When Poll is Better than Interrupt

Jisoo Yang Dave B. Minturn Frank Hady

{jisoo.yang | dave.b.minturn | frank.hady} (at) intel.com

Intel Corporation

Abstract

In a traditional block I/O path, the operating system completes virtually all I/Os asynchronously via interrupts. However, performing storage I/O with ultra-low latency devices using next-generation non-volatile memory, it can be shown that polling for the completion – hence wasting clock cycles during the I/O – delivers higher performance than traditional interrupt-driven I/O. This paper thus argues for the *synchronous completion* of block I/O first by presenting strong empirical evidence showing a stack latency advantage, second by delineating limits with the current interrupt-driven path, and third by proving that synchronous completion is indeed safe and correct. This paper further discusses challenges and opportunities introduced by synchronous I/O completion model for both operating system kernels and user applications.

1 Introduction

When an operating system kernel processes a block storage I/O request, the kernel usually submits and completes the I/O request asynchronously, releasing the CPU to perform other tasks while the hardware device completes the storage operation. In addition to the CPU cycles saved, the asynchrony provides opportunities to reorder and merge multiple I/O requests to better match the characteristics of the backing device and achieve higher performance. Indeed, this asynchronous I/O strategy has worked well for traditional rotating devices and even for NAND-based solid-state drives (SSDs).

Future SSD devices may well utilize high-performance next-generation non-volatile memory (NVM), calling for a re-examination of the traditional asynchronous completion model. The high performance of such devices both diminish the CPU cycles saved by asynchrony and reduce the I/O scheduling advantage.

This paper thus argues for the *synchronous I/O completion model* by which the kernel path handling an I/O request stays within the process context that initiated the I/O. Synchronous completion allows I/O requests to bypass the kernel's heavyweight asynchronous block I/O subsystem, reducing CPU clock cycles needed to process I/Os. However, a necessary condition is that the CPU has to spin-wait for the completion from the device, increasing the cycles used.

Using a prototype DRAM-based storage device to mimic the potential performance of a very fast next-generation SSD, we verified that the synchronous model completes an individual I/O faster and consumes less CPU clock cycles despite having to poll. The device is fast enough that the spinning time is smaller than the overhead of the asynchronous I/O completion model.

Interrupt-driven asynchronous completion introduces additional performance issues when used with very fast SSDs such as our prototype. Asynchronous completion may suffer from lower I/O rates even when scaled to many outstanding I/Os across many threads. We empirically confirmed this with Linux,* and examine the system overheads of interrupt handling, cache pollution, CPU power-state transitions associated with the asynchronous model.

We also demonstrate that the synchronous completion model is correct and simple with respect to maintaining I/O ordering when used with application interfaces such as non-blocking I/O and multithreading.

We suggest that current applications may further benefit from the synchronous model by avoiding the non-blocking storage I/O interface and by reassessing buffering strategies such as I/O prefetching. We conclude that with future SSDs built of next-generation NVM elements, introducing the synchronous completion model could reap significant performance benefits.

2 Background

The commercial success of SSDs coupled with reported advancements of NVM technology is significantly reducing the performance gap between mass-storage and memory [15]. Experimental storages device that complete an I/O within a few microseconds have been demonstrated [8]. One of the implications of this trend is that the once negligible cost of I/O stack time becomes more relevant [8,12]. Another important trend in operating with SSDs is that big, sequential, batched I/O requests need no longer be favored over small, random I/O requests [17].

In the traditional block I/O architecture, the operating system's *block I/O subsystem* performs the task of scheduling I/O requests and forwarding them to block device drivers. This subsystem processes *kernel I/O requests* specifying the starting disk sector, target memory address, and size of I/O transfer, and originating from a file system, page cache, or user application using direct I/O. The block I/O subsystem schedules kernel I/O requests by queueing them in a *kernel I/O queue* and placing the I/O-issuing thread in an I/O wait state. The queued requests are later forwarded to a low-level block device driver, which translates the requests into *device I/O commands* specific to the backing storage device.

Upon finishing an I/O command, a storage device is expected to raise a hardware interrupt to inform the device driver of the completion of a previously submitted command. The device driver's interrupt service routine then notifies the block I/O subsystem, which subsequently ends the kernel I/O request by releasing the target memory and un-blocking the thread waiting on the completion of the request. A storage device may handle multiple device commands concurrently using its own *device queue* [2,5,6], and may combine multiple completion interrupts, a technique called interrupt coalescing to reduce overhead.

As described the traditional block I/O subsystem uses asynchrony within the I/O path to save CPU cycles for other tasks while the storage device handles I/O commands. Also, using I/O schedulers, the kernel can reorder or combine multiple outstanding kernel I/O requests to better utilize the underlying storage media.

This description of the traditional block storage path captures what we will refer to as the *asynchronous I/O completion model*. In this model, the kernel submits a device I/O command in a context distinct from the context of the process that originated the I/O. The hardware interrupt generated by the device upon command completion is also handled, at first, by a separate kernel context. The original process is later awakened to resume its execution.

A block I/O subsystem typically provides a set of in-kernel interfaces for a device driver use. In Linux, a block device driver is expected to implement a 'request_fn' callback that the kernel calls while executing in an interrupt context [7,10]. Linux provides another callback point called 'make_request', which is intended to be used by pseudo block devices, such as a ramdisk. The latter callback differs from the former one in that the latter is positioned at highest point in the Linux's block I/O subsystem and called within the context of the process thread.

3 Synchronous I/O completion model

When we say a process completes an I/O synchronously, we mean the kernel's entire path handling an I/O request stays within the process context that initiated the I/O. A necessary condition for this *synchronous I/O completion* is that the CPU poll the device for completion. This polling must be realized by a spin loop, busy-waiting the CPU while waiting for the completion.

Compared to the traditional asynchronous model, synchronous completion can reduce CPU clock cycles needed for a kernel to process an I/O request. This reduction comes primarily from a shortened kernel path and from the removal of interrupt handling, but synchronous completion brings with it an extra clock cycles spent in polling. In this section, we make the case for the synchronous completion by quantifying these overheads. We then discuss problems with the asynchronous model and argue the correctness of synchronous model.

3.1 Prototype hardware and device driver

For our measurements, we used a DRAM-based prototype block storage device connected to the system with an early prototype of an NVM Express* [5] interface to serve as a model of a fast future SSD based on next-generation NVM. The device was directly attached to PCIe* Gen2 bus with eight lanes and with a device-based DMA engine handling data transfers. As described by the NVM Express specification the device communicates with the device driver via segments of main memory, through which the device receives commands and places completions. The device can instantiate multiple device queues and can be configured to generate hardware interrupts upon command completion.

I/O completion method	512B xfer	4KiB xfer
Interrupt, Gen2 bus, enters C-state	3.3 µs	4.6 µs
Interrupt, Gen2 bus	2.6 µs	4.1 µs
Polling, Gen2 bus	1.5 µs	2.9 µs
Interrupt, 8Gbps bus projection	2.0 µs	2.6 µs
Polling, 8Gbps bus projection	0.9 µs	1.5 µs

Table 1. Time to finish an I/O command, excluding software time, measured for our prototype device. The numbers measure random-read performance with device queue depth of 1.

Table 1 shows performance statistics for the prototype device. The 'C-state' refers to the latency when the CPU enters power-saving mode while the I/O is outstanding. The performance measured is limited by prototype throughput, not by anything fundamental, future SSDs may well feature higher throughputs. The improved performance projection assumes a higher throughput SSD on a saturated PCIe Gen3 bus (8Gbps).

We wrote a Linux device driver for the prototype hardware supporting both asynchronous and synchronous completion models. For the asynchronous model the driver implements Linux's 'request_fn' callback, thus taking the traditional path of using the stock kernel I/O queue. In this model, the driver uses a hardware interrupt. The driver executes within the interrupt context for both the I/O request submission and the completion. For the synchronous model, the driver implements Linux's 'make_request' callback, bypassing most of the Linux's block I/O infrastructure. In this model the driver polls for completion from device and hence executes within the context of the thread that issued the I/O.

For this study, we assume that hardware never triggers internal events that incur substantially longer latency than average. We expect that such events are rare and can be easily dealt with by having operating system fall back to traditional asynchronous model.

3.2 Experimental setup and methodology

We used 64bit Fedora* 13 running 2.6.33 kernel on an x86 dual-socket server with 12GiB of main memory. Each processor socket was populated with quad-core 2.93GHz Intel® Xeon® with 8MiB of shared L3 cache and 256KiB of per-core L2 cache. Intel® Hyper-Threading Technology was enabled totaling 16 architectural CPUs available to software. CPU frequency-scaling was disabled.

For measurements we used a combination of the CPU timestamp counter and reports from user-level programs. Upon events of interest in kernel, the device driver executed the 'rdtsc' instruction to read the CPU timestamp counter, whose values were later processed offline to produce kernel path latencies. For application IOPS (I/O Operations Per Second) and I/O system call completion latency, we used the numbers reported by 'fio' [1] I/O micro-benchmark running in user mode.

We bypassed the file system and the buffer cache to isolate the cost of the block I/O subsystem. Note that our objective is to measure the difference between the two completion models when exercising the back-end block I/O subsystem whose performance is not changed by the use of the file system or the buffer cache and would thus be additive to either completion model. The kernel was compiled with -O3 optimization and kernel preemption was enabled. The I/O scheduler was disabled for the asynchronous path by selecting 'noop' scheduler in order to make the asynchronous path as fast as possible.

3.3 Storage stack latency comparison

Our measurement answers following questions:

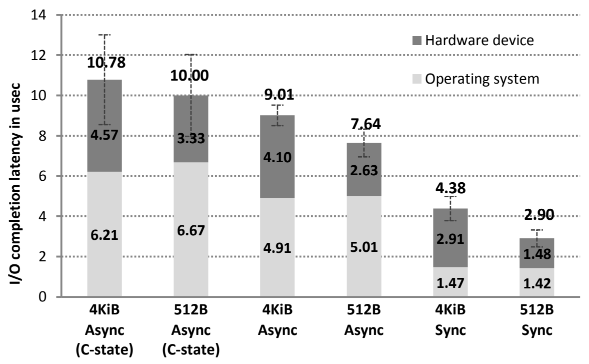

Figure 1. Storage stack block I/O subsystem cost comparison. Each bar measures application-observed I/O completion latency, which is broken into device hardware latency and non-overlapping operating system latency. Error bars represent +/- one standard deviation.

- How fast does each completion path complete application I/O requests?
- How much CPU time is spent by the kernel in each completion model?
- How much CPU time is available to another user process scheduled in during an asynchronous I/O?

Figure 1 shows that the synchronous model completes an I/O faster than asynchronous path in terms of absolute latency. The figure shows actual measured latency for the user application performing 4KiB and 512B random reads. For our fast prototype storage device the CPU spin-wait cost in the synchronous path is lower than the code-path reduction achieved by the synchronous path, completing a 4KiB I/O synchronously in 4.4µs versus 7.6µs for the asynchronous case. The figure breaks the latency into hardware time and non-hardware overlapping kernel time. The hardware time for the asynchronous path is slightly greater than that of the synchronous path due to interrupt delivery latency.

Figure 2 details the latency component breakdown of the asynchronous kernel path. In the figure, *Tu* indicates the CPU time actually available to another user process during the time slot vacated during asynchronous path I/O completion. To measure this time as accurately as possible, we implemented a separate user-level program scheduled to run on the same CPU as the I/O benchmark. This program continuously checked CPU timestamps to detect its scheduled period at a sub-microsecond granularity. Using this program, we measured *Tu* to be 2.7µs with 4KiB transfer that the device takes 4.1µs to finish.

The conclusion of the stack latency measurements is a strong one: the synchronous path completes I/Os faster and more efficiently uses the CPU. This is true despite spin-waiting for the duration of the I/O because the work the CPU performs in asynchronous path (i.e., *Ta* + *Tb* =

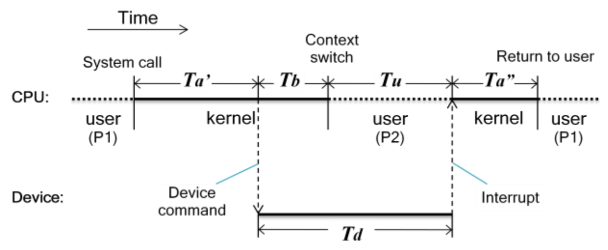

Figure 2. Latency component breakdown of asynchronous kernel path. *Ta* (= *Ta'* + *Ta''*) indicates the cost of kernel path that does not overlap with *Td*, which is the interval during which the device is active. Scheduling a user process P2 during the I/O interval incurs kernel scheduling cost, which is *Tb*. The CPU time available for P2 to make progress is *Tu*. For a 4KiB transfer, *Ta*, *Td*, *Tb*, and *Tu* measure 4.9, 4.1, 1.4 and 2.7µs, respectively.

6.3µs) is greater than the spin-waiting time of the synchronous path (4.38µs) with this fast prototype SSD. For smaller-sized transfers, synchronous completion by polling wins over asynchronous completion by an even greater margin.

With the synchronous completion model, improvement in hardware latency directly translates to improvement in software stack overhead. However, the same does not hold for the asynchronous model. For instance, using projected PCIe Gen3 bus performance, the spin-wait time is expected to be reduced from current 2.9µs to 1.5µs, making the synchronous path time be 3.0µs, while the asynchronous path overhead remains the same at 6.3µs. Of course the converse is also true, slow SSDs will be felt by the synchronous model, but not by the asynchronous model – clearly these results are most relevant for very low latency NVM.

This measurement study also sets a lower bound on the SSD latency for which the asynchronous completion model recovers absolutely no useful time for other processes: 1.4µs (*Tb* in Figure 2).

3.4 Further issues with interrupt-driven I/O

The increased stack efficiency gained with the synchronous model for low latency storage devices does not just result in lower latency, but also in higher IOPS. Figure 3 shows the IOPS scaling for increasing number of CPUs performing 512B randomly addressed reads. For this test, both the synchronous and asynchronous models use 100% of each included CPU. The synchronous model does so with just a single thread per CPU, while the asynchronous model required up to 8 threads per CPU to achieve maximum IOPS. In the asynchronous model, the total number of threads needed increases with number of processors to compensate for the larger per-I/O latency.

The synchronous model shows the best per-CPU I/O performance, scaling linearly with the increased number of CPUs up to 2 million IOPS – the hardware limitation

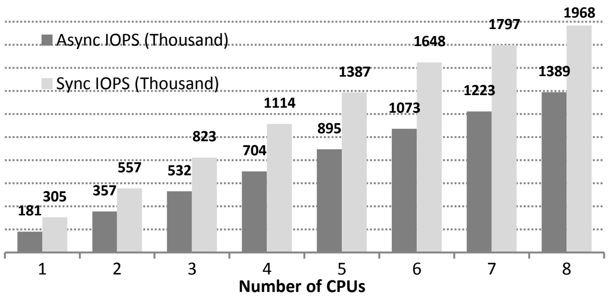

Figure 3. Scaling of storage I/Os per second (IOPS) with increased number of CPUs. For asynchronous IOPS, I/O threads are added until the utilization of each CPU reaches 100%.

of our prototype device. Even with its larger number of threads per CPU, the asynchronous model displays a significantly lower I/O rate, achieving only 60-70% of the synchronous model. This lower I/O rate is a result of inefficiencies inherent in the use of the asynchronous model when accessing such a low latency storage device. We discuss these inefficiencies in the following sections. It should be noted that this discussion is correct only for a very low latency storage device, like the one used here: traditional higher latency storage devices gain compelling efficiencies from the use the asynchronous model.

Interrupt overhead

The asynchronous model necessarily includes generation and service of an interrupt. This interrupt brings with it extra, otherwise unnecessary work increasing CPU utilization and therefore decreasing I/O rate on a fully loaded system. Another problem is that the kernel processes hardware interrupts at high priority. Our prototype device can deliver hundreds of thousands interrupts per second. Even if the asynchronous model driver completes multiple outstanding I/Os during a single hardware interrupt invocation, the device generates interrupts fast enough to saturate the system and cause user noticeable delays. Further while coalescing interrupts reduces CPU utilization overhead, it also increases completion latencies for individual I/Os.

Cache and TLB pollution

The short I/O-wait period in asynchronous model can cause a degenerative task schedule, polluting hardware cache and TLBs. This is because the default task scheduler eagerly finds any runnable thread to fill in the slot vacated by an I/O. With our prototype, the available time for a schedule in thread is only 2.7µs, which equals 8000 CPU clock cycles. If the thread scheduled is lower priority than the original thread, the original thread will likely be re-scheduled upon the completion of the I/O – lots of state swapping for little work done. Worse, thread data held in hardware resources such as memory cache and TLBs are replaced, only to be re-populated again when the original thread is scheduled back.

CPU power-state complications

Power management used in conjunction with the asynchronous model for the short I/O-wait of our device may not only reduce the power saving, but also increase I/O completion latency. A modern processor may enter a power-saving 'C-state' when not loaded or lightly loaded. Transition among C-states incurs latency. For the asynchronous model, the CPU enters into a power saving C-state when the scheduler fails to find a thread to run after sending an I/O command. The synchronous model does not automatically allow this transition to a lower C-state since the processor is busy.

We have measured a latency impact from C-state transition. When the processor enters into a C-state, the asynchronous path takes an additional 2µs in observed hardware latency with higher variability (Figure 1, labeled 'async C-state'). This additional latency is incurred only when the system has no other thread to schedule on the CPU. The end result is that a thread performing I/Os runs *slower* when it is the only thread active on the CPU – we confirmed this empirically.

It is hard for an asynchronous model driver to fine-tune C-state transitions. In asynchronous path, the C-state transition decision is primarily made by operating system's CPU scheduler or by the processor hardware itself. On the other hand, a device driver using synchronous completion can directly construct its spin-wait loop using instructions with power-state hints, such as mwait [3], better controlling C-state transitions.

3.5 Correctness of synchronous model

A block I/O subsystem is deemed correct when it preserves ordering requirements for I/O requests made by its frontend clients. Ultimately, we want to address the following problem:

> A client performs I/O calls '*A*' and '*B*' in order, and its ordering requirement is that *B* should get to the device after *A*. Does synchronous model respect this requirement?

For brevity, we assume that the client to be a user application using Linux I/O system calls. We also assume a file system and the page cache are bypassed. In fact, file system and page cache themselves can be considered as frontend clients using the block I/O subsystem.

We start with two assumptions:

A1. Application uses blocking I/O system calls.

A2. Application is single threaded.

Let us consider a single thread is submitting *A* and *B* in order. The operating system may preempt and schedule the thread on a different CPU, but it does not affect the ordering of I/O requests since there is only a single thread of execution. Therefore, it is guaranteed that *B* reaches to the device after *A*.

Let us relax A1. The application order requires the thread to submit *A* before *B* using non-blocking interface or AIO [4]. With the synchronous model, this means that the device has already completed the I/O for *A* at the moment that the application makes another non-blocking system calls for *B*. Therefore, the synchronous model guarantees that *B* reaches to the device after *A* with non-blocking I/O interface.

Relaxing A2, let us imagine two threads T1 and T2, each performing *A* and *B* respectively. In order to respect the application's ordering requirement, T2 must synchronize with T1 to avoid a race in such a way that T2 must wait for T1 before submitting *B*. The end result is that the kernel always sees *B* after kernel safely completes previously submitted *A*. Therefore, the synchronous model guarantees the ordering with multi-threaded applications.

The above exercise shows that an I/O barrier is unnecessary in the synchronous model to guarantee I/O ordering. This contrasts with asynchronous model where a program has to rely on an I/O barrier when it needs to force ordering. Hence, synchronous model has a potential to further simplify storage I/O routines with respect to guaranteeing data durability and consistency.

Our synchronous device driver written for Linux has been tested with multi-threaded applications using non-blocking system calls. For instance, the driver has withstood many hours of TPC-C* benchmark run. The driver has also been heavily utilized as a system swap space. We believe that the synchronous completion model is correct and fully compatible with existing applications.

4 Discussion

The asynchronous model may work better in processing I/O requests with large transfer sizes or handling hardware stalls that cause long latencies. Hence, a favorable solution would be a synchronous and asynchronous hybrid, where there are two kernel paths for a block device: the synchronous path is the fast path for small transfers and often used, whereas the asynchronous path is the slow fallback path for large transfers or hardware stalls.

We believe that existing applications have primarily assumed the asynchronous completion model and traditional slow storage devices. Although the synchronous completion model requires little change to existing software to run correctly, some changes to the operating system and to applications will allow for faster, more efficient system operation when storage is used synchronously. We did not attempt to re-write applications, but do suggest possible software changes.

Perhaps the most significant improvement that could be achieved for I/O intensive applications is to avoid using the non-blocking user I/O interface such as AIO calls when addressing a storage device synchronously. In this case, using the non-blocking interface adds overhead and complexity to the application without benefit because operating system already completes the I/O upon the return from a non-blocking I/O submission call. Although applications that use the non-blocking interface are functionally safe and correct with synchronous completion, the use of non-blocking interface negates the latency and scalability gains achievable in kernel with the synchronous completion model.

When the backing storage device is fast enough to complete an I/O synchronously, applications that have traditionally self-managed I/O buffers must reevaluate their buffering strategy. We observe that many I/O intensive applications existing today, such as databases, the operating system's page cache, and disk-swap algorithms, employ elaborate I/O buffering and prefetching schemes. Such custom I/O schemes may add overhead with little value for the synchronous completion model. Although our work in the synchronous model greatly simplifies I/O processing overhead in the kernel, application complexity may still become a bottleneck. For instance, I/O prefetching becomes far less effective and could even hurt performance. We have found the performance of page cache and disk-swapper to increase when we disabled page cache read-ahead and swap-in clustering.

Informing applications of the presence of synchronous completions is therefore necessary. For example, an ioctl() extension to query underlying completion model should help applications decide the best I/O strategy. Operating system processor usage statistics must account separately for the time spent at the driver's spin-wait loop. Currently there is no accepted method of accounting for this 'spinning I/O wait' cycles. In our prototype implementation, the time spent in the polling loop is simply accounted towards system time. This may mislead people to believe no I/O has been performed or to suspect kernel inefficiency due to increased system time.

5 Related work

Following the success of NAND-based storage, research interest has surged on the next-generation non-volatile memory (NVM) elements [11,14,16,19]. Although base materials differ, these memory elements commonly promise faster and simpler media access than NAND.

Because of the DRAM-like random accessibility of many next-generation NVM technologies, there is abundant research in storage-class memories (SCM), where NVM is directly exposed as a physical address space. For instance, file systems have been proposed on SCM-based architectures [9,21]. In contrast, we approach next-generation NVM in a more evolutionary way, preserving the current hardware and software storage interface, in keeping with the huge body of existing applications.

Moneta [8] is a recent effort to evaluate the design and impact of next-generation NVM-based SSDs. Moneta hardware is akin to our prototype device in spirit because it is a block device connected via PCIe bus. But implementation differences enabled our hardware to perform faster than Moneta. Moneta also examined spinning to cut the kernel cost, but its description is limited to latency aspect. In contrast, this paper studied issues relevant to the viability of synchronous completion, such as IOPS scalability, interrupt thrashing, power state, etc.

Interrupt-driven asynchronous completion has long been the only I/O model used by kernel to perform real storage I/Os. Storage interface standards have thus embraced hardware queueing techniques that further improve performance of asynchronous I/O operations [2,5,6]. However, these are mostly effective for the devices with slower storage medium such as hard disk or NAND flash.

It is a well-known strategy to choose a poll-based waiting primitive over an event-based one when the waiting time is short. A spinlock, for example, is preferred to a system mutex lock if the duration of the lock is held is short. Another example is the optional use of polling [18,20] for network message passing among nodes when implementing the MPI* library [13] used in high-performance computing clusters. In such systems communication latencies among nodes are just several microseconds due to the use of low-latency, high-bandwidth communication fabric along with a highly optimized network stack such as Remote Direct Memory Access (RDMA*).

6 Conclusion

This paper makes the case for the synchronous completion of storage I/Os. When performing storage I/O with ultra-low latency devices employing next-generation non-volatile memories, polling for completion performs better than the traditional interrupt-driven asynchronous I/O path. Our conclusion has a practical importance, pointing to the need for kernel researchers to consider optimizations to the traditional kernel block storage interface with next-generation SSDs, built of next-generation NVM elements in mind. It is our belief that non-dramatic changes can reap significant benefit.

Acknowledgements

We thank members of Storage Technology Group in Intel Corporation for supporting this work. We also thank our shepherd David Patterson and the anonymous reviewers for their detailed feedback and guidance. The views and conclusions in this paper are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Intel Corporation.

References

[1] Jen Axboe. Flexible I/O tester (fio). http://git.kernel.dk/?p=fio.git;a=summary. 2010.

[2] Amber Huffman and Joni Clark. Serial ATA native command queueing. Technical white paper, http://www.seagate.com/content/pdf/whitepaper/D2c_tech_paper_intc-stx_sata_ncq.pdf, July 2003.

[3] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1-3*. Intel, 2008.

[4] M. Tim Jones. Boost application performance using asynchronous I/O. http://www.ibm.com/developerworks/linux/library/l-async/, 2006.

[5] NVMHCI Work Group. NVM Express. http://www.nvmexpress.org/, 2011.

[6] SCSI Tagged Command Queueing, SCSI Architecture Model – 3, 2007.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, 3rd Ed., O'Reilly, 2005.

[8] Adrian M. Caufield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories, In *Proceedings of the 43rd International Symposium of Microarchitecture (MICRO)*, Atlanta, GA, December 2010.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 133–146, Big Sky, MT, October 2009.

[10] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*, 3rd Ed., O'Reilly, 2005.

[11] B. Dieny, R. Sousa, G. Prenat, and U. Ebels, Spin-dependent phenomena and their implementation in spintronic devices. In *International Symposium on VLSI Technology, Systems and Applications (VLSI-TSA)*, 2008.

[12] Annie Foong, Bryan Veal, and Frank Hady. Towards SSD-ready enterprise platforms. In *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, Singapore, September 2010.

[13] William Gropp, Ewing Lusk, Nathan Doss and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22:789-828, September 1996.

[14] S. Parkin. Racetrack memory: A storage class memory based on current controlled magnetic domain wall motion. In *Device Research Conference (DRC)*, pages 3-6, 2009.

[15] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using Phase-Change Memory technology. In *Proceedings of the 36th International Symposium of Computer Architecture (ISCA)*, Austin, TX, June 2009.

[16] 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, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52:465-480, 2008.

[17] Dongjun Shin. SSD. In *Linux Storage and Filesystem Workshop*, San Jose, CA, February 2008.

[18] David Sitsky and Kenichi Hayashi. An MPI library which uses polling, interrupts and remote copying for the Fujitsu AP1000+. In *Proceedings of the 2nd International Symposium on Parallel Architectures, Algorithms, and Networks (ISPAN)*, Beijing, China, June 1996.

[19] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80-83, May 2008.

[20] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 32-39, New York, NY, March 2006.

[21] Xiaojian Wu and Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*, Seattle, WA, November 2011.

* Other names and brands may be claimed as the property of others.