OS Support for High-Performance NVMs using Vector Interfaces

Vijay Vasudevan[*], David G. Andersen, Michael Kaminsky[1]
Carnegie Mellon University, [1]Intel Labs

Abstract

The move to many-core systems and the emergence of new non-volatile memories will require that we rethink and redesign systems for high-performance I/O. Individual core speeds are not increasing in sequential execution speed as quickly as non-volatile memories such as flash, PCM, and memristor technologies are improving speed. The formerly increasing CPU-I/O gap will therefore shrink considerably: what used to be I/O-bound will become CPU-bound with our existing I/O interfaces. Recent research by the authors and others has already observed that the system overhead for small I/O requests can cripple the performance of fast SSDs [2, 9]. These system overheads exist across the stack, including the filesystem, block device, device driver and interrupt handling layers.

Simultaneously, modern NVMs require a large number of outstanding requests to saturate device throughput because the devices are composed of a set of independently-addressable chips [5]. Some flash devices today already require maintaining at least ten in-flight requests, and future systems may require hundreds to cope with the increasing chip parallelism that helps improve device throughput and capacity. Providing tens to hundreds of in-flight requests from applications today requires either asynchronous I/O or extremely high thread counts using synchronous I/O, both of which can be difficult for applications to adapt to.

So we ask: How can we eliminate many of the system overheads observed in today's I/O stack, ensure that we can saturate the queue depth of modern highly-parallel NVM devices, all while providing a simple and understandable interface for developers?

In this work-in-progress, we argue that applications must be able to issue I/O requests to the operating system using a "vector interface" in order to give the OS the information it needs to satisfy these demands efficiently. Vector interfaces differ from today's existing interfaces because they must accept vectors of resources: For example, whereas today's `read()` interface accepts only a single file descriptor, buffer, and offset, we argue for having a `vec_read()` system call that takes in multiple file descriptors, buffers, and offsets as arguments. The OS can then issue this bundle of requests *efficiently* through the I/O stack for the following reasons:

First, vector interfaces can help eliminate redundant code execution in the I/O code path common across a batch of requests. For example, when opening several files in a single directory, the filesystem path resolution up until the parent need only be performed once, rather than once for every file. Second, these interfaces can provide opportunities for leveraging vector hardware (such as SSE and GPU hardware) when appropriate. Last, vector interfaces provide the basis for *automatic* interrupt mitigation through batched I/O completion, which in turn substantially reduces overhead in returning vectors of completed I/O events to applications.

This work-in-progress presents two major challenges for which we seek input from the community. The first involves understanding how *applications* should be exposed to these vector interfaces. Should they explicitly use raw vector interfaces, or should they be hidden behind libraries? Both approaches have their benefits and drawbacks: Raw vector interfaces provide ultimate control to the programmer to decide when and how to batch work together before submitting a call to the OS, whereas libraries hide the complexity of batching system calls but force applications to place more trust in the OS to decide how to schedule I/O. Prior work on batching system calls has taken multiple different forms: collective I/O, multicalls, asynchronous system calls, and compiler binary rewriting [3, 7, 10, 6]. These techniques differ significantly in how they expose vector interfaces to applications. Systems like FlexSC hide the asynchrony of its system calls behind backwards-compatible threading libraries, allowing unmodified applications to benefit from batched system call execution. The HPC community has proposed both "Multicollective I/O" [4] and HEC POSIX extensions (`readx/writex`) [8] as a way to batch I/O requests together and optimize access to shared storage, and its integration into MPI-IO demonstrates that HPC application writers are willing to embrace an explicit vector interface. We hope to explore the benefits and drawbacks

[*]Student author and presenter

of both the explicit and implicit interface approaches as part of our ongoing work.

Second, do we need hardware interface support at the storage level to take advantage of software vector interfaces? For example, requiring hardware to process vectors of commands could make interrupt scheduling more efficient: A set of related but discontiguous I/O requests sent to a storage device as a vector of commands can trigger an interrupt only when results for the entire set are available. In contrast, triggering one interrupt for each I/O increases interrupt load, and the OS must wait for the last interrupt from the set before delivering the results to the application. Vector storage interfaces also can provide opportunities for smarter I/O scheduling in NVMs [1], allowing a device to optimally re-order commands given a known batch of requests. Lastly, SSDs that require multiple in-flight commands will require algorithms and implementations that keep the device's queues occupied, and we believe networking and queuing theory ideas may apply well for these storage devices.

References

[1] S. Boboila and P. Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proc. USENIX Conference on File and Storage Technologies*, Feb. 2010.

[2] A. M. Caulfield, A. De, J. Coburn, T. Mollov, R. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO)*, Dec. 2010.

[3] K. Coloma, A. Ching, A. Chouhary, W. Liao, R. Ross, R. Thakur, and L. Ward. A new flexible MPI collective I/O implementation. In *Proceedings of the IEEE Conference on Cluster Computing*, Sept. 2006.

[4] G. Memik, M. T. Kandemir, W. Liao, and A. Choudhary. Multi-collective i/o: A technique for exploiting inter-file access patterns. volume 2, Aug. 2006.

[5] M. Polte, J. Simsa, and G. Gibson. Enabling enterprise solid state disks performance. In *Proc. Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, Mar. 2009.

[6] A. Purohit, C. P. Wright, J. Spadevcchia, and E. Zadok. Cosy: Develop in user-land, run in kernel-mode. In *Proc. HotOS IX*, May 2003.

[7] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlicting. Cassyopia: Compiler assisted system optimization. In *Proc. HotOS IX*, May 2003.

[8] R. Ross. HEC POSIX I/O API Extensions. `http://www.pdsi-scidac.org/docs/sc06/hec-posix-extensions-sc2006-workshop.pdf`.

[9] E. Seppanen, M. T. O'Keefe, and D. J. Lilja. High performance solid state storage under linux. In *26th IEEE Symposium on Massive Storage Systems and Technologies*, May 2010.

[10] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proc. 9th USENIX OSDI*, Oct. 2010.