Throttling I/O Streams to Accelerate File-IO Performance

Seetharami Seelam
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
sseelam@us.ibm.com

Andre Kerstens and Patricia Teller
University of Texas at El Paso
Department of Computer Science
El Paso, TX 79968
{akerstens,pteller}@utep.edu

ABSTRACT

To increase the scale and performance of scientific applications, scientists commonly distribute computation over multiple processors. Often without realizing it, file I/O is parallelized with the computation. An implication of this I/O parallelization is that multiple compute tasks are likely to concurrently access the I/O nodes of an HPC system. When a large number of I/O streams concurrently access an I/O node, I/O performance tends to degrade. In turn, this impacts application execution time.

This paper presents experimental results that show that controlling the number of synchronous file-I/O streams that concurrently access an I/O node can enhance performance. We call this mechanism file-I/O stream throttling. The paper (1) describes this mechanism and demonstrates how it can be applied either at the application or system software layers, and (2) presents results of experiments driven by the cosmology application benchmark MADbench, executed on a variety of computing systems, that demonstrate the effectiveness of file-I/O stream throttling. The results imply that dynamic selection of the number of synchronous file-I/O streams that are allowed to access an I/O node can result in improved application performance. Note that the I/O pattern of MADbench resembles that of a large class of HPC applications.

Keywords

File I/O, high performance computing, I/O performance, I/O stream throttling, I/O scheduling

1. INTRODUCTION

Many parallel high-performance computing (HPC) applications process very large and complex data sets stored on multiple I/O nodes. Often the file I/O needed to store and retrieve this data is parallelized along with the computation. This generates multiple I/O streams (e.g., one for each MPI task) that concurrently access the system's I/O nodes. Because the number of I/O nodes is usually much smaller than the number of compute processors, even when only one such application is executing on a system, it is often the case that an I/O node is concurrently accessed by multiple streams. Even when I/O parallelization is not taken into consideration, due to the complex data sharing characteristics of scientific applications (such as when data produced by one task are consumed by all neighbors), an I/O node is likely to be concurrently accessed by multiple tasks. Thus, in the future, as problem sizes grow with the number of processors of next-generation systems and as the ratio of compute processors to I/O nodes increases, the number of I/O streams concurrently accessing an I/O node is destined to increase. Consider Table 1 [15], which gives the ratio of compute processors to I/O nodes for some high-end HPC systems. As the table shows, current ratios of compute processors to I/O nodes are greater than 40. Due primarily to the introduction of multicore processors, the ratios for next-generation systems are projected to be larger than this.

In some cases, e.g., when the I/O nodes are RAIDs, concurrency with respect to the number of streams accessing the nodes can result in improved I/O performance. But, as indicated in [22] and as confirmed by our experimental results presented in Section 5, I/O performance drops rather precipitously at numbers over a certain threshold and negatively impacts application execution times. This behavior is due to events that prevent full I/O utilization, e.g., seek and rotational latencies, failure to take advantage of prefetching, and/or insufficient I/O request frequency.

For HPC applications similar to MADCAP CMB, traditional parallelization of file I/O can result in this behavior [5], which will be exacerbated in next-generation systems. The challenge is, given a workload, schedule I/O so that as much I/O node bandwidth as possible is realized. Although this paper does not address this challenge optimally, it describes and gives evidence of a potential solution, i.e., careful dynamic selection of the number concurrently-active synchronous file-I/O streams. We call this control mechanism *file-I/O stream throttling*.

The paper describes this potential I/O performance problem, shows its impact on a benchmark application that represents the I/O behavior of a large application class, presents the file-I/O stream throttling mechanism, and demonstrates the effect of this mechanism on I/O performance and application execution time. The I/O behavior of MADCAP CMB is represented in MADbench, the benchmark application used in the experiments described in this paper.

Table 1: Ratio of compute processors to I/O nodes for high-end HPC systems.

System	Compute Nodes	I/O Nodes	Ratio
SNL Intel Paragon	1,840	32	58:1
ASCI Red	4,510	73	62:1
Cray Red Storm	10,368	256	41:1
BG/L	63,536	1,024	64:1

The potential effectiveness of file-I/O stream throttling, hereinafter called *I/O-stream throttling* or *stream throttling*, is demonstrated in Figure 3(a). The figure compares MADbench execution times when one of 16, four of 16, eight of 16, and 16 of 16 MPI tasks generate I/O streams that concurrently access an I/O node. The depicted experimental results show that careful selection of the number of concurrently-active MADbench file-I/O streams can improve application execution time: in this case, the optimal number of I/O streams is four of 16. In contrast, with one of 16 or 16 of 16, execution time increases by 18% and 40%, respectively.

I/O-stream throttling can be performed at the application or system software layers. The former is achieved by having the application code specify the total number of streams concurrently accessing an I/O node; this is the method that is incorporated in the MADbench code. Throttling via system software could be implemented by stream-aware I/O schedulers or file system policies. The strengths and weaknesses of each are described in the paper.

The remainder of this paper is organized as follows. Section 2 describes related work, Section 3 describes two different stream throttling methods, and Section 4 describes the benchmark program MADbench. A summary of our experimental systems is presented in Section 5, while the experimental results and their implications are presented in Sections 6, 6.2, 6.3, and 7. Concluding remarks are included in Section 8.

2. RELATED WORK

There are several studies that concentrate on optimizing the parallel file-I/O performance of HPC applications by providing libraries and parallel file systems, and exploiting advancements in storage technologies [8, 21, 7]. Other optimization techniques include exploiting access patterns to assist file system prefetching, data sieving, and caching [14], overlapping computation with I/O [13], and employing asynchronous prefetching [16]. The latter technique is particularly suitable for HPC applications. Due to the largely regular nature of HPC application data access patterns, it is rather straightforward to predict the data needed for subsequent computations. Given sufficient disk system bandwidth, prefetching may minimize the effect of I/O node performance on application performance. But prefetching too aggressively increases memory traffic and leads to other performance problems [1]. In addition, as mentioned above, when bandwidth is limited, the problem of multiple streams concurrently accessing an I/O node is likely. This can result in a loss of the benefits of prefetching.

Our contribution differs from these methods but is complementary to the data sieving, data prefetching, and caching approaches. The focus of our work is on runtime control of the number of file-I/O streams that concurrently access an I/O node. Such control can minimize expensive disk-head positioning delays and, thus, increase both I/O and application performance.

3. FILE-I/O STREAM THROTTLING

File-I/O stream throttling controls the number of synchronous I/O streams that concurrently access the I/O system. In general, I/O-stream throttling must consider various *application characteristics*, e.g., request characteristics (sequential or random) and *system characteristics*, e.g., the numbers of I/O nodes and compute nodes, the numbers of processors and tasks on a compute node, storage system configuration, compute node memory size, and application data size. The large number of characteristics, as well as the complexities associated with them, make dynamic I/O-stream throttling a challenging task, one that has not yet been accomplished.

However, many HPC applications (see, e.g., [5, 11, 23]), have sequential data layouts, where each requested data stream is sequentially accessed from the storage system, and tend to read in an iterative fashion. For such data layouts and read behavior, the number of streams that should access an I/O node concurrently is dependent only on the I/O node configuration; the complexities associated with the application characteristics need not be considered. In this case, given a single-disk I/O node, I/O performance can be enhanced by minimizing expensive disk-head positioning delays; our experiments demonstrate this. For example, consider an application with a computational loop that is iterated multiple times by four MPI tasks that access unique files stored in an interleaved fashion on a shared disk. Figure 1 depicts this data layout, where (a) shows a more coarse, regular interleaving of data, which is the product of one writer being active at a time and (b) shows a finer, more irregular interleaving, which is the product of four writers being concurrently active. During every iteration, each task reads a chunk of data from its own file. When all four streams concurrently access the disk, with no coordination, the disk head continuously seeks between files; this behavior is illustrated in Figure 2(b). In contrast, as shown in Figure 2(a), when the number of streams that concurrently access the disk is one, i.e., the four streams access the disk one after another, the disk head hardly seeks. In this particular example, one stream results in the best performance because the storage system is one simple disk, the bandwidth of which is well utilized by the one stream. In contrast, an advanced storage system with multiple disks needs multiple streams to utilize the available bandwidth. Therefore, one should not conclude from this example that using one stream is the best for all storage systems.

The next two subsections describe the details of I/O-stream throttling implemented at the application and system software layers, respectively. The experimental results of throttling MADbench's I/O streams at runtime via an application software layer technique and a system software layer technique are described in Section 6.

3.1 Application Layer Stream Throttling

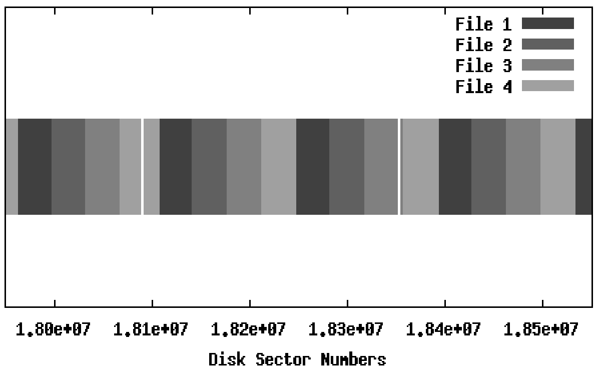
By similarly distributing computation and I/O operations across multiple processors, multiple I/O streams often access I/O nodes concurrently. One way to change (throttle) the number of streams concurrently accessing an I/O node is to wrap the parallelized I/O routines with code that implements a token-passing mechanism [5]. If one token is used by all MPI tasks then access to storage is sequentialized among the multiple I/O streams. At the other extreme, if the number of tokens is equal to the number of MPI tasks then all streams can concurrently access storage. This approach is used by some HPC applications, including MADbench.

The token-passing approach has two problems. First, it involves changes to the application code, which often is targeted to run on various compute and I/O node configurations. As shown in this paper, the selection of the number of concurrently active I/O streams depends on application I/O behavior and system characteristics. Thus, selecting the number requires extensive knowledge of both, as well as the ability to develop configuration-specific code that is tailored to the various systems. As demonstrated in [5], it is not impossible to throttle the number of I/O streams at runtime within an application. However, the use of higher-level I/O libraries such as MPI-I/O can make it very challenging to do so.

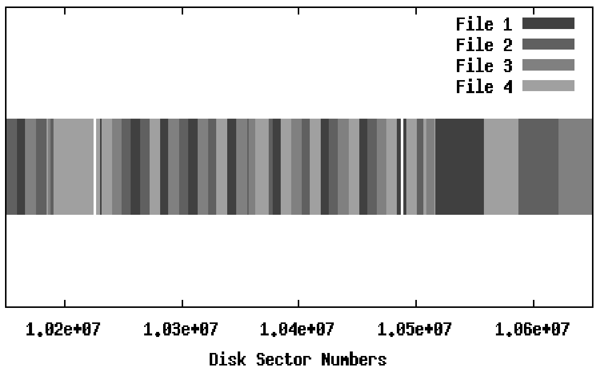
Second, I/O throttling is applicable only to synchronous I/O streams; it is not applicable to asynchronous I/O streams. A synchronous request blocks the execution of the requesting task until it is satisfied by the underlying I/O system. In contrast, asynchronous requests do not block task execution. In fact, asynchronous requests often are queued and delayed in the system's underlying memory buffer cache and scheduled to the relatively slower I/O systems at times that provide opportunities for improved I/O performance [19, 12, 17]. It is well known that larger asynchronous request queues provide better opportunities for optimizing disk head movement and for improving I/O system performance [19, 17]. Thus, throttling the number of asynchronous request streams is dangerous since it may constrain the number of streams that generate data to the memory buffer cache and, thus, restrict the number of outstanding asynchronous requests in the buffer cache queue.

Due to the two problems discussed above, I/O stream throttling at the application layer is not always the easiest and most scalable option. Other alternatives should be explored to improve I/O node performance. Possible alternate layers for throttling exist in the operating system (OS), file system, and/or middleware. In the next section, we discuss how I/O-stream throttling can be realized in the OS layer of an I/O node.

To properly understand the effect of stream throttling at the application layer, we must ensure that the number of synchronous I/O streams are not throttled at other layers between the appli-

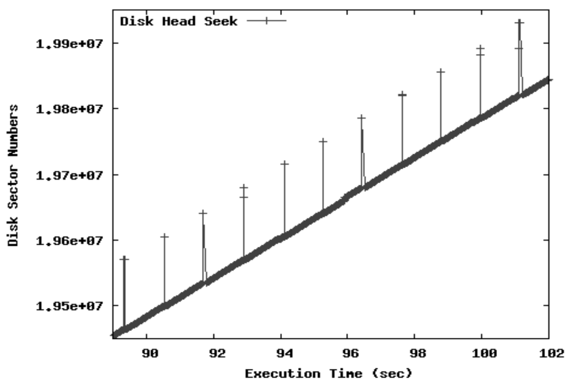


(a) One writer

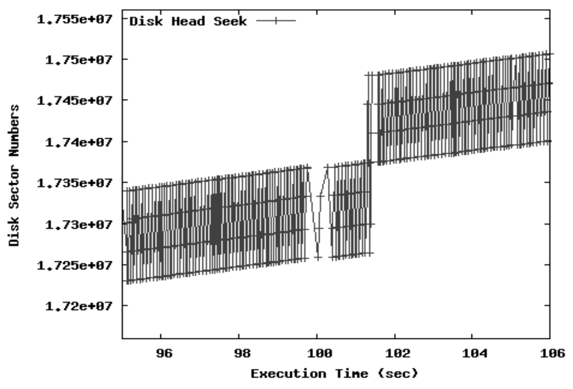


(b) Four concurrent writers

Figure 1: Disk data layout of four files, each accessed by one of four application tasks: (a) one active writer: more regular, coarser interleaving; (b) four concurrently-active writers: irregular, finer interleaving



(a) Sequential stream



(b) Concurrent streams

Figure 2: Seek behavior of the disk head when using one or multiple I/O streams.

cation and storage systems. Operating system I/O scheduling algorithms often tend to minimize the positional delays on the disk by servicing requests in a pseudo shortest-seek-first manner rather than a fair first-come-first-served manner. This is particularly true of our test OS, Linux. Fortunately, the latest versions of Linux allow users to select among four different I/O scheduling algorithms, one of which is a completely fair queuing algorithm (CFQ). We used CFQ for demonstrating our application throttling approach.

3.2 System Layer Stream Throttling

In Linux, the Anticipatory Scheduler (AS) [10, 18], unlike CFQ and other I/O schedulers, has the ability to detect sequentiality in an I/O stream and to allow a stream to continue to access the storage system for a certain period of time while it exhibits spatial locality of reference. This throttling reduces disk-head positioning delays even when multiple streams compete for the disk. In addition, the AS detects asynchronous and synchronous streams and throttles only the synchronous streams. Thus, we use AS to demonstrate the effect of system software throttling.

The main advantage of the system layer throttling approach, over the application layer approach, is that it is transparent to the application and application programmer. There are many ways to implement I/O-stream throttling at the system software layer, including the scheduler approach that we use in our experiments. Disk controllers on modern I/O systems have intelligence built into them. This means that I/O-stream throttling implemented at the controller could be more beneficial than if implemented at the I/O-node OS software layer. Also, due to adaptive prefetching policies, some disk controllers can handle multiple streams very efficiently. For instance, [22] shows that storage devices can handle as many as three streams efficiently. However, when the number of disk cache segments is less than the number of streams, performance degrades significantly. Unfortunately, the number of concurrently active I/O streams that the device can handle effectively is not always communicated to the OS software layer. Thus, in comparison to throttling at the device, throttling at the OS layer may lose some benefits. Also, since the AS is designed only to throttle down the number of streams to one and is not able to judiciously change the number of concurrently active I/O streams, this scheduler will not be optimal for all system configurations. Currently, a scheduler capable of judiciously selecting the number of concurrently-active I/O streams per I/O node is under investigation.

An experimental evaluation of our system software throttling approach is described in Section 6.3. Note that our experiments with system software throttling use the Anticipatory Scheduler, which incorporates a type of stream throttling, and are driven by the unthrottled, concurrently active file I/O streams of MADbench. In this case, every MPI task of MADbench concurrently generates I/O.

4. MADBENCH COSMOLOGY APPLICATION BENCHMARK

MADbench [5] is a light-weight version of the Microwave Anisotropy Dataset Computational Analysis Package (MADCAP) Cosmic Microwave Background (CMB) power spectrum estimation code [6]. It was developed using MPI and is one of a set benchmarks used by the National Energy Research Scientific Supercomputing Center (NERSC) at Lawrence Berkeley

National Laboratory (LBNL) to make procurement decisions regarding large-scale Department of Energy computing platforms. MADbench represents a large class of applications that deal with Cosmic Microwave Background Analysis. A significant amount of HPC resources are used by cosmologists, e.g., about 17% at NERSC. The I/O characteristics of MADbench are similar to those of many other applications and benchmarks, such as SMC (electron molecule collision), SeisPerf/Seis (seismic processing), CAM, FLASH [23], and codes based on the Hartree Fock algorithm like NWChem and MESSKIT [9, 11, 20]. This indicates that this study, which is based on MADbench, has relevance to a large class of applications and can be used to improve the I/O performance of applications with I/O behavior that is amenable to file-I/O stream throttling.

The MADbench code goes through three distinct I/O phases: (1) `dSdC`: Each MPI task calculates a set of dense, symmetric, positive semi-definite, signal correlation derivative matrices and writes these to unique files. (2) `invD`: Each MPI task reads its signal correlation matrices from the corresponding unique files and for each matrix it calculates the pixel-pixel data correlation (dense, symmetric, positive definite) matrix D and inverts it. Only reading and no writing takes place during this and the next phases. (3) `W`: Each MPI task reads its signal correlation matrices and for each matrix performs dense matrix-matrix multiplications with the results of `InvD`.

All three I/O phases, `dSdC`, `invD`, and `W`, are comprised of the following five steps: (1) computation, (2) synchronization (sync), (3) I/O, (4) synchronization, and (5) computation (except for `dSdC`). The time that each task spends in the the first sync step can be used to identify computational imbalance among the tasks, while the time that each task spends in the second sync step can be used to identify I/O imbalance among the tasks.

MADbench includes the following parameters that can be used to generate configurations that differ with respect to the stress they apply on the I/O system:

- `no_pix`: size of the pixel matrix in pixels. The size, S, in KBytes is $\frac{1}{8}(no\_pix)^2$. With P tasks, each task accesses a file of $\frac{S}{P}$ KBytes.
- `no_bin`: number of chunks of data in a file, where the chunk size, C, is $(S/P)/no\_bin$. The choice of `no_bin` impacts the layout and the interleaving of the files on the disk. The number of chunks of data in a file equals the number of iterations of the three phases.
- `no_gang`: number of processors for gang scheduling. In our case there is no gang scheduling, i.e., `no_gang` = 1.
- `blocksize`: ScaLAPACK block size.

In our experiments we use `no_bin` = 16 and `blocksize` = 32KB. These values are comparable to the values used in the original MADbench study [5]. The value for `no_pix` is based on the system's memory size and must be set large enough to stress the system's storage system. If `no_pix` is too small, all the data will fit in memory and the I/O system will not be stressed.

Because of the MADbench lock-step execution model, all tasks concurrently read or write their data from or to storage, creating multiple concurrently active I/O streams. Two additional input parameters, `RMOD` and `WMOD`, can be used to control the number of concurrent reader and concurrent writer streams, respectively. At most, one out of `RMOD` (`WMOD`) tasks are allowed to concurrently access the I/O system. With N tasks, the total number of concurrent readers is $\lceil \frac{N}{RMOD} \rceil$.

MADbench can be executed in two modes: computation and I/O, or I/O only. To execute it in the computation and I/O mode, the code must be linked with the scientific libraries ScaLAPACK, BLACS, and BLAS [3] and an MPI library like MPICH. In the second mode, I/O only, all the linear algebra computations done by the scientific libraries in phases `invD` and `W` are replaced by stubs containing sleep calls, where the sleep time is proportional to the flop count of the computation being replaced [4]. In this mode the code essentially does nothing but I/O in all phases.

5. EXPERIMENTAL SETUP

This section describes the systems, software, MADbench parameters, and tools used in our experiments. In all but one of our experiments MADbench is executed in I/O-only mode; the exception is used to indicate the impact of I/O performance on the execution time of MADbench and applications with similar I/O behavior.

5.1 Systems, System Setup, and System Tools

Table 2 describes the three systems used in our experiments (Intel Xeon, IBM p690, and Scyld Beowulf cluster) in terms of the number of processors, memory capacity per node, I/O system configuration, and I/O system throughput. The Intel Xeon system, which runs Linux 2.6, contains dual processors (2.80 GHz Pentium 4 Xeons with Hyper Threading) and is attached to a 7,200 RPM 10.2GB EIDE disk. The IBM p690 is a 16-way POWER4 SMP, which also runs Linux 2.6. Experiments on the p690 are of two types w.r.t. memory size and I/O system configuration. Memory size is either 16GB or 2GB, while the I/O system is either a 7,200 RPM 140GB SCSI disk or a 350GB DS4300 RAID-5 comprised of six 15,000 RPM SCSI disks. Each node of the Scyld Beowulf cluster contains two 2.0 GHz AMD Opteron processors and 4GB memory; it runs a custom Linux 2.4 kernel from Scyld. Experiments on this system also are of two types, but w.r.t. the I/O system configuration: either a 1.4TB NFS-mounted RAID-5 disk system comprised of six 7,200 RPM 250GB SATA disks or a local 7,200 RPM 120GB SATA scratch disk per node. To eliminate the interference of I/O requests from OS activities, on all the systems MADbench is configured to access a storage system different than that hosting the OS. In addition, to remove buffer cache effects, on all systems, before an experiment is started, the MADbench storage system is unmounted and remounted.

All but one set of experiments stress the storage systems described above: the data footprint does not fit in the buffer cache and, thus, data requests are satisfied by accesses to the storage system. The one exceptional case is described in Section 6.2.1. We use this case to provide a baseline for comparison. In this case, the memory buffer cache is large enough to hold the data footprint. Thus, after the first time data is requested (for writing), subsequent accesses (reads and writes) to the data are satisfied by the buffer cache rather than the I/O system. Accordingly, these experiments result in significantly lower application execution times.

The I/O request size may impact application execution time. For example, request coalescing or the use of larger requests improves application performance. Accordingly, our experiments use the largest possible I/O request size. For detailed analysis of seek behavior (as shown in Figure 2) and to understand locations of file blocks on the disk (as shown in Figure 1), we use `Blktrace` (see, e.g., [2, 17] for details), which has been part of the Linux kernel since 2.6.17-rc1.

6. RESULTS OF EXPERIMENTS

We conducted two different sets of experiments to describe the impact of application and system characteristics on I/O performance. The first is called the *base set of experiments*. It consists of four experiments driven by MADbench; they differ w.r.t. the number of readers and writers: (1) one reader/one writer, (2) one reader/all writers, (3) all readers/one writer, and (4) all readers/all writers, where "all" means the number of MPI tasks executing the program. The *base+ set of experiments* consists

Table 2: I/O and compute node configurations of systems.

System	Nodes	CPUs/Node	Memory/Node(GB)	I/O System(s)	Throughput (MB/s)
Intel Xeon	1	4	1	EIDE disk	23
IBM p690	1	16	16/2	SCSI disk/RAID-5	35/125
Beowulf Cluster	64	2	4	NFS with RAID-5	45

of the base set of experiments plus: (1) two readers/one writer, (2) four readers/one writer, and (3) eight readers/one writer. The base+ set is conducted on the experimental platforms with advanced storage systems, i.e., the p690 and the Beowulf cluster with RAIDs. Using all readers/all writers appears to be the default programming practice.

6.1 Importance of I/O Performance

To understand the relative importance of I/O performance on application execution time, we conducted the base set of experiments driven by MADbench in computation and I/O mode on the Intel Xeon system. (All other experiments are driven by MADbench in I/O-only mode.) For each experiment, Table 3 gives the total application execution time (COMP+I/O), the percentage of time spent performing I/O (%I/O time), and the time spent performing computation (COMP) and I/O (I/O) in each of the three phases of MADbench. On the Xeon system, I/O time, which depends on the number of concurrent readers and writers, ranges from 14% to 37% of the total execution time. Borrill, et al. [5] report that I/O consumes 80% of total execution time on larger systems with 256 or more processors. This indicates that for this class of applications I/O contributes significantly to total application execution time.

6.2 Application Layer Stream Throttling

The experiments discussed in this section use application throttling to control the number of concurrently-active MADbench readers (`RMOD`) and writers (`WMOD`). To isolate the effect of application throttling, these experiments use the CFQ I/O scheduler, which provides relatively fair disk access to all concurrently-active MADbench I/O streams and does not inherently perform any I/O stream throttling (as does the Anticipatory Scheduler).

The base set of experiments was conducted on four system configurations: the Xeon system, the p690 with a single disk and 16GB memory, the p690 with a single disk and 2GB memory, and the Beowulf cluster with a single disk per node. The base+ set was conducted on the p690 with a RAID and 2GB memory, and the Scyld Beowulf cluster with a RAID. Rows E1 through E3 in Table 4 show the input parameters used for the base set of experiments and rows E4 and E5 show the input parameters for the base+ set of experiments.

Table 4 also shows the best and second-best number of readers/writers for all experiments. Sections 6.2.1 through 6.2.5 provide details of each set of experiments. Except for the base set run on the p690 with 16GB memory, results of each experiment are presented in terms of total execution time and I/O times (LBSTIO and TIO). Since each MPI task waits at a barrier after each iteration of each MADbench phase, LBSTIO represents the average load balancing time, including synchronization at the barriers. TIO is the average I/O time of a task. Note that LBSTIO depends on the number of concurrently-active I/O streams and the underlying I/O scheduler. With a "fair" I/O scheduler, if all tasks are allowed to access the I/O system concurrently, all finish at approximately the same time, hence, the amount of time each waits at a barrier for other tasks to finish will be minimal. In contrast, as the number of concurrently active I/O streams decreases, tasks scheduled earlier finish earlier and experience longer delays at a barrier waiting for other tasks that are scheduled later. Because of this, we expect an increase in LBSTIO, indicating that some tasks finish their I/O earlier than others.

6.2.1 IBM p690: 16GB Memory

The results of the base set of experiments run on the IBM 690 with 16GB memory are shown in Table 5. The input parameters used are shown in row E1 of Table 4. As per the table, the total input data size is 2.98GB. Needless to say, with 16GB of main memory, the total amount of data (2.98GB) easily fits in memory. This experiment provides a baseline for comparison with experiments in Section 6.2.3, where memory is considerably smaller (2GB).

As shown in Table 5, in this case it is clear that only the initial writes of data are satisfied by the storage system, and all other accesses (reads and writes) are satisfied by the memory buffer cache. For example, in both cases where there are 16 readers, the `InvD` phase takes eight seconds to read 2.98GB of data. To accomplish this requires a bandwidth of over 380MB/s, while the associated SCSI disk only supports approx. 35MB/s. Following this logic, with the exception of the first time it is executed, one would expect the `dSdC` phase, which writes data, to have a similar execution time to that of the `InvD` and `W` phases, which both read data. However, in reality, the execution time of the `dSdC` phase is up to an order of magnitude greater. This is due to the File-IO sync call at the end of each write phase, which forces all the data to the disk before proceeding to the next phase. With 16 writers this takes about 80 seconds, which translates to 38 MB/s, which is approximately the throughput of the SCSI disk associated with the p690 system used to run the experiments.

With the explosive growth of the data needs of HPC applications, it is rare that the data ever fits in main memory. Thus, the remainder of our experiments ensure that the data size is larger than the combined size of the main memory of the participating compute nodes.

6.2.2 Intel Xeon System

The results of the base set of experiments run on the Intel Xeon system, which runs Linux 2.6, are shown in Table 6. The input parameters used are shown in row E2 of Table 4. The system contains dual processors (2.80 GHz Pentium 4 Xeons with Hyper Threading) and is attached to a 7,200 RPM 10.2GB EIDE disk. Accordingly, the number of logical processors and, thus, the number of MPI tasks, is four.

As shown in Table 6, the most common practice, i.e., have all tasks read or write (4/4), results in the longest application execution time. In contrast, employing one reader instead of four reduces execution time by as much as 22%. However, this decrease in execution time is accompanied by a nine-fold (more than two-fold) increase in the LBSTIO time in the `W` (`InvD`) phase. The implications of load balancing issues are further discussed in Section 7.

Given a fixed number of readers, the number of writers has a negligible impact on the total execution time, i.e., less than or equal to 5%. In contrast, given a fixed number of writers, one reader, versus "all readers", in this case four, improves the execution time by as much as 19.5%.

6.2.3 IBM p690: Single Disk/2GB Memory

The results of the base set of experiments run on the POWER4-based IBM p690 SMP with a 7,200 RPM 140GB SCSI disk and 2GB memory are shown in Table 7. For this system the number of processors and, thus, the number of MPI tasks, is 16. Similar to the results of the experiments run on the Intel Xeon

Table 3: **MADbench computation and I/O times on Intel Xeon system.**

Number of Writers	Number of Readers	Time(s)							% I/O Time
		Total	dSdC		InvD		W		
		COMP+I/O	COMP	I/O	COMP	I/O	COMP	I/O	
4	4	1102	32	23	38	137	740	132	27
4	1	978	31	6	28	99	731	83	19
1	4	1246	31	6	27	246	733	203	37
1	1	934	31	5	27	70	744	57	14

Table 4: Parameter values for experiments.

Exp. #	System/ Storage	# of tasks	`no_pix`	Data Size (GB)	Memory (GB)	Two Best Readers/Writers
E1	p690/single disk	16	5,000	2.98	16	16/16 and 1/16
E2	Xeon/single disk	4	3,000	1.1	1	1/1 and 1/4
E3	p690/single disk	16	5,000	2.98	2	1/1 and 1/16
E4	Cluster/RAID-5/NFS	16	25,000	74.5	64	8/1 and 4/1
E5	p690/RAID-5	16	10,000	11.9	2	2/1 and 4/1

Table 5: Application layer stream throttling: Total execution and I/O times for base set of experiments run on IBM p690 with 16GB memory.

Number of Writers	Number of Readers	Time (s)			
		Total	dSdC	InvD	W
16	16	79	70	8	1
1	16	105	96	8	1
16	1	83	74	5	4
1	1	106	97	5	4

Table 6: Application layer stream throttling: Total execution and I/O times for base set of experiments run on Intel Xeon system.

Number of Writers	Number of Readers	Time (s)						
		Total	dSdC		InvD		W	
			TIO	LBSTIO	TIO	LBSTIO	TIO	LBSTIO
4	4	243	12	9	83	3	77	1
1	4	235	12	5	83	2	77	1
4	1	199	12	12	53	13	45	10
1	1	189	18	9	48	10	42	10

system, given a fixed number of readers, the number of writers has a negligible impact on the total execution time, i.e., less than 3%. Again similar to the results associated with the Intel Xeon system, given a fixed number of writers, one reader, versus ”all readers”, improves the execution time. In this case execution time increases by as much as 14% with 16 readers; for the Xeon system it increases by as much as 19.5% with four readers. Again, accompanying this improvement in execution time is an increase in the I/O load balancing time, LBSTIO; it increases from less than ten seconds to as much as several tens of seconds.

6.2.4 *Beowulf Cluster: RAID-5*

The results of the base+ set of experiments run on the Scyld Beowulf cluster with a 1.4TB NFS-mounted RAID-5 disk system comprised of six 7,200 RPM 250GB SATA disks are shown in Table 8. Only 16 processors, i.e., one processor on each of 16 compute nodes and, thus, 16 MPI tasks were employed in the experiments. As shown in row E4 of Table 4, each of the 16 nodes has 4GB memory (making a total of 64GB), a larger problem size is used, i.e. $no\_pix = 25,000$, in order to force disk accesses.

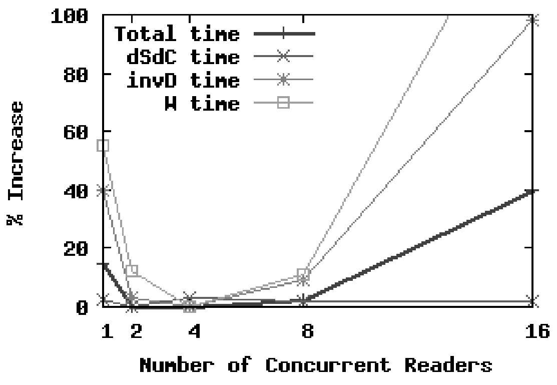
Since each node runs only one MPI task and the RAID-5 supports multiple concurrent streams, keeping the number readers the same, we expected that 16 (all) writers would result in a better execution time for the write phase (dSdC) compared to one writer. This was expected because one writer is not likely to fully utilize the bandwidth of the storage system. In addition, all writers results in larger I/O queues of asynchronous requests which are known to improve I/O performance [12, 17]. Referring to Table 8, as we hypothesized, ”all” writers does significantly reduce the `dSdC` write phase I/O time (TIO) by over 25% and result in exploiting full bandwidth of storage system. A total of 74.5 GB is written in 1900 seconds, which translates to 40MB/s, which is 90% of the peek bandwidth of the storage system (45 MB/s see, Table 2). Where as one writer takes about 3000 seconds for the same amount of data, which translates to 25.5 MB/s, which is less than 60% of the peek bandwidth. We should remark here that in addition reduced execution times, all writers also result in relatively smaller load imbalance times in the dSdC phase (see, LBSTIO for dSdC in Table 8). However, the reduction in execution time of dSdC phase using all writers does not translate into a smaller total execution time of the application. This is because using ”all” writers has a negative impact on the I/O times of the subsequent read phases: given a fixed number of readers, ”all” writers results in 60%, 69%, and 20% increases in the `InvD`, `W` TIO time, and total application execution time respectively. We speculate that this behavior is due to data placement on the disk systems. When multiple streams compete for storage, the file system tend to give out chunks in pseudo-round robin fashion to all streams and resulting files, although contiguous in memory, resides on non-contiguous blocks on the disk systems. Thus, with multiple writers the block allocation seems to be random (see, e.g., Figure 1(b)), which would significantly impact the subsequent read performance. Because the custom 2.4 Scyld

Table 7: Application layer stream throttling: Total execution and I/O times for base set of experiments run on IBM p690 with single disk and 2GB memory.

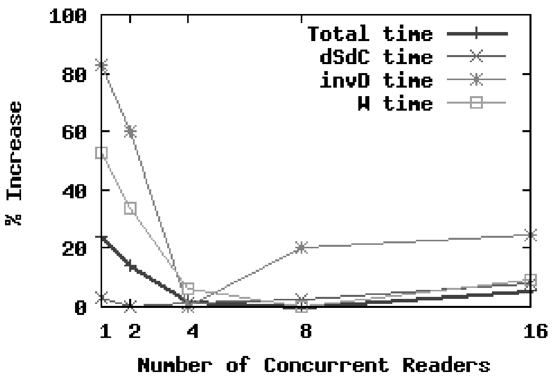
Number of Writers	Number of Readers	Time (s)						
		Total	dSdC		InvD		W	
			TIO	LBSTIO	TIO	LBSTIO	TIO	LBSTIO
16	16	355	56	3	157	6	144	5
1	16	358	33	34	152	7	144	7
16	1	318	59	10	60	48	52	37
1	1	311	31	37	59	46	51	43

Table 8: Application layer stream throttling: Total execution and I/O times for base+ set of experiments run on Beowulf cluster with RAID-5.

Number of Writers	Number of Readers	Time (s)						
		Total	dSdC		InvD		W	
			TIO	LBSTIO	TIO	LBSTIO	TIO	LBSTIO
16	16	6353	1330	66	2150	72	2210	68
16	1	8690	1320	67	1815	1596	1826	1606
1	1	6973	1559	1071	1148	793	1146	794
1	2	6406	1543	1000	1746	554	1128	568
1	4	5741	1583	1004	1073	273	1077	265
1	8	5627	1590	1035	1255	17	1249	17
1	16	6131	1781	1155	1374	12	1319	28



(a) IBM p690 with RAID-5



(b) Beowulf with RAID-5

Figure 3: Execution times of MADbench and the three individual phases with different concurrent readers on Beowulf cluster with RAID-5 and on p690 with RAID-5.

kernel that is running on this system does not have a tool like Blktrace, currently we cannot verify this claim. Because of the random placement with multiple writers, we use only a single writer for our subsequent experiments. We anticipate that a single writer will remove randomness in block allocation and results in consistent execution times.

Given one writer, we next identify the optimal number of readers (1, 2, 4, 8, or 16) for this system. For each number of readers, Figure 3(b) shows the percentage increase in both total execution time and execution times of three phases of MADbench. Table 8 shows the data. With 16 readers the total execution time is about 8% higher than with eight; with two readers, it is over 13% compared to eight, with four readers, it is less than 2% compared to eight, and with one reader, it is over 20% higher than with eight. Thus, the sweet spot is eight concurrently active readers. With one reader the disk system is underutilized (less than 85% peek bandwidth) as there are not enough I/O requests and with 16 readers, there are enough I/O requests but the disk system spends a significant amount of time in seeking to the different locations (position delays) to service the readers. In addition, as pointed out [22], 16 reader streams is likely to be more than the prefetch streams that the storage system can handle; modern storage arrays can prefetch 4 to 8 sequential streams [22]. With eight readers, however, the disk system is well utilized (close to 100% of the peek bandwidth), which only means that the positional delays are minimized and the storage system prefetching policies are effective.

In summary, one writer and eight readers case improves application execution time by as much as 11% over the default all writers all readers case (see Table 8), and by about 20% over the one writer, one reader case (see Table 8 and Figure 3).

6.2.5 IBM p690: RAID-5/2GB Memory

The last set of experiments described in the previous section, which explores the optimal number of readers given one writer, was also run on the IBM p690 with a high-end 350GB RAID-5 I/O system. The motivation for these experiments is to see if the observed behavior of throttling the number of readers over the default, indeed exist in other types of storage systems. Note that this storage system is from a different manufacturer, it uses more expensive hardware, and the disks are different; the spindle speed of these disks is 15000 RPM compared to the 7,200 RPM of the disks in the Beowulf cluster storage. This storage system gets a peek bandwidth of 125MB/s compared to the 45 MB/s of the cluster storage. However, both storage systems have same number of disks (six in this case), and both use same organization (RAID-5) so we expect similar results on this system.

The results are shown in Figure 3(a), which shows that four concurrently active readers is the sweet spot. In comparison, employing 16 readers results in an increase in execution time of about 40%; employing one results in an increase of over 13%, the difference is less than 2% for both 4 and 2 reader case. Reasoning and analysis similar to the one presented in the previous subsection can be used to explain the numbers and as expected the performance on this storage system is similar to the cluster storage system.

A couple of interesting observations can be made by comparing Figures 3(a) and 3(b). First, looking at the slope of the execution curve in both figures between 8 and 16 readers (although, we do not have the data for the intermediate points, but it is not required for this observation), one could easily conclude that the cluster storage handles high concurrency better than the IBM p690 system storage for this application. IBM storage's performance drop's rather dramatically compared to that of the cluster. Second, observe the slope of the execution time curve between two and four readers, the cluster storage seems more sensitive at a concurrency of two compared to the IBM storage. We must acknowledge that since the file system plays a major role, and since we use different file systems, this many not be attributed fully to the storage system alone.

6.3 System Layer Stream Throttling

As discussed in Section 3.2, our experiments with system software I/O stream throttling are implemented at the OS layer using the Linux 2.6 Anticipatory Scheduler (AS). This I/O scheduler does not throttle the number of concurrent writers; it throttles only the number of concurrent readers. In addition, it is designed to reduce the number of reader streams to one. We are working on a scheduler that allows throttling to an arbitrary number of streams.

Using the AS, which is available only under Linux, our system software throttling experiments reduce to experiments with all concurrent writers and one reader on the Intel Xeon and IBM p690 systems. For comparison purposes, we also present the no-throttling case, which is implemented using the CFQ I/O scheduler, and the best case from the application throttling experiments. We could not conduct the experiments on the Beowulf cluster because it runs a custom Linux 2.4 kernel from Scyld that does not provide multiple schedulers. Note that these experiments are driven by the unthrottled version of MADbench (`RMOD` and `WMOD` are both one).

The following sections describe the results of our system software throttling experiments. They indicate that the AS throttling behaves in a way that is consistent with the results of our application software throttling experiments. With few exceptions, the number of writers (in this case, fixed and in the case of application throttling, variable) has no significant impact on application execution time and one reader is a good choice.

6.3.1 Intel Xeon System

Table 9 presents the results of the system software throttling experiments conducted on the Intel Xeon system, for which the number of logical processors and the number of MADbench MPI tasks is four. The second row of the table presents the results of these experiments (System), while the first presents the results of the unthrottled experiments (UT), which use the CFQ scheduler, and the third row presents the results associated with the best-case application throttling experiment on this system (App). As the results indicate, although using the AS (System) decreases execution time by 12%, as compared to the unthrottled case (UT), application throttling performs best; its execution time is about 22% less than that of UT and 11% less than that of System. Thus, it appears that system throttling may provide a good alternative between no throttling and application throttling. It is worth remembering that a big advantage of system throttling is that it is transparent to the application programmer. In addition, as one can see from the LBSTIO times in Table 9, because the AS provides fine-grained throttling of streams, as compared to application throttling, it results in much lower I/O load imbalances in the read phases of MADbench. We further discuss the impact of this in Section 7.

6.3.2 IBM p690: RAID-5/2GB Memory

Table 10 presents the results of the system software throttling experiments conducted on our IBM p690, for which the number of processors and the number of MADbench MPI tasks is 16. Again, the second row of the table presents the results of these experiments (System), while the first and third rows present the results of the unthrottled experiments (UT), which use the CFQ scheduler, and the results associated with the best-case application throttling experiment on this system (App), respectively. Similar to the results of the experiments conducted on the Xeon system, although using the AS (System) decreases execution time by 5% (Xeon: 12%), as compared to the unthrottled case (UT), application throttling performs best; its execution time is about 12% (Xeon: 22%) less than that of UT and 8% (Xeon: 11%) less than that of System. This gives further credence to the indication that application-transparent, system throttling may provide a good alternative between no throttling and application throttling. As one can see from the LBSTIO times in Table 10, because the AS provides fine-grained throttling of streams, as compared to application throttling, it results in much lower I/O load imbalances (including the write phase of MADbench). In the results associated with the Xeon system, this is true only for the read phases.

7. IMPACT ON LOAD IMBALANCE

Figure 4 shows the execution times of a group of 16 tasks in the `InvD` phase of MADbench with (a) no throttling, (b) throttling at the application layer, and (c) and (d) two executions of throttling by system software. Referring to Figure 4(a), because of the fairness in the system with no throttling (provided by the CFQ scheduler), as expected, all tasks finish I/O at approximately the same time, i.e., at about 160 seconds, except for task 12[1]. As shown in Figure 4(b), (c), and (d), this does not happen when application or system software throttling is employed. These throttling strategies introduce unfairness in that they allow only a predetermined number of tasks to perform I/O at any given time. In other words, our resource distribution policy is unfair when we consider relatively small time intervals (e.g., during a one-second interval the I/O resource is not distributed fairly to all competing tasks). However, it is fair for relatively large time intervals (e.g., the time interval being the entire execution of a phase). The upshot of this throttling is an over 60% improvement in execution time (see Figure 4(a) and (b)); the downside is the performance variation across different executions, which is shown in Figure 4(c) and (d). Looking at the performance profiles of Figure 4(c) and (d), one can easily, but rather wrongly, conclude that there is an I/O load imbalance among the tasks. It is clear that the observed load imbalance is due to the adaptive stream throttling performed by the system software and not due to the data distribution among tasks. From our knowledge of the source code and as demonstrated by the no-throttling case (Figure 4(a)) each task works on the same amount of data.

Because of these performance variations or unpredictability in performance across executions introduced by stream throttling, it is difficult to conduct performance analysis for load balancing, to provide quality of service guarantees, and to forecast application performance. Therefore, it is not only important to understand the positive aspects of stream throttling, but it is equally

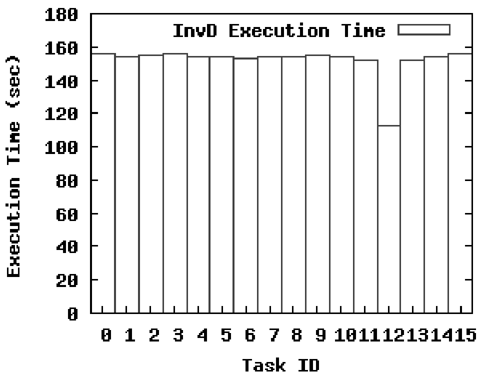
9 [1]The odd" behavior of task 12 is reproducible and we are investigating the issue.

Table 9: Total execution and I/O times of unthrottled case (UT), system software throttling (System), and best-case application throttling (App) on Intel Xeon system.

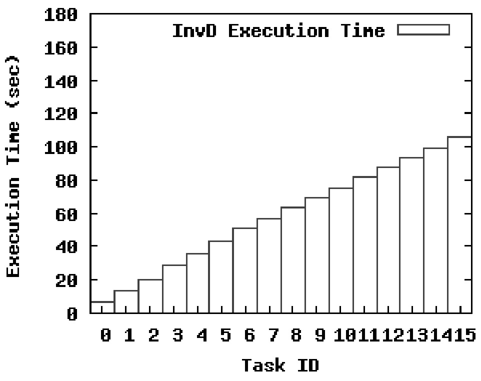
Method	Number	Number	Time (s)						
of	of	of	Total	dSdC		InvD		W	
Throttling	Writers	Readers	TIO+LBSTIO	TIO	LBSTIO	TIO	LBSTIO	TIO	LBSTIO
UT	4	4	243	12	9	83	3	77	1
System	1	1	213	67	12	72	5	63	5
App	1	1	189	18	9	48	10	42	10

Table 10: I/O and total execution times of unthrottled case (UT), system software throttling (System), and the best-case application throttling (App) on IBM p690.

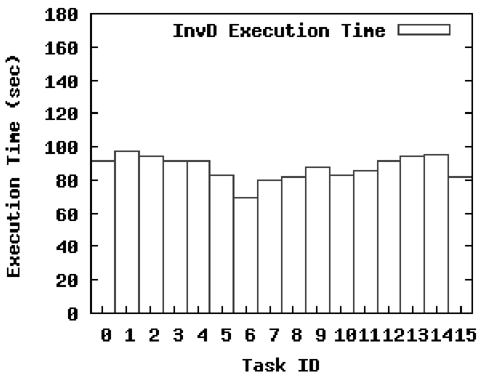
Method	Number	Number	Time (s)						
of	of	of	Total	dSdC		InvD		W	
Throttling	Writers	Readers	TIO+LBSTIO	TIO	LBSTIO	TIO	LBSTIO	TIO	LBSTIO
UT	16	16	355	56	3	157	6	144	5
System	1	1	339	25	18	97	30	82	20
App	1	1	311	31	37	59	46	51	43



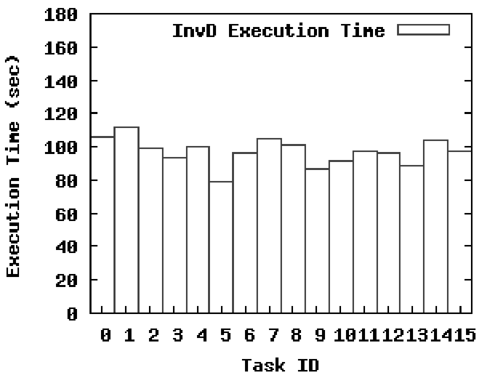
(a) No throttling



(b) Throttled by application



(c) Throttled by system software



(d) Throttled by system software

Figure 4: Impact of throttling on individual task execution times; (c) and (d) show the varying nature of task times across two executions.

important to understand the resulting complications and issues concomitant with such mechanisms.

8. SUMMARY AND CONCLUSIONS

The experimental results presented in this paper indicate that for a large class of applications execution time can be significantly enhanced by careful selection of the number of synchronous I/O streams that concurrently access the I/O nodes of HPC systems. For the various systems used to conduct our experiments, this I/O stream throttling, implemented within the application code, as compared to having all tasks read concurrently, which seems to be common practice, improves execution time by at least 8% and by at most 40%. With respect to the number of concurrently-active synchronous I/O streams, i.e., readers, on the single-disk systems, the best execution time was attained using one reader. In contrast, on the p690 with 16 processors and a RAID, four performed best; on the Beowulf cluster with 16 processors and a RAID, eight performed best. Clearly, our results show that the number of concurrently-active readers depends on the application I/O behavior and the characteristics of the system, in particular, I/O system characteristics.

Throttling of asynchronous I/O streams is a bit tricky because its effects are indirect. Thus, the paper focuses on the throttling of synchronous I/O streams. Nonetheless, we did conduct some experiments with application selection of the number of concurrently-active asynchronous I/O streams, i.e., writers. On systems with single disks, the results of experiments with a fixed number of readers show that the number of writers has a negligible effect on application execution time. However, on systems with RAIDs, although this throttling does not impact the time to execute the write phase, but it impacts the subsequent read performance and hence the application execution time.

I/O stream throttling can be implemented at the application layer or the system software layer. The former is not the easiest nor is it always the most scalable. It requires changes to the application code, which often is targeted to run on various compute and I/O node configurations. As shown in this paper, the selection of the number of concurrently-active I/O streams depends on application I/O behavior and system characteristics, and requires extensive knowledge of both. The major advantage of I/O stream throttling implemented at the system software layer is that it is transparent to applications. It can be implemented in the operating system (OS), file system, and/or middleware. The paper demonstrates it via the Anticipatory Scheduler in Linux, which constrains the throttling that can be accomplished, i.e., it only can reduce it to one.

In summary, the work presented in this paper indicates that a mechanism within system software is needed in order to address the potential I/O problem demonstrated in the paper, i.e., either too few or too many concurrently-active I/O streams access an I/O node, which causes the I/O node to be underutilized and, thus, degrades both I/O and application performance. This mechanism, given the characteristics of both the application and the system, must be capable of selecting the number of I/O streams that should be concurrently active in order to realize best performance of the I/O systems and best application execution time. We are working towards this goal via the development of a dynamically adaptive stream throttling system that will reside in the OS or file system of I/O nodes. Because of the load balancing issues discussed in the paper, we plan to provide controls to enable and disable this adaptation.

9. REFERENCES

[1] A. Aggarwal. Software caching vs. prefetching. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 157–162, New York, NY, USA, 2002. ACM Press.

[2] J. Axboe. Patch: Block device I/O tracing.

[3] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[4] J. Borrill. Lawrence berkely national laboratory (LBNL), personal communication.

[5] J. Borrill, J. Carter, L. Oliker, and D. Skinner. Integrated performance monitoring of a cosmology application on leading HEC platforms. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing (ICPP'05)*, pages 119–128, Washington, DC, USA, 2005. IEEE Computer Society.

[6] J. Carter, J. Borrill, and L. Oliker. Performance characteristics of a cosmology package on leading HPC architectures. In *HiPC*, pages 176–188, 2004.

[7] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[8] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel I/O performance optimization in panda. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, 1998.

[9] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, 1995. IEEE Computer Society Press.

[10] S. Iyer and P. Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, pages 117–130, October 2001.

[11] R. Kendall, E. Apr, D. Bernholdt, E. Bylaska, M. Dupuis, G. Fann, R. Harrison, J. Ju, J. Nichols, J. Nieplocha, T. Straatsma, T.L.Windus, and A.T.Wong. High performance computational chemistry: an overview of NWChem a distributed parallel application. In *Computer Phys. Comm.*, number 128, pages 260–283, 2000.

[12] C. Lever and P. Honeyman. Linux NFS client write performance. In *Proceedings of the Usenix Technical Conference, FREENIX track*, pages 29–40, June 2002.

[13] X. Ma, X. Jiao, M. Campbell, and M. Winslett. Flexible and efficient parallel I/O for large-scale multi-component simulations. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2003.

[14] T. M. Madhyastha and D. A. Reed. Exploiting global input/output access pattern classification. In *Proceedings of SC '97*, pages 1–18, November 1997.

[15] R. A. Oldfield, P. Widener, A. B. Maccabe, L. Ward, and T. Kordenbrock. Efficient data movement for lightweight I/O. In *Proceedings of the 2006 Workshop on high-performance I/O techniques and deployment of Very-Large Scale I/O Systems (HiPerI/O)*, 2006.

[16] P. Sanders. Asynchronous scheduling of redundant disk arrays. *IEEE Transactions on Computers*, 52(9):1170–1184, September 2003.

[17] S. Seelam, J. S. Babu, and P. Teller. Rate-controlled scheduling of expired writes for volatile caches. In *Proceedings of the Quantitative Evaluation of Sys-Tems (QEST'06)*, September 2006.

[18] S. Seelam, R. Romero, W. Buros, and P. Teller. Enhancements to linux I/O scheduling. In *Proceedings of the 2005 Linux Symposium*, July 2005.

[19] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the Winter 1990 USENIX Conference*, pages 313–324, January 1990.

[20] E. Smirni, C. L. Elford, A. J. Lavery, and A. A. Chien. Algorithmic influences on I/O access patterns and parallel file system performance. In *ICPADS '97: Proceedings of the 1997 International Conference on Parallel and Distributed Systems*, pages 794–801, Washington, DC, USA, 1997. IEEE Computer Society.

[21] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, February 1999.

[22] E. Varki, A. Merchant, J. Xu, and X. Qiu. Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):559–574, June 2004.

[23] G. Weirs, V. Dwarkadas, T. Plewa, C. Tomkins, and M. Marr-Lyon. Validating the Flash Code: Vortex-Dominated Flows. *apss*, 298:341–346, July 2005.