DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express

Dejan Vučinić,[1] Qingbo Wang,[1] Cyril Guyot,[1] Robert Mateescu,[1] Filip Blagojević,[1] Luiz Franca-Neto,[1] Damien Le Moal,[1] Trevor Bunker,[2] Jian Xu,[2] Steven Swanson[2] and Zvonimir Bandić[1]
[1]*HGST San Jose Research Center,* [2]*University of California, San Diego*

Abstract

Phase Change Memory (PCM) presents an architectural challenge: writing to it is slow enough to make attaching it to a CPU's main memory controller impractical, yet reading from it is so fast that using it in a peripheral storage device would leave much of its performance potential untapped at low command queue depths, throttled by the high latencies of the common peripheral buses and existing device protocols.

Here we explore the limits of communication latency with a PCM-based storage device over PCI Express. We devised a communication protocol, dubbed DC Express, where the device continuously polls read command queues in host memory without waiting for host-driven initiation, and completion signals are eliminated in favor of a novel completion detection procedure that marks receive buffers in host memory with incomplete tags and monitors their disappearance. By eliminating superfluous PCI Express packets and context switches in this manner we are able to exceed 700,000 IOPS on small random reads at queue depth 1.

1 Introduction

The development of NAND flash and the market adoption of flash-based storage peripherals has exposed the limitations of a prior generation of device interfaces (SATA, SAS), prompting the creation of NVM Express [1] (NVMe), a simplified protocol for Non-Volatile Memory (NVM) storage attached to PCI Express. In the course of researching the capabilities of several novel memory technologies vying to displace flash, we set out to build NVMe-compliant prototypes as technology demonstrators. We found, however, that the maximal performance permitted by NVMe throttles the potential of many emerging memory cell technologies.

Phase Change Memory, one of the most promising contenders, achieves non-volatility by re-melting a material with two distinguishable solid phases to store two or more different bit values. Discovered in 1968 [2], this effect is today widely used in DVD-RW media, and is now making inroads into lithographed memory devices thanks to its favorable device size and scaling properties [3], high endurance [4] and very fast readout.

The most dramatic advantage PCM has over NAND flash is that its readout latency is shorter by more than two orders of magnitude. While its write latency is about fifty times longer than reads at current lithographic limits, it is already comparable with NAND flash and is expected to improve further with advances in lithography [5]. This makes PCM a very attractive alternative in the settings where the workload is dominated by reads.

The main motivation for the work we present in this paper is the desire to build a block storage device that takes advantage of the fast readout of PCM to achieve the greatest number of input-output operations per second (IOPS) permitted by the low physical latency of the memory medium. While spectacular numbers [6] of IOPS are touted for flash-based devices, such performance is only possible at impractically high queue depths. The fact remains that most practical data center usage patterns revolve around low queue depths [7, 8], especially under completion latency bounds [9]. The most critical metric of device performance in many settings is the round-trip latency to the storage device as opposed to the total bandwidth achievable: the latter scales easily with device bus width and speed, unlike the former. Under this more stringent criterion, modern flash-based SSDs top out around 13 kIOPS for small random reads at queue depth 1, limited by over 70 μs of readout latency of the memory medium (our measurements).

Here we describe how, starting from NVMe as the state of the art, we proceeded to slim down the read-side protocol by eliminating unnecessary packet exchanges over PCI Express and by avoiding mode and context switching. In this manner we were able to reduce the average round-trip protocol latency to just over 1 μs, a tenfold improvement over our current implementation of NVMe-compliant interface protocol. The resulting protocol, DC Express, exceeds 700 kIOPS at queue depth 1 on a simple benchmark with 512 B reads from PCM across a 4-lane 5 GT/s PCI Express interface, with modest impact on the total power consumption of the system.

We believe one cannot go much faster without retooling the physical link to the storage device.

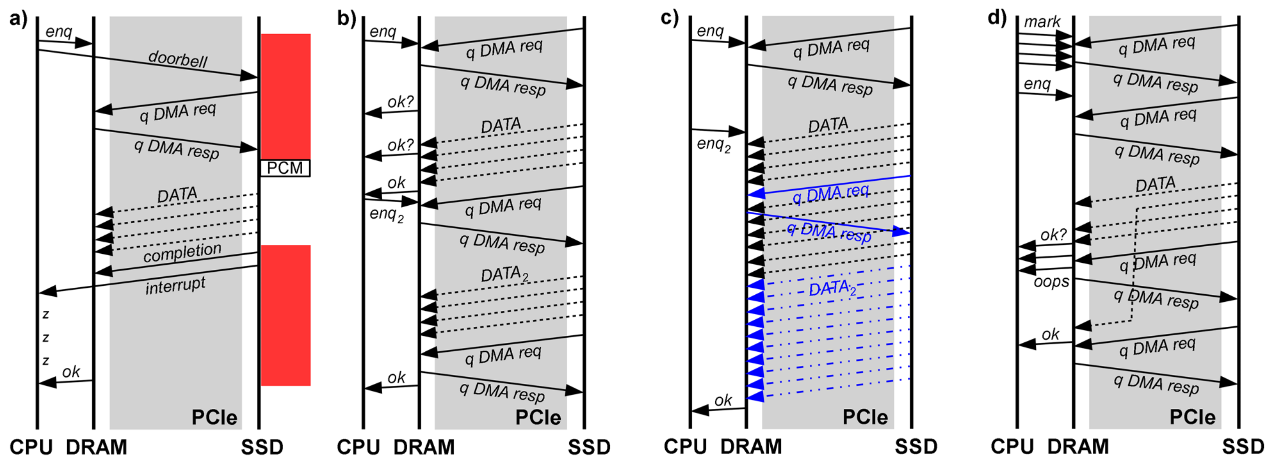


Figure 1: Timing diagrams illustrating the NVM Express and DC Express protocols. Time flows down, drawings are not to scale. a) NVMe: host CPU enqueues (***enq***) a command and rings doorbell; the device sends DMA request for the queue entry; the DMA response arrives; the command is parsed and data packets sent to the host DRAM, followed by completion queue entry and interrupt assertion; the host CPU thread handles interrupt. Red bars at right mark irreducible protocol latencies; rectangle illustrates the time when PCM is actually read. b) DC Express protocol at queue depth 1. There are no distinct doorbell nor completion signals. Device sends out DMA requests for new commands continuously. c) DC Express at higher queue depths. Subsequent DMA requests (***blue***) for new commands are launched prior to completing data transmission for the previous command (***black***) to take advantage of full-duplex nature of PCI Express and allow for seamless transmission. d) DC Express checks completion by marking each TLP-sized chunk of the receive buffer with an incomplete tag (***mark***) then monitoring for their disappearance. In case of out-of-order arrival the incomplete tag is found in one of the chunks (***oops)*** prompting a longer wait for all the data to settle.

2 Endpoint-initiated queue processing

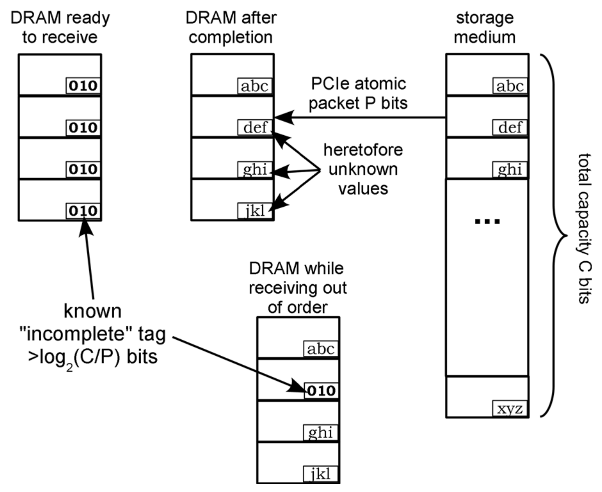
Prompted by the observation that the latency of one PCI Express packet exchange exceeds the time required to transfer a kilobyte of data, the approach we took to try to maximize the performance of a small read operation was to eliminate all unnecessary packet exchanges. At a minimum, even the leanest protocol requires a way to initiate and complete a transaction. In this section we describe endpoint-driven queue polling as an alternative to “doorbell” signals traditionally used for initiation; in the next section we discuss a minimalist way of signaling completion.

A rough outline of one NVMe-compliant read operation is shown in Figure 1a. The protocol for reading one block from the storage device begins with the host CPU preparing a read command in host DRAM and initiating the transaction by sending a “doorbell” packet over PCI Express. This is a signal to the device that there is a new read command waiting, hence it is to initiate a direct memory access (DMA) request—another PCI Express packet—to pick up that command from the queue in host DRAM.

Since every round trip over PCI Express incurs well over 0.6 μs latency on today’s fastest hardware, with such a protocol we waste a microsecond of back-and-forth over the bus before the device can even commence the actual reading of data from the non-volatile storage medium. In the past, with the fundamental read latency of NAND flash between 25 and 80 μs, this extra request latency was but a small fraction of total transaction time and so was deemed negligible. In contrast, the fundamental latency to first byte read from a modern PCM chip is 110 ns, so now the protocol becomes severely limiting when trying to maximize the overall performance of the storage device for small random reads at queue depth 1.

The most important regime we strive to optimize for is at high load. In this case there will almost always be a new command waiting in the queue should the device ask for one, making the sending of a doorbell signal for every small read superfluous. In the quest for best performance under these conditions latency becomes the key factor, and a given fraction of “no news” transfers we treat as an acceptable overhead.

With this scenario in mind, and taking advantage of the full-duplex nature of PCI Express, we present the first key ingredient of DC Express in Figures 1b and 1c. The device keeps sending out requests for one or more commands in the read queue in host DRAM without waiting for doorbell signals, so that there is almost always a request “in flight.” Further, since we can probe the actual round-trip latency for one DMA request to complete on given hardware, we can send anticipatory



DRAM ready to receive
DRAM after completion
storage medium
010
abc
def
ghi
jkl
PCIe atomic packet P bits
heretofore unknown values
...
xyz
total capacity C bits
DRAM while receiving out of order
known "incomplete" tag >log2(C/P) bits

Figure 2: Detecting completion by pre-populating locations of packet trailing bits in the receive buffer with incomplete tags. The disappearance of all incomplete tags is a robust signal that the entire data transfer has completed.

queue read requests prior to sending all data packets for a previous request so that the next commands, if available, arrive at the device just in time when the device is able to service another command (Figure 1c).

3 Tagging receive buffers as incomplete

To notify the host process that a read operation from NVM has completed, an NVMe-compliant PCI Express endpoint writes an entry into a "completion" queue in host DRAM with a DMA transaction, followed by the assertion of an interrupt signal to wake up the sleeping thread (*cf.* Figure 1a). This protocol has two adverse performance implications in addition to the bandwidth consumed by the completion signal itself.

First, PCI Express allows for out-of-order arrival of transaction-level packets (TLPs), meaning that the possibility exists for the completion packet to settle into DRAM prior to all its data having arrived—which would open a window for data corruption of random duration (*cf.* Figure 1d). To ensure that all the data packets have reached host DRAM prior to issuing the completion signal, the endpoint must declare "strict packet ordering" for that traffic class by setting a particular bit in the TLP header. Since PCI Express flow control works by prior exchange of "transaction credits," one subtle negative effect of strict ordering is that any delayed data packet and all its successors, including the corresponding completion packet, will be holding up the credits available until all the transactions complete in turn, which can slow down the rest of PCI Express traffic.

Second, the context switching [10] and mode switching overhead of interrupt-based completion signaling can easily exceed the latency of a small PCM read operation by two orders of magnitude. On a modern x86 processor running Linux, two context switches between processes on the same core take no less than 1.1 μs, so it is imprudent to relinquish the time slice if the read from the storage device is likely to complete in less time. Even if the interrupt signal is ignored by a polling host CPU, the act of asserting it entails transmitting a packet over the PCI Express link—which again results in a small penalty on the maximal payload bandwidth remaining.

To avoid these performance penalties, we reasoned that the lowest latency test of payload's arrival into DRAM would be to simply poll the content of the trailing bits in the receive buffer from a CPU thread: seeing them change would be the signal that the read operation has completed. This spin-wait would not necessarily increase CPU utilization since the cycles spent waiting for request completions would otherwise be spent on context switching and interrupt handling.

There are two obstacles to implementing this simple protocol. As already mentioned, the individual TLPs that comprise one read operation may arrive into DRAM out of order, so the last word does not guarantee the arrival of the entire buffer. And, the CPU does not know what final bits to look for until they have already been read from the device.

The solution, and the second key ingredient of DC Express, we elaborate in Figures 1d and 2. Since the granularity of TLPs on a given PCI Express link is known, in addition to checking the trailing bits of the entire receive buffer the protocol also checks the trailing bits of every TLP-sized chunk of host DRAM. In the event of out-of-order packet reception, such checking will reveal a chunk that has not yet settled, as shown in the bottom panel of Figure 2.

Instead of looking for particular bit patterns to arrive into the trailing bits of every atomic transfer, we choose an "incomplete tag," a pre-selected bit pattern that does **not** appear in the data that is about to arrive from the device. The protocol then writes this known tag to the receive buffer prior to initiating the read operation, and looks for its disappearance from the location of every packet's trailing bits as the robust completion signal. In this way we are using the fast link from CPU to DRAM to avoid sending any extraneous bits or packets over the much slower PCI Express link.

3.1 Strategy for choosing incomplete tag value

Obviously, the bit pattern used for the incomplete tag in our scheme must be different from the trailing bits of every TLP arriving. If we choose a pattern of length greater than $\log_2(C/P)$ bits (*cf.* Figure 2), where C is the total capacity of the storage device and P the size of one TLP, then in principle we can always select a pattern such that no TLP arriving from that particular storage device at that time will have trailing bits that match our choice of pattern. Note that this characteristic of storage device interfaces is different from, for instance, network interface protocols, where we are not privy to the content of arriving data even in principle.

One very simple strategy for choosing the pattern for the incomplete tag is to pick it at random. Although probabilistic, this method is adequate for the vast majority of computing applications that have no hard real time latency bounds.

Let's illustrate for the case of a device with 128 GiB of PCM and 128 B size of TLP payload. Dividing device capacity by the TLP size,[1] there are 2^{30} possible values at the trailing end of any one TLP-sized transfer. If we set the size for the incomplete tag at 32 bits, a randomly generated 32-bit pattern will then have at most $2^{30}/2^{32}$ = 25% chance of being repeated somewhere on the storage device—the worst case scenario where every one of the 2^{30} possible patterns is present on the device. If the random choice was unlucky and the generated pattern is present on the device, that read operation will get stuck since the arrival of that packet will go unnoticed, *i.e.* there will be a "collision."

The strategy, then, is to pick the length of the tag such that we can declare the probability of collision to be low enough. If it encounters a collision, the protocol simply times out the stuck read operation and chooses a new tag at random before retrying. The time to timeout we set to the product of maximum queue depth and maximum latency to complete one read operation.

For applications that do have hard real time latency bounds it is possible to devise more complex strategies such that the incomplete tag value is always chosen so no collision is possible. This would be done at the storage device at first power-up and whenever a write to the device invalidates the existing choice of pattern. One such strategy would be for the device to pick values at random and compare internally with the current contents of the device. This would incur no communication overhead as the storage medium accesses would be confined to the PCM controller on the device. If even that much latency cannot be tolerated, additional computing resources can be provided in the device to monitor writes to keep track of intervals of values not present in the currently stored data so that the selection of a new tag can always complete in constant time.

[1] We assume only block-aligned reads are allowed.

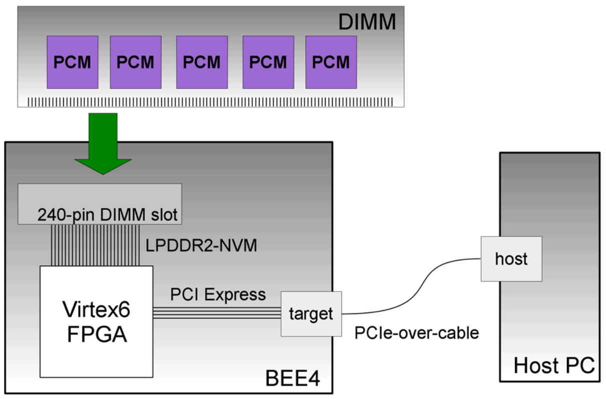


Figure 3: Diagram of our prototype system.

4 Performance

To implement DC Express we built a prototype NVM storage device (Figure 3) using a BEE4 FPGA platform (BEEcube, Inc., Fremont, CA) equipped with a custom-built DIMM card containing 5 Gib of Phase Change Memory (Micron NFR0A2B0D125C50). The NVM device exposed a 4-lane 5 GT/s ("gen2") PCI Express link from a Virtex6 FPGA running a custom memory controller that communicated with the PCM chips over the LPDDR2-NVM bus. The host systems used for testing included a Dell R720 server with an Intel Xeon E5-2690 CPU (Sandy Bridge-EP, TurboBoost to 3.8 GHz) and a Z77 Extreme4-M motherboard with an Intel i7-2600 CPU (Sandy Bridge, TurboBoost to 3.4 GHz). The NVM device was normally connected to the PCI Express lanes on the CPU dies. Alternatively, on the Z77 host we could use the lanes connecting to the Z77 chipset to measure the impact of the retransmission. All measurements were done on

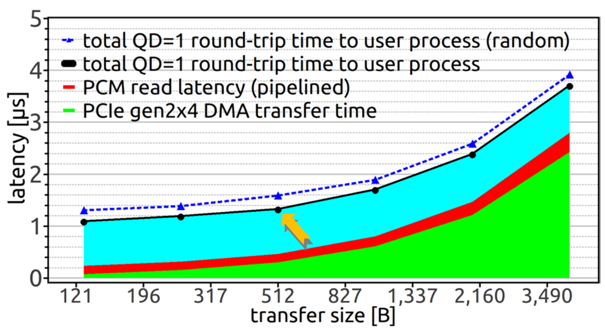


Figure 4: Average latency of a small random read operation when using the DC Express protocol at queue depth 1.

component	latency [µs]	kIOPS
data transfer (4 kiB)	2.432	
PCM read	0.368	
protocol + command parsing	0.863	273(1)
block driver	0.99	
read() call (kernel entry/exit)	1.17	
fio	0.506	158(2)

Table 5: Breakdown of contributions to average round-trip latency of DC Express for 4 kiB random reads at queue depth 1. IOPS were measured from a user space process (1) or linux block device driver (2). The total latency to a given layer is the sum of all latencies above it.

Linux kernel version 3.5 (Ubuntu and Fedora distributions).

We first exercised the bare protocol from a user space process by mmap()-ing a kernel buffer where the queues and receive buffer locations were pre-allocated. This allowed measurement of raw performance without mode or context switching overhead. The results are shown in Figure 4 for different transfer sizes. We designed the NVM device so that the bandwidth of data retrieval from PCM matches that of PCI Express transmission. Therefore, only the initial PCM row activation and local LPDDR2-NVM memory bus overhead (*red*) contribute to the irreducible protocol latency; the remainder is pipelined with PCI Express transfer (*green*). The remaining (*cyan*) component consists of PCI Express packet handling and command parsing, in addition to the polling from both ends of the link.

When we exercise the protocol in a tight loop, or with predictable timing in general, we can adjust the endpoint polling to anticipate the times of arrival of new commands into the read queue so that a new command gets picked up by the queue DMA request soon after its arrival into the queue. The total round-trip latency for this use case (shown by the solid black line in Figure 4) we measured as the inverse of the total number of read operations executed in a tight loop. For traditional 512 B blocks (arrow in Figure 4) the total latency seen by a user-space process averages 1.4 μs, over 700,000 IOPS.

If we fully randomize read command arrival times so that no predictive optimization of endpoint-driven queue polling is possible, there is additional latency incurred by the average delay between the arrival of a read command into the queue and the time when the next queue DMA hits. For this use case we measured the completion latencies using Intel CPU's time stamp counter (dashed blue line in Figure 4).

Next we constructed a lightweight block device driver to measure the impact of kernel entry and exit. We derived our driver from the Linux ramdisk device example. The read block size was limited to 4 kiB. We list the additional latencies in Table 5. One memory-to-memory copy of the retrieved block accounts for a small fraction of the time spent inside the block driver. Note that the tool used for measuring the latency of the block device, fio, contributes a significant amount of its own latency to these measurements. For comparison, our current implementation of NVMe-compliant device accessed through the Linux NVMe device driver under similar conditions reaches 78 kIOPS at queue depth 1, nearly 13 μs per 4 kiB read operation.

The latencies measured on the i7 system were comparable to those on the E5 server system when our device was connected to CPU lanes. Routing the packets through the Z77 chipset resulted in about two microseconds of additional latency per PCI Express round trip.

4.1 Power and congestion considerations

One concern with a protocol that continuously queries the host DRAM for new commands is the waste of resources at idle. To better understand the magnitude of this component relative to the baseline idle consumption of a modern server configuration, for this exercise we disabled all but one core on the single socket populated by the E5-2690 on the Dell R720 server equipped with 16 GiB of DDR3-1600 DRAM.

In Figure 6 we show the dependence of DC Express protocol performance and system power usage on the clock frequency of the CPU core doing the spin-wait. As expected, higher polling frequency reduces the average round-trip latency. Surprisingly, the optimal operating point, as defined by the Joules-per-IOP measure, is not at the lowest core frequency. Dominated by the

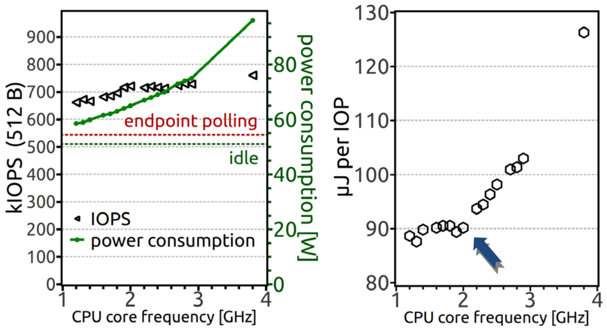


Figure 6: DC Express protocol performance for 512 B packets and total server power consumption as a function of the E5-2690 CPU core frequency.

significant idle power consumption of the entire server, the energy cost of one read operation stays relatively flat at low clock settings, suggesting a cost-optimal operating point near 2 GHz for this configuration (arrow in Figure 6) before hardware depreciation is taken into account.

Note that the overall impact of constant polling from the PCI Express endpoint is modest, about six percent of idle power consumption of the server. This is the worst case scenario where there is always a DMA request in flight, *i.e.* at queue depth 1 every other read of the command queue is guaranteed to be wasted. In this regime, fetching one 64 B command at a time would tie up less than six percent of the upstream PCI Express bandwidth.

5 Discussion

In this paper we described our attempts to wring the last drop of performance out of the widely adopted PCI Express interface, driven by the possibility of much higher performance frontiers uncovered by Phase Change Memory and other emerging non-volatile storage technologies. By eliminating unnecessary packet exchanges and avoiding context and mode switching we were able to surpass 700,000 IOPS at queue depth 1 when reading from a PCM storage device on commodity hardware. The performance increases further for smaller transfers to just under a million reads per second, the hard limit set by bus and protocol latency. By increasing the number of PCI Express lanes or the per-lane bandwidth it will be possible in the future to asymptotically approach this limit with larger transfers, but going even faster will require a fundamental change to the bus.

The unsolicited polling of DRAM from the endpoint to check for presence of new read commands results in a reduction in average protocol latency, but at the expense of slightly higher idle power consumption. We have shown that the worst-case impact is modest, both on power consumption and on the remaining PCI Express bandwidth. In settings with high load variability this component of overall power usage can be greatly mitigated ever further by, for instance, making the switch to DC Express at a given load threshold while reverting to the traditional “doorbell” mode of operation at times of low load.

Our focus was exclusively on small random reads, as that is the most interesting regime where PCM greatly outperforms the cheaper NAND flash. Write latency of the current generation of PCM is 55 times higher than read latency, so we did not attempt to modify the write-side protocol as the performance benefit would be small. For new memory technologies with much lower write latencies, *e.g.* STT-MRAM [11], a similar treatment of the write-side protocol could result in similarly large round-trip latency improvements, and will be the subject of future work.

Prior work on accessing low-latency NVMs over PCI Express has elaborated the advantages of polling over interrupts [12]. Our work goes two steps further: we introduce polling from both ends of the latency-limiting link, and we do away with the separate completion signal in favor of low-latency polling on all atomic components of a compound transfer.

While the advance in read performance we report is quite dramatic, it is important to note the high cost of using our protocol through kernel facilities. To maximize the read performance of PCM storage we resorted to a user-space library which did not provide security. To take advantage of the low latency while still enjoying safety guarantees from the operating system one must implement an additional protocol layer of negotiation through the kernel, such as *Moneta Direct* [13].

Our work casts PCM-based peripheral storage in a new light. Rather than using it in the traditional fashion, just like spinning disk of yore, we envision a new storage tier that fills a niche between DRAM and NAND flash. Using our FAST protocol will enable exposing very large non-volatile memory spaces that can still be read in-context with intermediate read latencies but without the several Watts per gigabyte penalty of DRAM refresh. On the other hand, treating PCM as block storage alleviates the need to rethink the cache hierarchy of contemporary CPUs, which would be necessary to achieve reasonable write performance in architectures where PCM is the main and only memory.

Beyond our work, almost an order of magnitude of further improvement in small random read latency is possible in principle before we hit the limits of the underlying physics of phase change materials. At this time, such advances would require either the use of parallel main memory buses together with deep changes to the cache hierarchy, or the use of fundamentally different high speed serial buses, such as HMCC [14], with shorter minimal transaction latencies. The latter, while promising, is still in the future, and is geared toward devices soldered onto motherboards as opposed to field-replaceable peripheral cards. It therefore appears that the niche for low read latency PCI Express peripheral storage based on Phase Change Memory is likely to persist until the arrival of future generations of peripheral buses and CPUs.

References

[1] http://nvmexpress.org/wp-content/uploads/2013/05/NVM_Express_1_1.pdf

[2] Ovshinsky, Stanford R. "Reversible electrical switching phenomena in disordered structures." *Physical Review Letters* **20**: 1450–1453, 1968.

[3] Servalli, G. "A 45nm generation phase change memory technology." *Electron Devices Meeting (IEDM), 2009 IEEE International*. IEEE, 2009.

[4] Goux, L. *et al*., "Degradation of the Reset Switching During Endurance Testing of a Phase-Change Line Cell." *IEEE Transactions on Electron Devices* vol.56(2), pp.354–358, 2009.

[5] Loke, D., et al. "Breaking the speed limits of phase-change memory." *Science* 336.6088: 1566–1569, 2012.

[6] Fusion-io: http://www.fusionio.com/overviews/9m-iops-technology-showcase/

[7] Seltzer, M., Chen, P. and Ousterhout, J. "Disk scheduling revisited." *Proceedings of the Winter 1990 USENIX Technical Conference*. USENIX Association, 1990.

[8] Personal communications with customers.

[9] Stanovich, Mark J., Baker, Theodore P., and Wang, An-I A. "Throttling on-disk schedulers to meet soft-real-time requirements." *Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2008.

[10] Li, C., Ding, C., and Shen, K. "Quantifying the cost of context switch." *ACM Workshop on Experimental Computer Science*. ACM, 2007.

[11] Huai, Yiming, et al. "Observation of spin-transfer switching in deep submicron-sized and low-resistance magnetic tunnel junctions." *Applied Physics Letters* 84.16: 3118-3120, 2004.

[12] Yang, J., Minturn, D. B., and Hady, F. "When poll is better than interrupt." *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association, 2012.

[13] Caulfield, Adrian M., et al. "Providing safe, user space access to fast, solid state disks." *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012.

[14] http://www.hybridmemorycube.org/