Path processing using Solid State Storage

Manos Athanassoulis[‡*] Bishwaranjan Bhattacharjee[◇] Mustafa Canim[◇] Kenneth A. Ross[◇§]

[‡]École Polytechnique Fédérale de Lausanne
manos.athanassoulis@epfl.ch

[◇]IBM Watson Research Labs
{*mustafa, bhatta, rossak*}*@us.ibm.com*

[§]Columbia University
kar@cs.columbia.edu

ABSTRACT

Recent advances in solid state technology have led to the introduction of Solid State Drives (SSDs). Todays SSDs store data persistently using NAND flash memory. While SSDs are more expensive than hard disks when measured in dollars per gigabyte, they are significantly cheaper when measured in dollars per random I/O per second. Additional storage technologies are under development, Phase Change Memory (PCM) being the next one to enter the marketplace. PCM is nonvolatile, it can be byte-addressable, and in future Multi Level Cell (MLC) devices, PCM is expected to be denser than DRAM. PCM has lower read and write latency compared to NAND flash memory, and it can endure orders of magnitude more write cycles before wearing out.

Recent research has shown that solid state devices can be particularly beneficial for latency-bound applications involving dependent reads. Latency-bound applications like path processing in the context of graph processing or Resource Description Framework (RDF) data processing are typical examples of these applications. We demonstrate via a custom graph benchmark that even an early prototype Phase Change Memory device can offer significant improvements over mature flash devices (1.5x - 2.5x speedup in response times). We take this observation further by building *Pythia*, a prototype RDF repository tailor-made for Solid State Storage to investigate the predicted benefits for these type of workloads that can be achieved in a properly designed RDF repository. We compare the performance of our repository against the state of the art RDF-3X repository in a limited set of tests and discuss the results. We finally compare the performance of our repository running on a PCM-based device against a state of the art flash device, showing that there is indeed significant gain to be achieved by using PCM for RDF processing.

1. INTRODUCTION

Solid State Storage (a.k.a. Storage Class Memory [24]) is here to stay. Today, a well known Solid State Storage technology is NAND flash. Another technology on the horizon is Phase Change Memory (PCM). Both can be used in chip form, for example, as a small storage element in a portable device. The read and write latencies of PCM cells are very low, already in the same ballpark as DRAM [21, 22]. For large scale storage, many chips can be packaged into a single device that provides the same functionality as disk drives, supporting the same basic APIs. SSDs can provide much faster random I/O than magnetic disks because there is no mechanical latency between requests. We focus here on database applications that demand enterprise level storage in this form factor.

NAND flash technology is relatively mature and represents the state-of-the art in the marketplace. Companies have been building storage devices out of flash chips for two decades, and one can find a huge variety of flash-based devices from consumer to enterprise storage. PCM is a relative newcomer, and until now there has been little opportunity to evaluate the performance characteristics of large scale PCM devices. The purpose of this paper is to provide insights on where solid state devices can offer a big advantage over traditional storage and to highlight possible differences between two representative technologies, flash and PCM.

Flash devices have superior random read performance compared to magnetic hard-drives but suffer from several limitations. First, there is a significant asymmetry in read and write performance. Second, only a limited number of updates can be applied on a flash device before it becomes unusable; this number is decreasing with newer generations of flash [3]. Third, writing on flash not only is much slower than reading and destructive of the device, but it has proven to interfere with the redirection software layers, known as Flash Translation Layers (FTL) [17].

PCM addresses some of these challenges. The endurance of PCM cells is significantly higher than NAND flash [4], although still not close to that of DRAM. Unlike NAND flash, PCM does not require the bulk erasure of large memory units before it can be rewritten. Moreover, while cost is still uncertain, for our purposes, we assume normal cell size competitiveness and standard volume economics will apply to this technology as it ramps into high volume.

The most pronounced benefit of solid state storage over hard disks is the difference in response time for random accesses. Hence, we identify *dependent reads* as an access pattern that has the potential for significant performance gains. Latency-bound applications like path processing [25] in the context of graph processing, or RDF-data processing are typical examples of applications with such access patterns. The Resource Descrption Framework (RDF) [6] data model is widely adopted for several on-line, scientific or knowledge-based datasets because of its simplicity in modelling and the variety of information it can represent.

We find that PCM-based storage is an important step towards better latency guarantees with no bandwidth penalties and we iden-

[*]The majority of this work was completed while the author was an intern at IBM T. J. Watson Research Center, Hawthorne, NY.

tify a trade-off between maximizing bandwidth and minimizing latency. In order to measure the headroom of performance benefit (decrease of response time) in long path queries we implement a simple benchmark and we compare the response times when using flash and PCM. We observe that PCM can yield 1.5x to 2.5x smaller response times for any bandwidth utilization without any graph-aware optimizations[1] some of which we leave for future work. We take this observation one step further and we design a new data layout suitable for RDF data and optimized for a solid state storage layer. The proposed layout increases the locality of related information and decreases the cost of graph traversals by storing more linkage information (i.e., metadata about how to navigate faster when a graph is traversed).

In this paper we show the benefits of path processing applications over data that is resident in solid state storage. First, we present a custom graph benchmark that is used to highlight the differences between two solid state technologies through two representative devices: a state-of-the-art flash device and an enterprise-level PCM prototype provided to us by Micron. Second, we develop a prototype RDF repository to show the benefits that RDF processing can have if it adopts PCM storage.

We have the following main contributions:

- We devise a custom benchmark to highlight the qualitative and quantitative differences between two representative solid state devices (flash and PCM).
- We find that PCM can natively support higher workload parallelization with near-zero latency penalty — an observation that can be used to shift the algorithm design.
- We find that applications with *dependent reads* are natural candidates for exploiting PCM-based devices. Our graph-based benchmark allows us to measure the benefit that path traversal queries can have from such devices.
- We develop a prototype RDF repository to illustrate a specific application that can benefit from PCM storage. Our prototype corroborates the opportunity analysis performed using the graph-based benchmark.

Our PCM device is a prototype, with a device driver having limited sophistication. It is possible that the commercial version of this PCM device could perform better when brought to market.

The rest of the paper is organized as follows. Section 2 summarizes prior work. Section 3 presents the custom graph benchmark and identifies potential performance improvements when using PCM in path processing. Section 4 presents the RDF repository and shows that it can achieve the anticipated improvements. Section 5 discusses further opportunities and limitations that PCM presents, and we conclude in Section 6.

2. RELATED WORK

In this section we present related work in terms of technology and data representation. Flash is the main representative of solid state storage technologies used today. In this paper we focus on understanding the differences between flash and PCM and the benefits that PCM can offer.

2.1 Phase Change Memory

PCM stores information using resistance in different states of phase change materials: amorphous and crystalline. The resistance in the amorphous state is about five orders of magnitude higher than the crystalline state, and it differentiates between 0 (high resistance) and 1 (low resistance) [21] [31]. Storing information on PCM is performed through two operations: set and reset. During the set operation, current is applied on the device for a sufficiently long period to crystallize the material. During the reset operation higher current is applied for shorter duration in order to melt the material and then cool it abruptly, leaving the material in the amorphous state. Unlike flash, PCM does not need the time consuming erase operation to write the new value. PCM devices can employ a simpler driver than the complex FTL that flash devices use to address the wear leveling and performance issues [11]. In the recent literature there is already a discussion about how to place PCM in the existing memory hierarchy. While proposed ideas [21] include placing PCM side-by-side DRAM as an alternative non-volatile main memory, or even using PCM as the main memory of the system, current prototype approaches consider PCM as a secondary storage device providing PCIe connectivity. There are three main reasons why this happens: (i) the endurance of each PCM cell is typically 10^6–10^8, which is higher than flash (10^4–10^5 with a decreasing trend [3]) but still not enough for a main memory device, (ii) the only available interface to date is PCIe, and (iii) the PCM technology is new, so the processors and the memory hierarchy do not yet have the appropriate interfaces for it.

The Moneta system is a hardware-implemented PCIe storage device with PCM emulation by DRAM [18, 19]. Performance studies on this emulation platform have highlighted the need for improved software I/O latency in the operating system and file system. The Onyx system [11] replaces the DRAM chips of Moneta with first-generation PCM chips,[2] yielding a total capacity of 10 GB. Onyx is capable of performing a 4KB random read in $38\mu s$ and a 4KB write request[3] in $179\mu s$. For a hash-table based workload, Onyx performed 21% better than an ioDrive, while the ioDrive performed 48% better than Onyx for a B-Tree based workload [11].

The software latency (as a result of operating system and file system overheads) is measured to be about $17\mu s$ [11]. On the other hand, the hardware latency for fetching a 4K page from a hard disk is on the order of milliseconds and for a high-end flash device is about $50\mu s$. Early PCM prototypes need as little as $20\mu s$ to read a 4K page increasing the software contribution in relative latency from 25% ($17\mu s$ out of $67\mu s$) for a flash device like FusionIO, to 46% ($17\mu s$ out of $37\mu s$) for a PCM prototype. Minimizing the impact of software latency is a relatively new research problem acknowledged by the community [11]. In [19], Caulfield et al. point out this problem and propose a storage hardware and software architecture to mitigate the overheads to take better advantage of low latency devices such as PCM. The architecture provides a private, virtualized interface for each process and moves file system protection checks into hardware. As a result, applications can access file data without operating system intervention, eliminating OS and file system costs entirely for most accesses. The experiments show that the new interface improves latency and bandwidth for 4K writes by 60% and 7.2x respectively, OLTP database transaction throughput by up to 2.0x, and Berkeley-DB throughput by up to 5.7x [19].

Jung et al. [28] ran the `fio` profiler over the same Micron PCM

[1]Such optimizations, in addition to the new data layout presented in this paper, include complementary pages comprised of auxiliary data structures such as: (i) cached attributes, (ii) aggregate links, (iii) node popularity, (iv) node priority and (v) reverse links.

[2]The PCM chips used by Onyx are the same as those used in our profiled device, but the devices themselves are different.

[3]The write numbers for Onyx use early completion, in which completion is signalled when the internal buffers have enough information to complete the write, but before the data is physically in the PCM. Early completion is also used by SSD devices, supported by large capacitors to ensure that the writes actually happen in case of a power failure.

prototype available to us and a popular flash device (OCZ Revodrive), showing qualitative differences between the PCM device and the flash device. The PCM device, unlike the flash device, shows no difference between latency for random and sequential accesses for different values of IO depth (number of concurrent outstanding requests) or page sizes.

Lee et al. [29] introduce a new In-Page Logging (IPL) design that uses PCRAM as a storage device for log records. They claim that the low latency and byte addressability of PCRAM can allow one to avoid the limitations of flash-only IPL. Papandreou et al. [31] present various programming schemes for multilevel storage in PCM. The proposed schemes are based on iterative write-and-verify algorithms that exploit the unique programming characteristics of PCM in order to achieve significant improvements in resistance-level packing density, robustness to cell variability, programming latency, energy per-bit and cell storage capacity. They present experimental results from PCM test-arrays to validate the proposed programming schemes. In addition, the reliability issues of multilevel PCM in terms of resistance drift and read noise are discussed.

2.2 RDF data model and RDF processing

The Resource Description Framework (RDF) [6] is today the standard format to store, encode and search machine readable information in the semantic web [34], as well as scientific [12], business and governmental data [1, 2]. This trend is strengthened by efforts like Linked Open Data [14] which to date consists of more than 25 billion RDF triples collecting data from more than 200 data sources.

2.2.1 RDF datasets and benchmarks

The wide adoption of RDF format has led to the design of numerous benchmarks and datasets [23] each one focusing on the different usage scenarios of RDF data. Benchmarks include:

- LUBM [26]: a benchmark consisting of university data.
- BSBM [16]: a benchmark built around an e-commerce data use case that models the search and navigation pattern of a consumer looking for a product.
- SP2Bench [33]: a benchmark based on the DBLP database of article publications, modeling several search patterns.
- Yago [9]: data from Wikipedia, Wordnet and GeoNames.
- UniProt [12]: a comprehensive, high-quality and freely accessible database comprised of protein sequences.
- DBpedia [15]: a dataset consisting of data extracted from Wikipedia and structured in order to make them easily accessible.

Several of the aforementioned benchmarks and workloads include path processing queries that could be inefficiently evaluated if the graph-like form of data is not taken into account. Viewing RDF data as relational data may make it more difficult to apply optimizations for graph-like data access patterns such as search. While each triple can conceptually be represented as a row, it has more information than a single row since it signifies a relation between two nodes of a graph. Two neighboring nodes may end up in the same search very often (a simple locality case) or nodes connected with two neighboring nodes may end up in the same search often. Moreover, a specific path between two nodes can be the core of the query. In this case, and especially if there are few or no branches in the path, evaluating a path of length k as $k-1$ joins can substantially increase the cost of answering one such query.

2.2.2 Storing RDF data

RDF data is comprised by statements, each represented as a triple of the form <*Subject, Predicate, Object*>. Each triple forms a statement which represents information about the $Subject$. In particular, $Subject$ is connected to the $Object$ using a specific $Predicate$ modelling either a connection[4] between the $Subject$ and the $Object$ or the value of a property of the $Subject$.[5] In fact, in RDF triples, sometimes, the $Predicate$ is called $Property$ and the $Object$ is called $Value$. We will maintain the terminology <*Subject, Predicate, Object*>.

RDF data form naturally sparse graphs but the underlying storage model in existing systems is not always tailored for graph processing [27]. There are two trends as far as how to physically store RDF data: (i) using as underlying storage a relational database system (either a row-store [5, 30] or a column store [10]) or (ii) design a native store [8, 35], a storage system designed particularly for RDF data which can be tailored to the needs of a specific workload. Support for RDF storage and processing assuming an underlying relational data management system is proposed from the industrial perspective [13] as well.

The RDF data layout we present in this paper is different in three ways. First, the approach proposed in this paper does not assume a traditional relational data layout but only the notion of variable sized tuples (having in effect a variable number of columns). Second, while our approach resembles prior art [13] as far as storing several triples with the same $Subject$ (or $Object$) physically close by, it is not bound by the limitations of relational storage, and it avoids repetition of information (e.g., for the $Objects$ that are connected to a specific $Subject$ with the same $Property$ the identifier of the $Property$ is stored only once). Third, we depart from the relational execution model, which is vital because graph traversals using relational storage lead to repetitive self-joins. We can support optimized graph-traversal algorithms without paying the overheads that come with relational query evaluation.

3. PATH PROCESSING

Current PCM prototypes behave as a "better" flash [28], in the sense that they have faster and more stable reads. We argue that the best way to make the most of this behavior is in the domain of applications with *dependent reads*. Hence, we create a simple benchmark that performs path traversals over randomly created graphs to showcase the potential benefits using PCM as secondary storage for such applications. (More information about performance characteristics of the devices we used can be found in Appendix B.)

Table 1: Description of the benchmark

Dataset	Randomly generated graph
Degree	Randomly between 3 and 30
# nodes	1.3M (approximately)
Size on disk	5GB

Custom graph-based benchmark. We create a benchmark that we call the *Custom graph-based benchmark*. We model path traversal queries by graph traversal over a custom built graph. The graph (see description in Table 1) is stored in fixed-size pages (each page has one node) and the total size of the graph is 5GB. Each node has an an arbitrary number of edges (between 3 and 30). The path

[4]For example, the triple <*Alice, isFriendWith, Bob*> shows that a friendship connection between $Alice$ and Bob exists.

[5]For example, the triple <*Alice, birthday, 01/04/1980*> shows that the property $birthday$ of $Alice$ has value $01/04/1980$.

Table 2: Hardware specification of the PCM device

Brand	Micron
PCM type	Single Level Cell (SLC)
Integration	90nm
Size	12GB
Interface	PCI-Express 1.1x8
Read Bandwidth	800MB/s (random 4K)
Write Bandwidth	40MB/s (random 4K)
H/W Read Latency	$20\mu s$ (4K)
H/W+S/W Read Latency	$36\mu s$ (4K)
H/W Write Latency	$250\mu s$ (4K)
H/W+S/W Write Latency	$386\mu s$ (4K)
Endurance	10^6 write cycles per cell

traversal queries are implemented as link following traversals of a random edge in each step. Each query starts from a randomly selected node of the graph and it follows at random one of the descendant nodes. When multiple queries are executed concurrently, because of the absence of buffering, locality will not yield any performance benefits. Each query keeps reading a descendant node as long as the maximum length is not reached.

Table 3: Parameters used for the custom graph benchmark.

Path length	2, 4, 10, 100 nodes per query
Concurrent threads	1, 2, 4, 8, 16, 32, 64, 96, 128, 192
Page Size	4K, 8K, 16K, 32K
Page processing time	$0\mu s$, $50\mu s$, $100\mu s$

Experimental setup. We use a 74GB FusionIO ioDrive (SLC) [7] and a 12GB Micron PCM prototype (SLC). The PCM device offers 800MB/s maximum read bandwidth, $20\mu s$ hardware read latency[6] (for 4K reads) and $250\mu s$ hardware write latency (for 4K writes), while the endurance is estimated to be 10^6 write cycles. While PCM chips can be byte-addressable the PCI-based PCM prototype avilable to us uses 4KB pages for reads and writes[7]. The flash device offers 700MB/s read bandwidth (for 16K accesses) and hardware read latency as low as $50\mu s$ in the best case. The details about the two devices can be found in Tables 2 and 4. The system used for the experiments was a 24-core 2.67GHz Intel Xeon X5650 server with the 74GB ioDrive and the 12GB PCM device. The operating system is Ubuntu Linux with the 2.6.32-28-generic x86_64 kernel and the total size of available RAM is 32GB.

Experimental evaluation. We present a set of experiments based on the custom graph benchmark. We compare flash and PCM technology as secondary storage for path queries. We vary the page size (and consequently node size), the length of the path, the number of concurrent requests and the page processing time, all of them summarized in Table 3. The variation of the values for each parameter plays a different role: different path lengths help us reveal if there is a cumulative effect when a query over a longer path is evaluated; different numbers of concurrent threads show the impact of multiple I/O requests on flash and PCM; different page sizes (a.k.a. access granularity) signify which is the best mode of operation for each device; and varying page processing time helps us understand what is the benefit when the workload is "less" I/O-bound. We

[6]Software read latency is about 16–17μs, which is negligible compared to magnetic disk I/O latency, but is close to 50% of the total latency for technologies like PCM.

[7]It is noteworthy that, unless the s/w stack is optimized, smaller page accesses will make the impact of s/w latency a bigger issue as the percentage of s/w latency over the total page read latency will increase [11].

Table 4: Hardware specification of the flash device

Brand	FusionIO
NAND type	Single Level Cell (SLC)
Integration	30-39nm
Size	74GB
Interface	PCI-Express x4
Read Bandwidth	700MB/s (random 16K)
Write Bandwidth	550MB/s (random 16K)
Mixed Bandwidth	370MB/s (70R/30W random 4K mix)
IOPS	88,000 (70R/30W random 4K mix)
H/W Read Latency	$50\mu s$ Read (512b)
H/W+S/W Read Latency	$72\mu s$ (4K)
H/W+S/W Write Latency	$241\mu s$ (4K)
Endurance	24yrs (@ 5TB write-erase/day)

observe that varying the path length leads to the same speedup because the reduced latency remains the same throughout the execution of the query regardless of the length of the query. Moreover, higher page processing time reduces gradually the benefit of PCM, which is expected since the longer IO time for the flash device is amortized over the page processing time. In particular, when page processing is increased from $0\mu s$ to $100\mu s$ the maximum speedup that PCM offers is reduced by 20%.

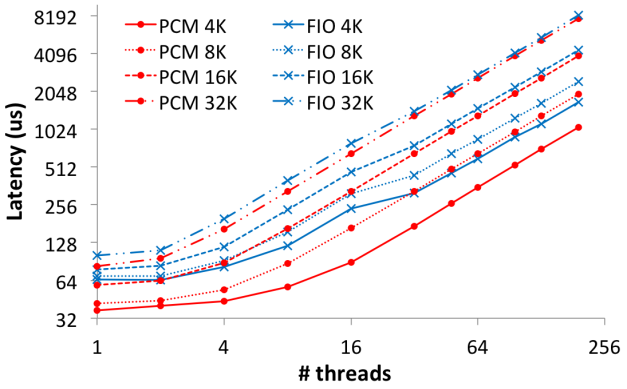

Figure 1: Latency for 4K, 8K, 16K, 32K page size when varying concurrent queries. Note the logarithmic scale on both axes.

Figure 1 presents the average latency and Figure 2 the average bandwidth during the execution of the custom graph benchmark, when we vary the page size and the number of concurrent threads. The page processing time for the presented experiments is set to $0\mu s$ and the path length to 100 nodes.

Figure 1 shows the average read latency per I/O request (y-axis) as a function of the number of concurrent threads issuing queries (x-axis) for different page sizes (different graphs). The red lines correspond to PCM and the blue lines to flash. In all four cases (page size equal to 4K, 8K, 16K, 32K) PCM shows lower I/O latency but the best case is when the page size is smallest, i.e., 4K. Extrapolating this pattern we anticipate that when PCM devices have smaller access granularity (512-byte pages are part of the roadmap) the benefit will become even larger.

In Figure 2 we see the bandwidth achieved when running the custom graph benchmark. Similarly to the previous graphs, the x-axis of the different graphs represents the number of threads issuing concurrent requests, the y-axis is bandwidth and different graphs correspond to different page sizes. The capacity for both devices is roughly 800 MB/s. The PCM device reaches very close

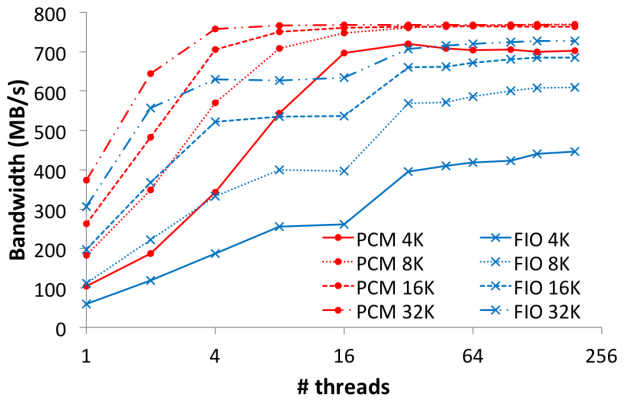

Figure 2: Bandwidth for 4K, 8K, 16K, 32K page size when varying concurrent queries

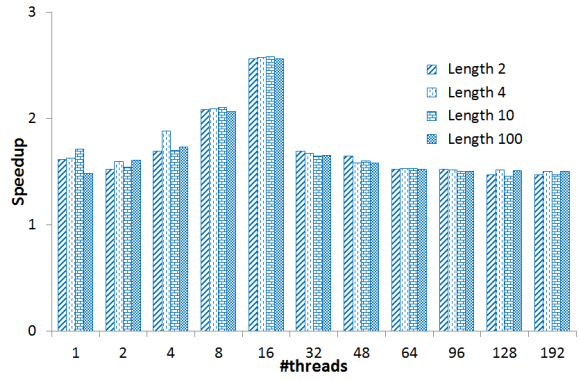

Figure 3: Custom path processing benchmark: Speedup of PCM over flash

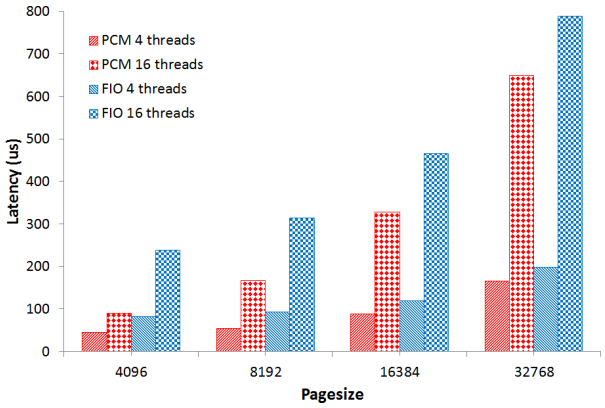

Figure 4: Latency per page request size for PCM and flash, varying page size and number of concurrent threads issuing requests

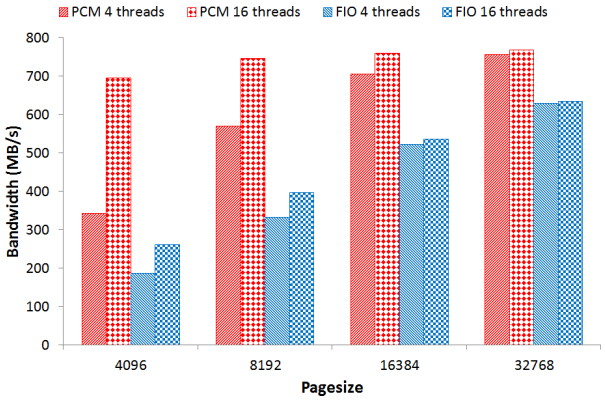

Figure 5: Bandwidth for PCM and flash, varying page size and number of concurrent threads issuing requests

to the maximum sustained throughput (close to 800MB/s) with 4K pages when 16 queries are issued concurrently. When the page size is increased the maximum can be achieved when 4-8 queries are issued concurrently. When increasing the page size and the number of concurrent queries any additional (small) benefit has a high cost in increased latency. On the other hand, flash has qualitatively different behavior. For each page size the maximal bandwidth is achieved with 32 concurrent queries and a bigger page size is needed for a better result. Using the *iostat* and *sar* Linux tools we were able to verify that when 32 queries are issued concurrently, the full potential of the flash device can be achieved by increasing the I/O queue size to 64 (which is the maximum possible) without having delays due to queuing. Thus, having 32 concurrent queries is considered to be the sweet spot where flash has the optimal bandwidth utilization. Going back to the latency figures we can now explain the small bump for 16 threads. In fact, the observed behavior is not a bump at 16 threads but an optimal behavior at 32 threads. This phenomenon is highlighted in Figure 3.

In Figure 3 we present the speedup of the query response time for different values of path length and number of concurrently issued queries. The speedup varies between 1.5x and 2.5x having the maximum number of 16 threads. We observe as well that the length of the query does not play any important role. The sudden drop in speedup for 32 threads is attributed to the previously described sweet spot for flash for this number of concurrent queries.

When we increased page processing time from $0\mu s$ to $100\mu s$ the maximum speedup was reduced from 2.5x to 2.0x. Figure 4 shows a different way to read the data from Figure 1. On the x-axis we vary the page size (4K, 8K, 16K, 32K) and on the y-axis we present the latency per I/O when we run the custom graph benchmark. The solid bars correspond to experiments using 4 threads and the checkered bars to experiments using 16 threads, the red color represents PCM and the blue color flash. There are several messages to be taken from this graph. First, when page size is equal to 4K (the best setting for both devices) using PCM leads to the highest benefit in latency (more than 2x speedup). Secondly, we observe that the average latency per I/O when using PCM with 16 threads is almost the same (7% higher) compared to average latency per I/O when using flash with 4 threads. In other words we can have 4 times more queries accessing 4 times more data with the similar latency per group of I/O (which correponds to 4 reads for flash and 16 reads for PCM). This observation can be used to create search algorithms for PCM which can take advantage of *near-zero penalty concurrent reads*, which we outline in Section 5. Similarly, for 8 concurrent threads reading 4K pages from PCM the latency is $57\mu s$ and for 2 concurrent threads reading 4K pages from flash the latency is $65\mu s$ allowing PCM to fetch 4 times more data in less time. Finally, we see that the benefits from PCM decrease as we increase page size, which help us make the case that we should expect even higher benefits when PCM devices offer finer access granularity.

Figure 5 presents the bandwidth as a function of page size. Even for 4K page size, PCM can achieve 22% higher bandwidth with 4 threads than the flash device with 16 threads. If we compare the 16-thread cases, we can almost saturate PCM with 4K page size. On the other hand, we are unable to saturate the flash device even with 32K page size. The last two figures demonstrate that a PCM device can show important latency and bandwidth benefits, relative to flash, for workloads with dependent reads.

4. RDF PROCESSING ON PCM

In this section we describe our prototype RDF repository, called *Pythia*. We identify the need to design an RDF-processing system which takes into account the graph-structure of the data and has the infrastructure needed to support any query over RDF data. Pythia is based on the notion of an *RDF-tuple*.

4.1 The RDF-tuple

An *RDF-tuple* is a hierarchical tree-like representation of a set of triples given an ordering of subject, predicate, and object. In Pythia, we will store RDF-tuples for two complementatry hierarchies: the subject/predicate/object (SPO) hierarchy and the object/predicate/subject (OPS) hierarchy. Each triple will thus be represented in two places, the SPO-store and the OPS-store.

In the SPO store, the root of the tree contains the information of the common subject of all triples. The children of the subject-node are the property-nodes, one for each property that is connected with the given subject. For each property-node, its children are in turn the identifiers of object-nodes that are connected with the subject of the root node through the property of its parent property-node. The RDF-tuple design allows us to locate within a single page[8] the most significant information for any given subject. Furthermore, it reduces redundancy by omitting repeated instances of the subject and predicate resources. Conceptually, the transformation of RDF triples to an RDF-tuple in the SPO-store is depicted in Figure 6. The OPS-store is analogous.

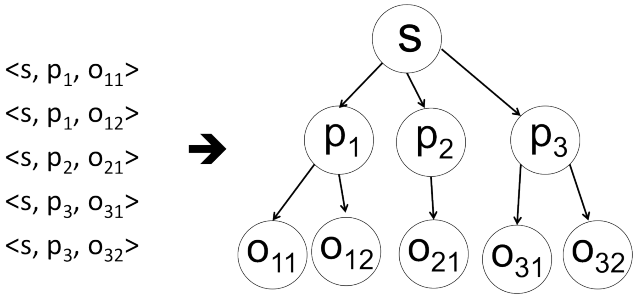

Figure 6: RDF-tuple in the SPO-store

We chose to materialize the SPO and OPS hierarchy orders in Pythia, but not other orders such as PSO. This choice is motivated by our observation that the large majority of use cases in the benchmarks of Section 2.2 need subject-to-object or object-to-subject traversal. Usually the query had a subject (or object) in hand and needed to find connections of given types to other objects (subjects). Rarely did the query specify a predicate and ask for all object/subject pairs. We thus avoid the extra storage requirements of representing additional versions of the data, at the cost of occasionally needing a more expensive plan for a predicate-oriented query.

[8]See Section 4.2 for a discussion of tuples that don't fit in a page.

We envision RDF-tuples to be stored as a tuple with variable length in a database page containing many such tuples. Figure 7 shows how an RDF-tuple is laid out within a page. We employ a

LEN	Sptr	nofP	Optr	dicID	Optr	dicID	…	…	…
…	nofO	local	ORpt	pID	tID	local	ORpt	pID	tID
…	…	nofO	local	ORpt	pID	tID	…	…	…
<Subject>			<Object1_1>				<Object1_2>		
…		…		…		<Object2_1>			…

Figure 7: RDF-tuple layout

standard slotted page scheme with variable size tuples, but the internal tuple organization is different from prior work. An RDF-tuple is organized in two parts: the metadata part (first three lines of Figure 7) and the resource part (the remaining lines). In the metadata part the above tree structure is stored with internal tuple pointers (offset) to the representation of the nodes and the resources. The offsets to the resources point to the appropriate locations in the resource part of the tuple. In more detail, the metadata part consists of the following variables:

- the length of the tuple,
- the offset of the subject's resource,
- the number of predicates,
- for each predicate the offset of the predicates resource and an offset to the objects (for the combination of subject and predicate),
- for each predicate's objects: the number of objects, the object's resource offset, a flag saying whether the resource is stored locally and the page id and tuple id for every object as a subject.

In this representation there are several possible optimizations. For example, the resource of the predicate can typically be stored in a dictionary because in every dataset we analyzed the number of different predicates is very small (in the order of hundreds). Moreover, in the case that the object is a literal value (e.g., a string constant) the value can be stored in a separate table with a unique identifier.

4.2 The Internals of Pythia

The SPO-store and the OPS-store are each enhanced by a hash index; on the Subject for the SPO store and on the Object for the OPS store. There is a separate repository for "very large objects" (VLOBs), i.e., RDF-tuples which require more storage than what is available in a single page. VLOBs are important when the dataset includes objects that are connected with many subjects and create a huge RDF-tuple for the OPS store. (Imagine the object "human"; a variety of other subjects will be linked with this using the predicate "is-subclass-of", leading to a huge OPS RDF-tuple.) Additionally, we employ two dictionaries mainly for compression of the RDF-tuples: the *Literals dictionary* and the *Predicate dictionary*. The former is used when a triple consists of a literal value as an object and the latter is used to avoid storing the resource of the predicates multiple times in the two stores. In the majority of RDF datasets the variability of predicates is relatively low leading to a small Predicte dictionary, always small enough to fit in memory. On the other hand, different datasets may lead to a huge Literal dictionary and may need a more complex solution.

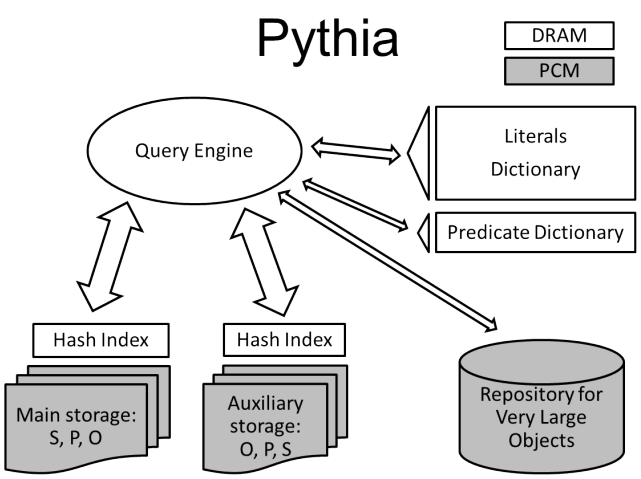

Figure 8: Pythia Architecture

Figure 8 presents the architecture of the prototype RDF repository Pythia we designed and experimented with. The greyed parts (SPO-store, OPS-store and VLOB-store) reside in secondary storage. The remaining parts are stored in persistent secondary storage, but are maintained in RAM during the time the system is running.

4.3 Experimental Workload

We use the popular dataset Yago2 [9] to investigate the benefits of the proposed approach in an environment with *dependent reads*. Yago2 is a semantic knowledge base derived mainly from Wikipedia, WordNet and GeoNames. Using the RDF-tuple design for the SPO- and OPS-stores, we store the initial 2.3GB of raw data (corresponding to 10M entries and 460M facts) into a 1.5GB uncompressed database (the SPO-store and OPS-store together account for 1.3GB, and VLOBs account for 192MB). The large objects can be aggressively shrunk down by employing page-level compression, an optimization which we leave for future work. When the system operates the in-memory structures require 121MB of RAM for the hash indexes and 569MB of RAM for the dictionaries, almost all of it for the Literals dictionary. Another future optimization for the literals is to take into account the type and store them in a type-friendly compressible way. In the SPO store more than 99% of the RDF-tuples fit within a 4K page.

We hard-code and experiment with six queries over the Yago2 dataset:

1. Find all citizens of a *country* that have a specific *gender*.
2. Find all events of a specific *type* that a given *country* participated in.
3. Find all movies of a specific *type* that a given *actor* participated in.
4. Find all places of a specific *type* that are located in a given *country* (or in a place ultimately located in this *country*).
5. Find all ancestors of a given *person*.
6. Find all ancestors with a specific *gender* of a given *person*.

The equivalent SPARQL code for these queries can be found in Appendix A.

The first three queries search for information of the given subject or object; in other words they are searching for *local knowledge*. The last three queries require a traversal the graph in an iterative mode to compute the correct answer. The last three queries typically require dependent reads.

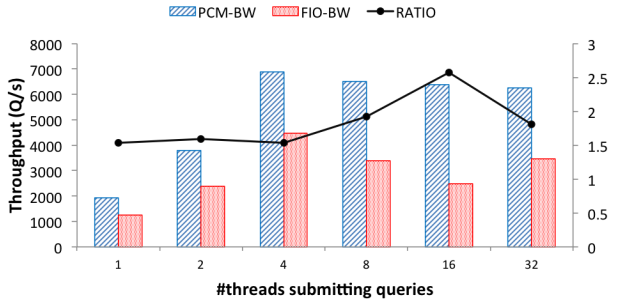

Figure 9: Bandwidth for Pythia for 1, 2, 4, 8, 16, 32 concurrent clients submitting queries

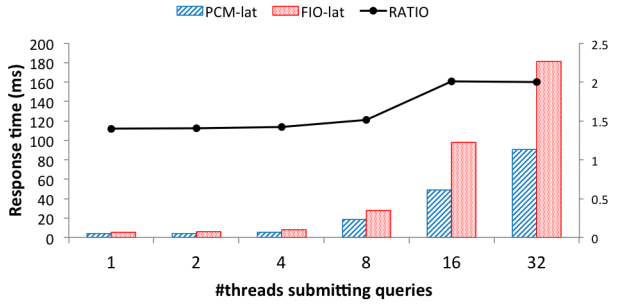

Figure 10: Latency for Pythia for 1, 2, 4, 8, 16, 32 concurrent clients submitting queries

4.4 Experimental Evaluation

We used the same 24-core Intel Xeon X5650 described previously. Similarly to the previous experiments, we experimented using as backend storage a state-of-the-art 80GB FusionIO flash device (SLC) and a 12GB Micron PCM device prototype. The workload for the first two experiments was a mix of the six aforementioned queries with randomized parameters. We executed the workload using either flash or PCM as main storage in anticipation of a benefit similar to what our custom graph benchmark showed in Section 4.

Figure 9 shows the throughput achieved by Pythia when Flash (red bars) or PCM (blue bars) is used for secondary storage for a varying number of threads submitting queries concurrently. The black line shows the relative speedup when PCM is employed. In this experiment we corroborate the performance benefit achieved in the simple benchmark with a more realistic system and dataset. Both devices scale close to linearly until 4 threads (7KQ/s for PCM and 4.5KQ/s for flash leading to 1.56x speedup), while for higher number the PCM device is more stable showing speedup from 1.8x to 2.6x.

Figure 10 shows the response time achieved by Pythia when Flash (red bars) or PCM (blue bars) is used for secondary storage for a varying number of threads submitting queries concurrently. The black line shows the relative speedup when PCM is employed. In terms of response time PCM is uniformly faster than flash from 1.5x to 2.0x.

4.5 Comparison with RDF-3X

In this section we compare Pythia with RDF-3X [30] showing that our approach is competitive against the research prototype con-

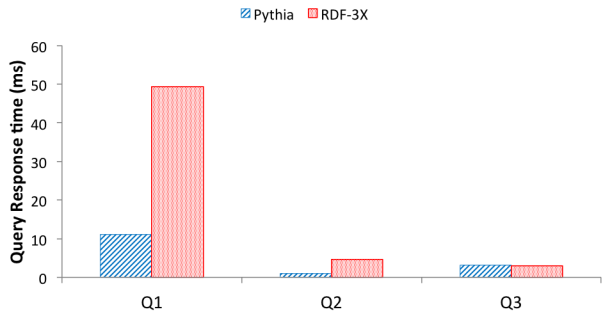

Figure 11: Query latency using Pythia for Q1, Q2 and Q3

sidered to be the current state-of-the-art. We instrument the source code of RDF-3X with time measurement code and we time the pure execution time of the query engine of RDF-3X without taking into account the parsing or optimization phases. We compare the response time of three queries when we execute them using Pythia and RDF-3X. We store the Yago2 data on PCM and we use the following three queries to understand whether Pythia is competitive with an established RDF repository.

Q1. Find all male citizens of Greece.

Q2. Find all OECD member economies that Switzerland deals with.

Q3. Find all mafia films that Al Pacino acted in.

Figure 11 shows that Pythia (blue bars) performs equally well or faster than RDF-3X (red bars) for all three queries. For Q1 Pythia is 4.45x faster, which can be attributed to the fact that the gender information for every person is stored in the same page as the rest of the person's information, thus incurring no further IOs to read it. In Q2 the benefit of Pythia is about 4.69x for the same reason: the information about whether a country is an OECD member is stored in the same page with the country. The third query incurs a higher number of IOs in both cases because we have to touch both the SPO- and OPS-stores, hence Pythia and RDF-3X perform almost the same (RDF-3X is 3% faster).

5. PCM OPPORTUNITIES

Any device that can handle concurrent requests in parallel can be kept busy if there are enough independent I/O streams. Under such conditions, the device will saturate its bandwidth capabilities. When there are too few independent I/O streams to saturate the device with a single I/O request per stream, one might allow each I/O stream to submit multiple concurrent requests.

When the access pattern requires dependent reads, identifying concurrent I/O units for a single stream can be challenging. Roh et al. [32] consider a B-tree search when multiple individual keys are being searched at the same time. If multiple child nodes of a single internal node need to be searched, I/Os for the child nodes can be submitted concurrently. This is a dependent-read pattern in which multiple subsequent reads may be dependent on a single initial read. Roh et al. demonstrate performance improvements for this workload on several flash devices. They implemented an I/O interface that allows the submission of several I/O requests with a single system call, to reduce software and hardware overheads.

Both PCM devices and flash devices can handle a limited number of concurrent requests with a minor latency penalty. For our PCM device running with 8 concurrent threads, we can achieve roughly double the bandwidth and half the latency of the flash device, without significantly increasing the latency beyond what is observed for single-threaded access. Thus our PCM device has the potential to outperform flash when I/O streams can submit concurrent I/O requests.

5.1 Algorithm redesign

A graph-processing example of such a workload is *best first search*. In best first search, the priority of nodes for subsequent exploration is determined by an application-dependent heuristic, and nodes at the current exploration fringe are visited in priority order. (Breadth first search and depth first search are special cases in which the priority function is suitably chosen.) On a device that efficiently supports n concurrent I/O requests, we can submit the top n items from the priority queue as an I/O batch, effectively parallelizing the search. The children of the newly fetched nodes are inserted into the priority queue for the next round of the search algorithm.

6. CONCLUSIONS

Commercial scale PCM technology is relatively new, but we see that it is already competitive against mature flash devices. A PCM device with a very basic device driver can outperform a mature flash rival in terms of the read latency and read bandwidth of a single device. Applications like graph processing can take advantage of PCM naturally because they employ *dependent reads*. Our experimentation with a custom graph benchmark shows that using an early prototype PCM device can yield significant performance benefits over flash when running path traversal queries (1.5x-2.5x).

We have described our experiences and observations of a real PCM prototype. At least for the near future, PCM-based devices are going to be "better" flash devices, rather than a new incomparably fast and more efficient storage technology. The limits of interconnecting interfaces, and the overhead of the software stack used to access the devices, are significant determinants of overall system performance. As new generations of PCM become available, and as new software technologies are developed to minimize overheads, PCM device performance and access granularity constraints are likely to improve.

7. ACKNOWLEDGMENTS

The authors would like to thank Micheal Tsao of IBM T.J. Watson Research for his help in setting up the machine used in the experiments, and Micron Inc. for providing access to an early PCM prototype.

8. REFERENCES

[1] Data.gov. http://www.data.gov/.

[2] Data.gov.uk. http://www.data.gov.uk/.

[3] NAND Flash Trends for SSD/Enterprise. http://www.bswd.com/FMS10/FMS10-Abraham.pdf.

[4] Ovonic Unified Memory, page 29. http://ovonyx.com/technology/technology.pdf.

[5] RDF Data Model in Oracle. http://download.oracle.com/otndocs/tech/semantic_web/pdf/w3d_rdf_data_model.pdf.

[6] Resource Description Framework (RDF). http://www.w3.org/RDF/.

[7] The FusionIO drive. Technical specifications available at: http://www.fusionio.com/data-sheets/.

[8] Virtuoso RDF. http://www.openlinksw.com/dataspace/dav/wiki/Main/VOSRDF.

[9] YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. http://www.mpi-inf.mpg.de/yago-naga/yago/.
[10] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, 2007.
[11] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A Protoype Phase Change Memory Storage Array. In *HotStorage*, 2011.
[12] R. Apweiler et al. Uniprot: the universal protein knowledgebase. *Nucleic Acids Research*, 32(Database-Issue), 2004.
[13] B. Bhattacharjee, K. Srinivas, and O. Udrea. Method and system to store RDF data in a relational store. Published Patent Application, Sept. 2011. US 2011/0225167 A1.
[14] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3), 2009.
[15] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - A crystallization point for the Web of Data. *J. Web Sem.*, 7(3), 2009.
[16] C. Bizer and A. Schultz. The Berlin SPARQL Benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2), 2009.
[17] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.
[18] A. M. Caulfield et al. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *HiPC*, 2010.
[19] A. M. Caulfield, T. I. Mollov, L. Eisner, A. De, J. Coburn, and S. Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *ASPLOS*, 2012.
[20] S. Chen. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *SIGMOD*, 2009.
[21] S. Chen, P. B. Gibbons, and S. Nath. Rethinking Database Algorithms for Phase Change Memory. In *CIDR*, 2011.
[22] E. Doller. Phase change memory and its impacts on memory hierarchy. CMU PDL presentation 2009: http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.
[23] S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *SIGMOD*, 2011.
[24] R. F. Freitas. Storage class memory: technology, systems and applications. In *SIGMOD*, 2009.
[25] A. Gubichev and T. Neumann. Path Query Processing on Very Large RDF Graphs. In *WebDB*, 2011.
[26] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3), 2005.
[27] O. Hassanzadeh, T. Kementsietsidis, and Y. Velegrakis. Publishing relational data in the semantic web. *ESWC*, 2011. http://db.disi.unitn.eu/pages/Rel2RDFTutorial2011/.
[28] J.-Y. Jung, K. Ireland, J. Ouyang, B. Childers, S. Cho, R. Melhem, D. Mosses, et al. Characterizing a Real PCM Storage Device (poster). *NVMW*, 2011.
[29] S.-W. Lee, B. Moon, C. Park, J. Y. Hwang, and K. Kim. Accelerating In-Page Logging with Non-Volatile Memory. *IEEE Data Eng. Bull.*, 33(4):41–47, 2010.
[30] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 2008.
[31] N. Papandreou et al. Programming algorithms for multilevel phase-change memory. In *ISCAS*, 2011.
[32] H. Roh et al. B+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. *PVLDB*, 5(4), 2012.
[33] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. *CoRR*, abs/0806.4627, 2008.
[34] N. Shadbolt, T. Berners-Lee, and W. Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3), 2006.
[35] K. Wilkinson. Jena property table implementation. In *SSWS*, 2006.

APPENDIX

A. SPARQL CODE FOR TEST QUERIES

In this section we present the equivalent SPARQL version of the queries implemented during the evaluation of Pythia in Section 4. These queries are used to compare against the research prototype RDF-3X [30] using the *pathfilter* operator introduced in the literature [25].

1. Find all citizens of a *country* that have a specific *gender*.
   SPARQL: select ?s where { ?s <isCitizeonOf> *country*. ?s <hasGender> *gender* }
2. Find all *type* events that a *country* participated in.
   SPARQL: select ?s where { *country* <participatedIn> ?s. ?s <type> *eventType* }
3. Find all *type* movies that an *actor* participated in.
   SPARQL: select ?s where { ?s <type> *movieType*. *actor* <actedIn> ?s }
4. Find all places of a specific *type* that are located in a *country* (or in a place ultimately located in this *country*).
   SPARQL: select ?s where { ?s <type> *placeType*. ?s ?path *country*. pathfilter(containsOnly(??path, <isLocatedIn>)) }
5. Find all ancestors of a *person*.
   SPARQL: select ?s where { ?s ??path *person*. pathfilter(containsOnly(??path, <hasChild>)) }
6. Find all ancestors having gender *gender* of a *person*.
   SPARQL: select ?s where { ?s ??path *person*. ?s <hasGender> *gender*. pathfilter(containsOnly(??path, <hasChild>)) }

Q1. Find all *male* citizens of *Greece*.
   select ?s where { ?s <hasGender> <male>. ?s <isCitizenOf> <Greece> }

Q2. Find all *OECD member economies* that *Switzerland* deals with.
   SPARQL: select ?s where
   { ?s <type> *wikicategory_OECD_member_economies*.
   <Switzerland> <dealsWith> ?s }

Q3. Find all *mafia* films that *Al Pacino* acted in.
   SPARQL: select ?s where { ?s <type> *wikicategory_Mafia_films*.
   <Al_Pacino> <actedIn> ?s }

B. FLASH VS PCM

In this section we perform a head-to-head comparison between the flash and the PCM devices that we used throughout our experimentation.

Experimental methodology. The experimental setup is the same as described in Section 3. We show that PCM technology has already an important advantage for read-only workloads. We present several experiments comparing the read latency of the aforementioned devices. In particular, we measure the latency for direct access to the devices bypassing any caching (operating system or file system) using the O_DIRECT and O_SYNC flags in a custom C implementation.

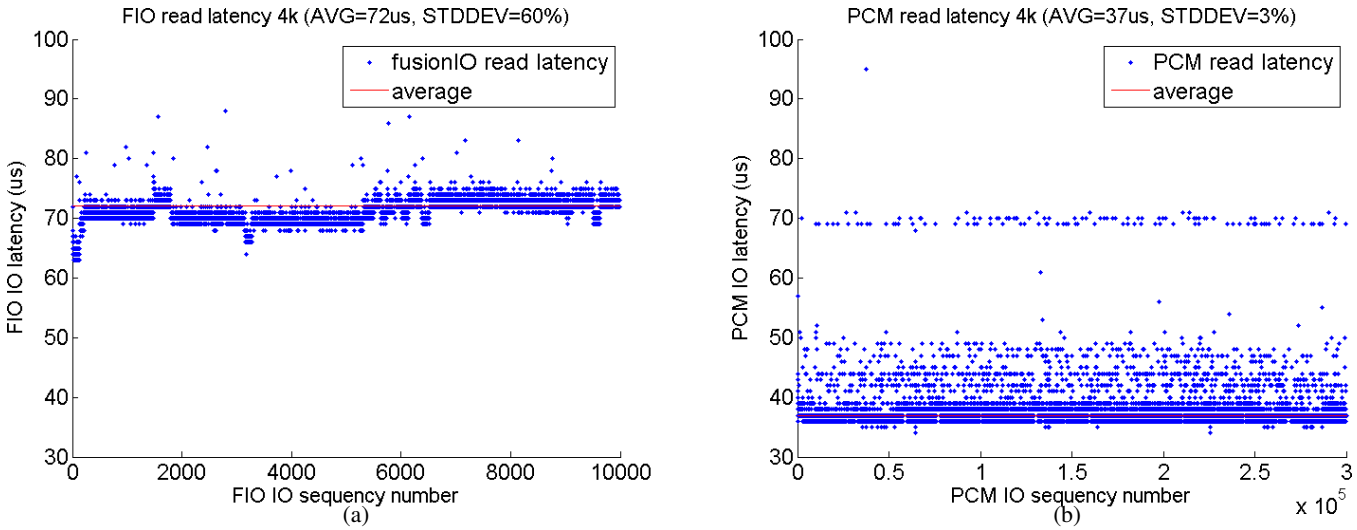

Figure 12: Read latency for (a) flash (zoomed in the most relevant part) and (b) PCM

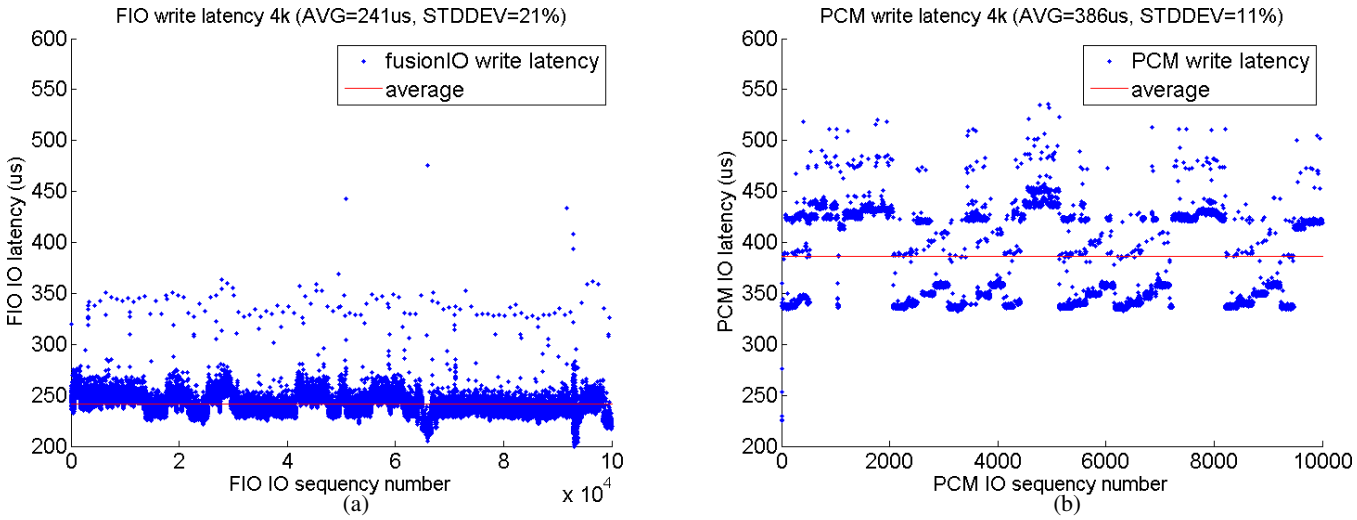

Figure 13: Write latency for (a) flash (zoomed in the most relevant part) and (b) PCM

Read latency. The first experiment measures the read latency per 4K I/O request using flash and PCM. We perform ten thousand random reads directly to the device. In the flash case, there are a few outliers with orders of magnitude higher latency, a behavior encountered in related literature as well [17, 20]. Figure 12(a) is in fact a zoomed in version of the overall data points (excluding outliers) in order to show the most interesting part of the graph; there is some variation between $65\mu s$ and $90\mu s$. The average read latency for 4K I/Os using flash is $72\mu s$. The standard deviation of the read latency in flash is 60% of the average read latency. The PCM device, however, behaves differently both qualitatively and quantitatively. Firstly, there are no outliers with orders of magnitude higher read latency. The values are distributed between $34\mu s$ and $95\mu s$ with the vast majority (99%) focused in the area 34–$40\mu s$ (Figure 12(b)), averaging $36\mu s$. Secondly, the standard deviation in terms of a percentage of the average is much smaller compared with flash, only 3%. The first two experiments show clearly that the very first PCM devices will already be competitive for latency-bound read-intensive workloads both because of the average performance and the increased stability and predictability of performance.

Write latency. For write intensive workloads the available PCM prototype performs worse than the popular flash representative device, but it is still more stable. In particular, the PCM device has average write latency $386\mu s$ with 11% standard deviation while the flash device has average write latency $241\mu s$ with 21% standard deviation, which is depicted in Figures 13(a) and (b). On the other hand, the basic PCM firmware can support only 40MB/s writes. This prototype limitation in write performance is attributed to the fact that it was designed with read performance in mind. Newer PCM chips, however, are expected to increase write speed significantly over time, consistent with the raw capability of the technology, bridging the gap in write performance at the device level. Note that the flash device uses a complex driver (the result of 3 or more years of optimizations and market feedback) while the PCM prototype uses a naive driver that provides basic functionality.

Our read and write latency numbers are consistent with numbers from Onyx [11], except that our write latency is higher because our device does not use early completion.