is used for cleaning after uncommitted transactions, in case of a system failure or a transaction abort issued by the program [30]. The redo log is used to re-apply those transactions which were committed but yet not materialized before the system failure.

In the case of NVM-based design, if internal buffers are not employed and all updates are materialized directly into the NVM address space then the need and criticality of the redo log can be relaxed [27]. However, the undo log will still be needed to recover from a system failure.

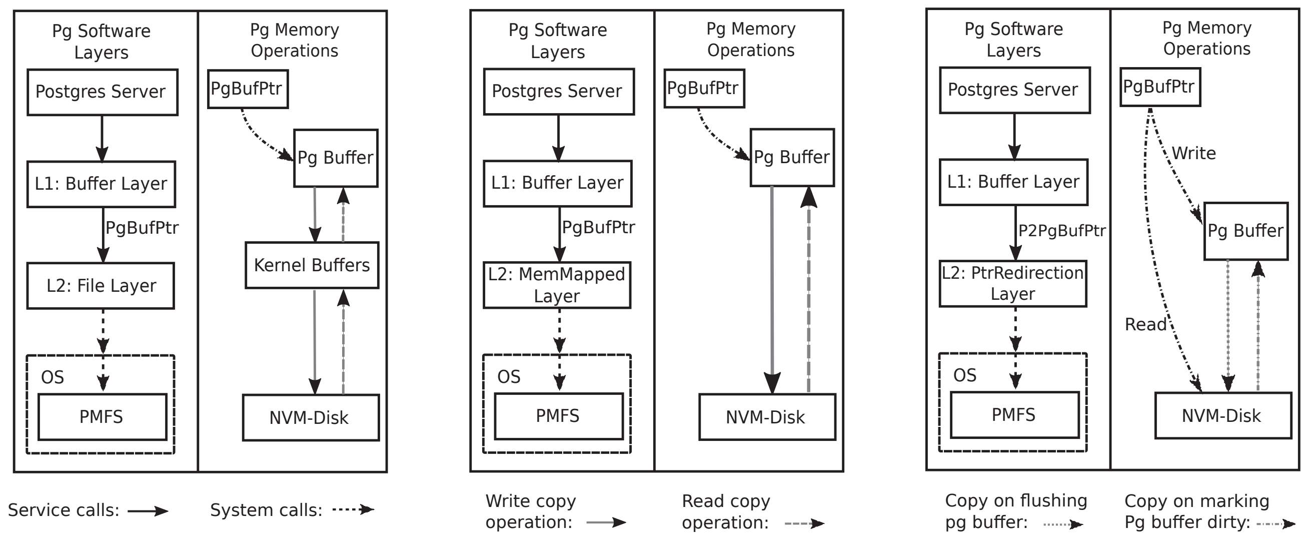
IV. A Case Study: PostgreSQL

PostgreSQL is an open source object-relational database system. It is fully ACID compliant and runs on all major operating systems including Linux [31].

In this section we study the storage engine (SE) of PostgreSQL and apply necessary changes to make it more NVM-aware. We first describe the read-write architecture of PostgreSQL and then explain our modifications.

A. *Read-Write Architecture of PostgreSQL*

Figure 3a shows the original PostgreSQL architecture from the perspective of read and write file operations. The left column in the figure shows the operations performed by



(a) Storage engine operation - PostgreSQL. (b) Modified storage engine - SE1 (c) Modified storage engine - SE2

Fig. 3. High level view of read and write memory operations in PostgreSQL (read as "pg" in short form) and modified SEs.

software layers of PostgreSQL, while the right column shows the corresponding data movement activities. Note that, we used PMFS for file operations. Furthermore, as shown in Figure 3a, we assume that the disk is replaced by NVM for storing the database.

PostgreSQL heavily relies on file I/O for read and write operations. Since the implementation of the file I/O APIs in PMFS is the same as that of a traditional FS, using a particular FS does not make any difference.

The PostgreSQL server calls the services of the *Buffer Layer* which is responsible for maintaining an internal buffer cache. The buffer cache is used to keep a copy of the requested page which is read from the storage. Copies are kept in the cache as long as they are needed. If there is no free slot available for a newly requested page then a replacement policy is used to select a victim. The victim is evicted from the buffer cache and if it is a dirty page, then it is also flushed back to the permanent storage.

Upon receiving a new request to read a page from storage, the *Buffer Layer* finds a free buffer cache slot and gets a pointer to it. The free buffer slot and corresponding pointer are shown in Figure 3a as *Pg Buffer* and *PgBufPtr*, respectively. The *Buffer Layer* then passes the pointer to the *File Layer*. Eventually the *File Layer* of PostgreSQL invokes the file read and write system calls implemented by the underlying FS.

For a read operation, PMFS copies the data block from NVM to a kernel buffer and then the kernel copies the requested data block to an internal buffer slot pointed by *PgBufPtr*. In the same way, two copies are made for write operation but in the opposite direction.

Hence, the SE of original PostgreSQL incurs two copy operations for each miss in the internal buffer cache. This is likely to become a large overhead for databases running queries on large datasets. Since PMFS can map the entire NVM address space into the kernel's virtual address space [25], the copy overhead can be avoided by making modifications in the SE. We applied these modifications in two incremental steps which are described in the following subsections.

B. *SE1: Using Memory Mapped I/O*

In the first step towards leveraging the features of NVM, we replaced the *File Layer* of PostgreSQL by a new layer named *MemMapped Layer*. As shown in Figure 3b, this layer still receives a pointer to a free buffer slot from the *Buffer Layer*, but instead of using the file I/O interface, it uses the memory mapped I/O interface of PMFS. We term this storage engine *SE1*.

Read Operation: When accessing a file for a read operation, we first open the file using the `open()` system call, same as in original PostgreSQL. Additionally, we create a mapping of the file using `mmap()`. Since we are using PMFS, `mmap()` returns a pointer to the mapping of the file stored in NVM. The implementation of `mmap()` by PMFS provides the application with direct access to mapped pages of files residing in NVM.

As a result we do not need to make an intermediate copy of the requested page from NVM into kernel buffers. We can directly copy the requested page into internal buffers of PostgreSQL by using an implicit `memcpy()` as shown in Figure 3b. When all requested operations on a given file are completed and it is not needed anymore, the file can be closed. Upon closing a file, we delete the mapping of the file by calling the `munmap()` function.

Write Operation: The same approach as in the read operation is used for writing data into a file. The file to be modified is first opened and a mapping is created using `mmap()`. The data to be written into the file is copied directly from internal buffers of PostgreSQL into NVM using `memcpy()`.

A SE with the above mentioned modifications does not create an intermediate copy of the data in kernel buffers. Hence we reduced the overhead to one copy operation for each miss in the internal buffer cache of PostgreSQL.

C. SE2: Direct Access to Mapped Files

In the second step of modifications to the SE, we replaced the *MemMapped Layer* of SE1 by the *PtrRedirection Layer* as shown in Figure 3c. Unlike the *MemMapped Layer*, the *PtrRedirection Layer* in SE2 receives the pointer to *PgBufPtr* (i.e *P2PgBufPtr*), which itself points to a free slot of buffer cache. In other words, *PtrRedirection Layer* receives a pointer to a pointer from the *Buffer Layer*.

Read Operation: When accessing a file for a read operation, we first open the file using `open()` system call, same as in original PostgreSQL and SE1. Additionally, we also create a mapping of the file using `mmap()`. Originally *PgBufPtr* points to a free slot in the internal buffer cache. Since `mmap()` makes the NVM mapped address space visible to the calling process, the *PtrRedirection Layer* simply redirects the *PgBufPtr* to point to the corresponding address of the file residing in NVM. Pointer redirection in case of read operation is shown by a black dashed arrow with the “Read” label in Figure 3c.

As a result of doing pointer redirection and the visibility of the NVM address space enabled by PMFS, we incur no copy overhead for read operations. This can represent a significant improvement, since read operations are predominant in queries that operate on large datasets.

Write Operation: PMFS provides direct write access for files residing in NVM. Since PostgreSQL is a multiprocess system, modifying the NVM-resident file can be dangerous. Direct write operations can leave the database in an inconsistent state.

To avoid this issue, SE2 performs two actions before modifying the actual content of the page and marking it as dirty. First, if the page is residing in NVM, it copies the page back from NVM into the corresponding slot of internal buffer cache, i.e. *Pg-Buffer*. Second, it undoes the redirection of *PgBufPtr* such that it again points to the corresponding slot in the buffer cache and not to the NVM mapped file. This is shown by a black dashed arrow with the “Write” label in Figure 3c. This way, SE2 ensures that each process updates only its local copy of the page.

V. Methodology

System-level evaluation for NVM technologies is challenging due to lack of real hardware. Software simulation infrastructures are a good fit to evaluate systems in which NVM is used as a DRAM replacement, or in conjunction with DRAM as a hybrid memory system. However, when using NVM as a permanent storage replacement, most software simulators fail to capture the details of the operating system, and comparisons against traditional disks are not possible due to the lack of proper simulation models for such devices. As the authors of PMFS [25] noted, an emulation platform is the best way to evaluate such a scenario.

TABLE I
Test machine characteristics.

Component	Description
Processor	Intel Xeon E5-2670 @ 2.60Ghz HT and TurboBoost disabled
Caches	Private: L1 32KB 4-way split I/D, L2 256KB 8-way Shared: L3 20MB 16-way
Memory	256GB DDR3-1600, 4 channels, delivering up to 51.5GB/s
OS	Linux Kernel 3.11.0 with PMFS support [25, 26]
Disk storage	Intel DC S3700 Series, 400GB, SATA 6Gb/s Read 500MBs/75k iops, Write 460MBs/36k iops
PMFS storage	224 GB of total DRAM

For this reason, we have set up an infrastructure similar to that used by the PMFS authors. We first recompiled the Linux kernel of our test machine with PMFS support. Using the *memmap* kernel command line option we reserve a physically contiguous area of the available DRAM at boot-time, which is later used to mount the PMFS partition. In other words, a portion of the DRAM holds the disk partition managed by PMFS and provides features similar to those of NVM, such as byte-addressability and lower latency compared to a disk. Table I lists the test machine characteristics. We configure the machine to have a 224GB PMFS partition, leaving 32GB of DRAM for normal main memory operation. A high-end SSD is used as regular disk storage.

A technological advantage of NVMs over traditional disks is their lower read access latencies. To quantify the performance impact this can have in query executions, we evaluate two baselines using unmodified PostgreSQL 9.5, (i) with the dataset stored in a regular high-end disk (*disk_base95*), and (ii) in the PMFS partition (*pmfs_base95*). In addition, we evaluate the modified storage engines - SE1 and SE2. These are run with the dataset stored on the PMFS partition and are termed as *pmfs_se1* and *pmfs_se2*, respectively.

To test these system configurations we employ decision support system (DSS) queries from the TPC-H [18] benchmark with a scale factor of 100, which leads to a dataset larger than 150GB when adding the appropriate indexes. Like most data intensive workloads, these queries are read dominant. Since DRAM read latencies are expected to be quite similar to projected NVM read latencies, the emulation platform employed provides good performance estimations. In our experiments, we report wall-clock query execution times as well as data obtained with performance counters using the *perf* toolset. We report results for 16 of the 22 TPC-H queries since some queries failed to complete under PMFS storage.

VI. Evaluation

In this section we show the performance impact that the modified storage engines (SE) have on kernel execution time and on wall-clock execution time for TPC-H queries. Later, we identify potential issues current DBMSs and applications in general may face in order to harness the benefits from directly accessing data stored in NVM memory.

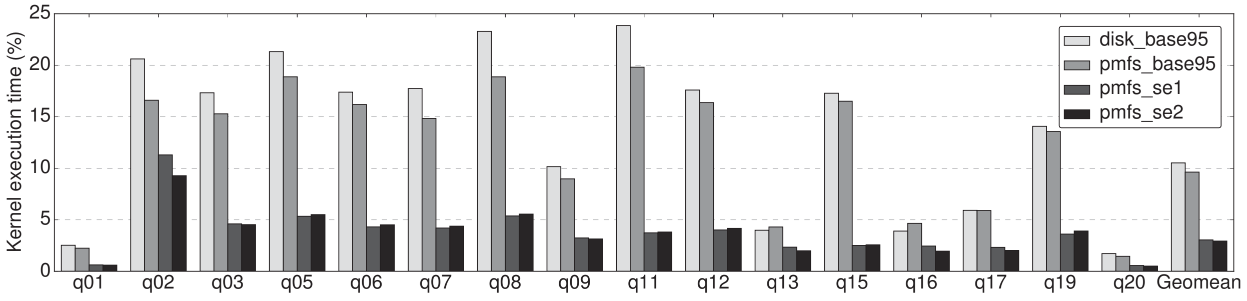


Fig. 4. Percentage of kernel execution time for each query.

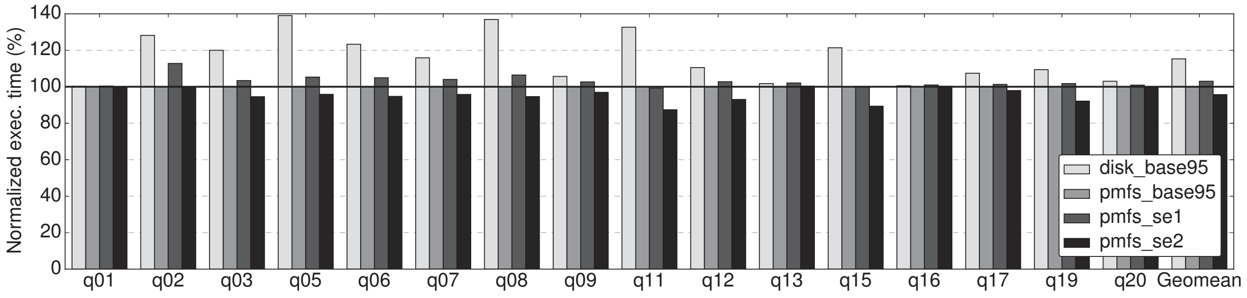


Fig. 5. Wall-clock execution time normalized to *pmfs_base95*.

A. *Performance Impact on Kernel Execution Time*

Figure 4 shows the percentage of kernel execution time for each of the evaluated queries running on the four evaluated systems. When using traditional file operations (e.g. `read()`), like those employed in unmodified PostgreSQL, the bulk of the work when accessing and reading data is done inside the kernel. As can be seen, the baseline systems spend a significant amount of the execution time in kernel space: up to 24% (Q11 - *disk_base95*) and 20% (Q11 - *pmfs_base95*), with an average of around 10%. The kernel space execution time is dominated by the time it takes to fetch data from the storage medium into a user-level buffer. These overheads are high in both disk and NVM storage, and are likely to increase as datasets grow in size.

However, when using *SE1* or *SE2*, this movement of data can be minimized or even avoided. For *pmfs_se1* we observe that the amount of time spent in kernel space decreases substantially and it is very similar to that observed for *pmfs_se2*. This is because the two systems are doing a similar amount of work on the kernel side, with the difference that *SE1* is doing an implicit `memcpy()` operation into a user-level buffer, but this is now done in user-level code. Overall, we see that the modified SEs are able to lower kernel space execution time significantly in most queries: Q02 to Q12, Q15, and Q19. A few queries show lower reductions because they operate over a small amount of data, e.g, Q1, Q13, Q16, and Q20. An important thing to note is that, for *SE1* and *SE2*, the kernel space time is likely to remain constant as datasets grow, as no work is done to fetch data.

B. *Query Performance Improvement*

Figure 5 shows wall-clock execution time for each query and evaluated system. The data is normalized to *pmfs_base95*. We observe that the benefits of moving from disk to a faster storage can be high for read intensive queries such as Q05 (40%), Q08 (37%), and Q11 (35%). However, for compute intensive queries, such as Q01 and Q16, the benefits are non-existent. On average, the overhead of using disk over PMFS storage is of 16%.

For *SE1*, the time reductions observed in terms of kernel execution time do not translate into reductions in overall query execution time. The main reason for this is the additional `memcpy()` operation performed to copy the data into the application buffer. In fact, we find that this operation in PMFS is sometimes slower than the original `read()` system call employed in the baseline, leading to a 3% slowdown on average.

When using *SE2* there is no data movement at the time of fetching data into an application-accessible memory region, due to the possibility to directly reference data stored in PMFS. However, this has a negative side effect when accessing the data for processing later on, as it has not been cached by the processing units. Therefore, the benefits of avoiding data movement to make it accessible are offset by the penalty to fetch this data close to the processing units for processing at a later stage. In order to mitigate this penalty, *SE2* incorporates a simple software prefetching scheme that tries to fetch in advance the next element to be processed within a data block. When compared to *pmfs_base95*, *SE2* is able to achieve significant performance improvements in read dominant queries such as Q11 (14.4%), Q15 (11.9%), and Q19 (8.6%). On average, *SE2* is 4.5% faster than *pmfs_base95* and 20.5% faster than *disk_base95*.

Figure 6 shows a classification of each cycle of execution as 'compute', if at least one instruction was committed during that cycle, or as 'stalled' otherwise. These categories are further

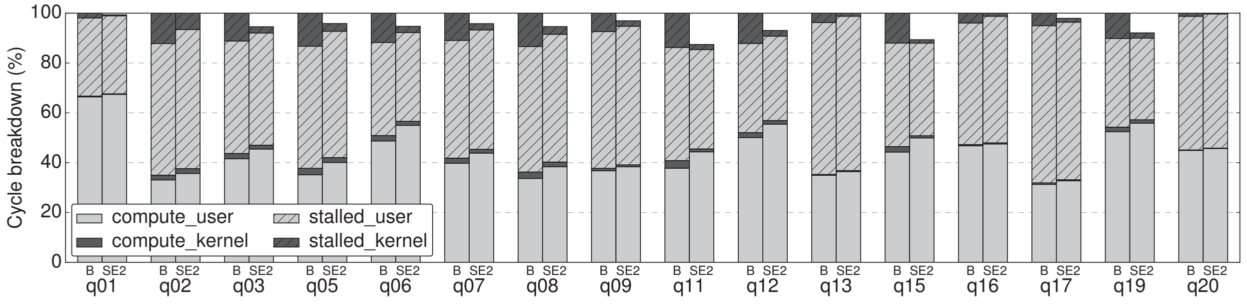


Fig. 6. Execution-time breakdown for compute and stalled cycles — B = *pmfs_base95*, SE2 = *pmfs_se2*.

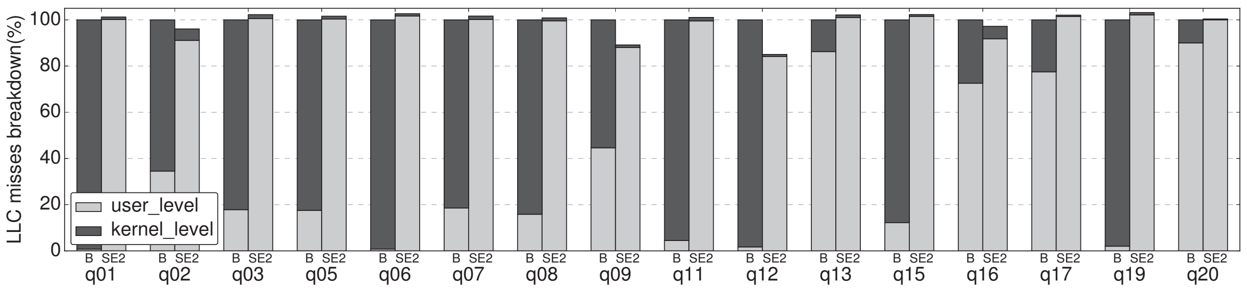


Fig. 7. Last-level cache (LLC) misses breakdown — B = *pmfs_base95*, SE2 = *pmfs_se2*.

broken down into user and kernel level cycles. Data is shown for *pmfs_base95* and *SE2*, normalized to the former. As can be seen, the *stalled_kernel* component correlates well with the kernel execution time shown in Figure 4, and this is the component that is reduced in *SE2* executions. However, we observe that for most queries some of the savings shift to *stalled_user* since data needs to be brought close to the processing unit when it is needed for processing. There are some exceptions, i.e., Q11, Q15, and Q19, for which the simple prefetching scheme is able to mitigate this fact effectively.

Figure 7 shows a breakdown of user and kernel last-level cache (LLC) misses. Here, we can clearly see how the number of LLC misses remains quite constant when comparing *pmfs_base95* and *SE2*, but the misses shift from kernel level to user level. Moreover, in our experiments we have observed that user level misses have a more negative impact in terms of performance because they happen when the data is actually needed for processing, and a full LLC miss penalty is paid for each data element. On the other hand, when moving larger data blocks to an application buffer, optimized functions are employed and the LLC miss penalties can be overlapped.

C. Discussion

We have shown that there is a mismatch between the potential performance benefits shown in Figure 4 and the actual benefits obtained shown in Figure 5. Direct access to memory regions holding persistent data can provide significant benefits, but this data needs to be close to the processing units when it is needed. To this end we have employed simple software prefetching schemes that have provided moderate average performance gains. However, carefully crafted ad-hoc software prefetching is challenging, and applications may not be designed in a way that makes it easy to hide long access latencies even with the use of prefetching, as happens with PostgreSQL. Moreover, such a solution is application and architecture dependent.

For these reasons, we advocate for the need to have additional software libraries and tools that aid programmability in such systems. These libraries could implement solutions like helper threads for prefetching particular data regions, effectively bringing data closer to the core (e.g., LLC) with small application interference. This approach would provide generic solutions for writing software that takes full advantage of the capabilities that NVM can offer.

VII. Related Work

Previous work on leveraging NVM for DBMS design can be divided into two categories: (i) employing NVM for whole database storage and (ii) for the logging components.

The work reported in [27, 32] reduces the impact of disk I/O on transaction throughput and response times by directly writing log records into an NVM component instead of flushing them to disk. Authors of [33] employ NVM for distributed logging on multi-core and multi-socket hardware to reduce contention of centralized logging with increasing system load. Pelley *et al.* [34] explore a two level hierarchy with DRAM and NVM, and study different recovery methods. Finally, Arulraj *et al.* [16] use a single tier memory hierarchy, i.e., without DRAM, and compare three different storage management architectures using an NVM-only system.

VIII. Conclusion

In this paper, we study the implications of employing NVM in the design of DBMSs. We discuss the possible options to incorporate NVM into the memory hierarchy of a DBMS computing system and conclude that, given the characteristics of NVM, a platform with a layer of DRAM where the disk is completely or partially replaced using NVM is a compelling scenario. Such an approach retains the programmability of current systems and allows direct access to the dataset stored in NVM. With this system configuration in mind we modified the PostgreSQL storage engine in two incremental steps - SE1 and SE2 - to better exploit the features offered by PMFS using memory mapped I/O.

Our evaluation shows that storing the database in NVM instead of disk for an unmodified version of PostgreSQL improves query execution time by up to 40%, with an average of 16%. Modifications to take advantage of NVM hardware improve the execution time by 20.5% on average as compared to disk storage. However, current design of database software proves to be a hurdle in maximizing the improvement. When comparing our baseline and *SE2* using PMFS, we achieve significant speedups of up to 14.4% in read dominated queries, but moderate average improvements of 4.5%.

We find that the limiting factor in achieving higher performance improvements is the fact that the data is not close to the processing units when it is needed for processing. This is a negative side effect of directly accessing data from NVM, rather than copying it into application buffers to make it accessible. This leads to long latency user level cache misses eating up the improvement achieved by avoiding expensive data movement operations. Therefore, software libraries that help mitigate this negative side effect are necessary to provide generic solutions to efficiently develop NVM-aware software.

Acknowledgments

The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement number 318633, the Ministry of Science and Technology of Spain under contract TIN2015-65316-P, and a HiPEAC collaboration grant awarded to Naveed Ul Mustafa.

References

[1] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian *et al.*, "Scuba: diving into data at facebook," *Proceedings of the VLDB Endowment*, 2013.

[2] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: data management for modern business applications," *ACM Sigmod Record*, 2012.

[3] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila, "IBM solidDB: In-Memory Database Optimized for Extreme Speed and Availability," *IEEE Data Engineering Bulletin*, 2013.

[4] J. Baulier, P. Bohannon, S. Gogate, S. Joshi, C. Gupta, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. Narayan *et al.*, "DataBlitz: A High Performance Main-Memory Storage Manager," in *Proceedings of the 24th VLDB Conference*, 1998.

[5] T. J. Lehman and M. J. Carey, "A study of index structures for main memory database management systems," in *Proceedings of the 12th VLDB Conference*, 1986.

[6] ——, "A Recovery Algorithm for a High-performance Memory-resident Database System," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 1987.

[7] I. Lee and H. Y. Yeom, "A single phase distributed commit protocol for main memory database systems," in *Proceedings of the Parallel and Distributed Processing Symposium*, 2001.

[8] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, 2010.

[9] H. Plattner, "SanssouciDB: An In-Memory Database for Processing Enterprise Workloads," in *BTW*, 2011.

[10] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd, "Efficient transaction processing in SAP HANA database: the end of a column store myth," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[11] J. Pisharath, A. Choudhary, and M. Kandemir, "Reducing energy consumption of queries in memory-resident database systems," in *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, 2004.

[12] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens, "Challenges and future directions for the scaling of dynamic random-access memory (dram)," *IBM Journal of Research and Development*, 2002.

[13] A. Driskill-Smith, "Latest advances and future prospects of stt-ram," in *Non-Volatile Memories Workshop*, 2010.

[14] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, 2008.

[15] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, 2008.

[16] 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*, 2015.

[17] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Computer Architecture News*, 2009.

[18] "The TPC-H Benchmark." [Online]. Available: http://www.tpc.org/tpch/

[19] K. Wang, J. Alzate, and P. K. Amiri, "Low-power non-volatile spintronic memory: STT-RAM and beyond," *Journal of Physics D: Applied Physics*, 2013.

[20] T. Perez and C. A. De Rose, "Non-volatile memory: Emerging technologies and their impacts on memory systems," *Technical Report, Porto Alegre*, 2010.

[21] Y. Huang, T. Liu, and C. J. Xue, "Register allocation for write activity minimization on non-volatile main memory for embedded systems," *Journal of Systems Architecture*, 2012.

[22] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ACM SIGARCH computer architecture news*, 2009.

[23] "3D XPoint Technology," 2016. [Online]. Available: https://www.micron.com/about/emerging-technologies/3d-xpoint-technology

[24] Intel, "Architecture Instruction Set Extensions Programming Reference," *Intel Corporation, Feb*, 2016.

[25] 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*, 2014.

[26] "Linux-pmfs." [Online]. Available: https://github.com/linux-pmfs/pmfs

[27] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proceedings of the VLDB Endowment*, 2014.

[28] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger, "Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.

[29] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, 1992.

[30] J. Gray, "The transaction concept: Virtues and limitations," in *Proceedings of the 7th International Conference on VLDB*, 1981.

[31] B. Momjian, *PostgreSQL: introduction and concepts*. Addison-Wesley New York, 2001, vol. 192.

[32] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," in *IEEE 27th International Conference on Data Engineering (ICDE)*, 2011.

[33] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proceedings of the VLDB Endowment*, 2014.

[34] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, "Storage management in the NVRAM era," *Proceedings of the VLDB Endowment*, 2013.