In the discussion of choice of storage media for secondary storage in Exascale systems, a consensus seems to be that the hard disk will still be adopted as the major device to implement scratch, file, and archival systems [13]. Hard disk provides clear advantages in capacity, storage density, durability, and cost relative to extant alternatives. In comparison, flash and PCM memory cannot sustain a sufficient level of rewrites required in Exascale systems to take on the replacement role as the major storage device. While the challenge of very high bandwidth, such as for checkpointing, could be addressed by attaching more disks to each node, the performance loss from unaligned I/O is still a major issue for data-intensive applications to run efficiently on Exascale systems.

**Challenges Addressed:** When disk seeks are involved in the request service of a hard disk, service time can be significantly increased. To keep the system’s I/O throughout from being correspondingly reduced, we need to ensure that requests have a size of a megabyte or larger so that the seek time is amortized. However, there are two challenges that keep disk drives from receiving large requests. One is unaligned requests at the application level. In Exascale computing, the degree of concurrency will be very high, and if individual processes/threads do not issue large requests, the only chance to form large requests is to align requests from different processes/threads. Techniques such as Collective I/O to synchronize requests from hundreds of thousands of processes/threads will be infeasible at such a high degree of concurrency. New techniques in OS and Runtime (OS/R) must be developed to facilitate creating large requests with flexibility and low cost. The second challenge is unaligned requests at device level. Because file data are usually striped over a number of disks on one node or on a number of data nodes, a large request formed at the application level may be split into a number of sub-requests, each served by one disk. Though in a general setting a large striping unit size can ensure large sub-requests, for requests unaligned with the file striping pattern, their first and last sub-requests can be much smaller than the unit size and will be served inefficiently. Such unaligned accesses can readily result from, for example, files with small headers in front of regular data. Programmers alone cannot solve this problem—indeed, may not even be aware of the problem—so the OS/R must be designed to address the issue.

**Proposed Solutions:** To address the challenges we propose three components in the OS/R to ensure that only large requests reach the hard disks and are served efficiently. The first two components are to form large requests through effective request scheduling, and the third one attempts to eliminate small requests issued to the hard disk with the use of flash memory.

In the first component we relax the tight coupling specified by synchronous requests between requested data and process execution to generate a larger pool of requests for scheduling and thus increase the probability of being able to align requests from the same or different processes/threads. To this end, we need to know requests for data to be consumed in the future. This can be achieved either by providing interfaces to programmers such that they may register future data needs and having the runtime pre-issue these future requests, or by using pre-execution to automatically generate future requests. When a program runs in its I/O phase compute time can be much smaller than I/O time. In this case conventional prefetching is not effective because future requests cannot be served ahead of synchronous requests. We propose to temporally block the normal processes/threads to keep them from issuing synchronous requests, and let the future requests be efficiently served in a well-aligned manner. The processes/threads will then be released to consume the data when the data are ready in the memory. This data-driven execution is enabled for programs only when they are detected to be in an I/O phase (roughly, when the I/O-compute ratio exceeds a given threshold).

The second component targets the I/O scheduler. The I/O scheduler is the primary mechanism in the OS for sorting and merging requests from processes/threads. These requests are received into the dispatch queue, which is located in DRAM memory in current OSes. A longer queue allows more requests to be
collected and scheduled, and accordingly more requests to be aligned. Our study indicates that increasing the queue size from 128 to 8192 (for example) can double or even triple I/O throughput even for sequential access [10]. But because simply increasing the queue size in memory can run the risk of losing dirty data for write requests and consume excessive memory space, we propose to place the extended dispatch queue in flash memory, which is usually of larger size and more energy efficient. As we expect that flash memory or SSD will be installed on individual nodes in an Exascale system, this proposal is a feasible, simple, and effective method for improving I/O performance.

The third component is concerned with unaligned access due to data striping. Small sub-requests are especially detrimental to system’s performance: a request is not completed until each of its sub-requests has been served, and the low efficiency of serving its small sub-requests can bring down the service efficiency of an entire request, even if large. We propose to use flash memory to serve small requests, as this does not require large flash space and the number of rewrites is small compared to the use of flash as the major storage device or as a burst buffer. To enable the servicing of small read requests on flash we can track the data access pattern and make the data layout aware of and adaptive to the pattern. Data transfer between flash and hard disk can be scheduled when the I/O system is not busy.

**Advantages of the Solutions:** These three components have been individually implemented on LANL’s Darwin cluster of 120 nodes, among them 116 are 48-core (12-core by 4 socket) 2GHz AMD Opteron 6168, and are the nodes on which our experiments were performed. The evaluation results for the first two components have been reported in publications [10, 11]. The performance improvements are very impressive—for some I/O-intensive applications the throughput improvements can be more than 5X.

Though these proposed solutions have only demonstrated their effectiveness on a smaller-scale system, the ideas should be readily adaptable to much larger systems. First, they do not require expensive global synchronization; each decision is made locally. Second, high concurrency can produce many outstanding requests and easily fill an extended request dispatch queue for exploiting spatial locality. Third, flash memory is used sparingly and only as a performance booster, exploiting its advantage in serving small random requests and without imposing excessive writes to it.

There have been many efforts on improving I/O efficiency for disk-based storage systems in a parallel execution environment, including collective I/O [1,9], data sieving [1], I/O Orchestration [9], data prefetching [6,7], caching in the system buffer, and the use of flash memory (or SSD) to form hybrid disks [4,5] or burst buffers [2,3]. However, these schemes are not designed for Exascale systems and will have respective limitations in larger systems. For example, collective I/O and I/O Orchestration need process coordination, which may involve hundreds and thousands of processes/threads in the Exascale systems. Conventional data prefetching does not provide a sufficient number of requests for re-alignment before synchronous requests are issued. Caching in the system buffer is not effective unless I/O data has strong temporal locality, which may not be true for application processing “big data.” The same problem exists for the proposal of using SSD as a burst buffer. If SSD is used with the hard disks to form hybrid disks, care must be taken to ensure that only performance-critical data are placed on the SSD and the SSD must be used intelligently to minimize writes to it. Our third component represents a promising effort in this direction.

**Implementation Efforts:** The effort expended on prototype implementation on the Darwin cluster was moderate and mostly involved only isolated system components. The major OS/R components that would be instrumented or enhanced are (1) the MPICH2 library software to receive programmers’ prefetch hints; (2) the Linux device mapper for selectively directing requests to either SSD or the hard disk; (3) the PVFS2 functions to identify sub-requests for evaluating the effect of misalignment on I/O performance to decide the placement of data on SSD or hard disk. We expect that it would be relatively straightforward to port our existing implementation to a larger-scale system, and feasible to re-implement it for an Exascale system. The implementation does not require any significant changes to conventional OS/R structure or require comprehensive modifications of system modules. The existing implementation and its evaluation have demonstrated substantial performance advantages [10, 11, 12], so the risk for the proposed development is relatively modest.
References: