Enabling Event Tracing At Leadership-class Scale Through I/O Forwarding Middleware

Terry Jones*, Jason Cope†, Dries Kimpe†, Kamil Iskra†, Andreas Knüpfer‡,
Joseph Schuchart‡, Thomas Ilsche*, Stephen Poole*, Robert Ross†, Wolfgang E. Nagel‡

*Oak Ridge National Lab
Mailstop 5164
Oak Ridge, TN 37831
{trj,ilschett,spoole}
@ornl.gov

†Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439
{copej,dkimpe,iskra,rross}
@mcs.anl.gov

‡TU Dresden, ZIH
01062 Dresden Germany
{andreas.knuepfer,joseph.
schuchart,wolfgang.nagel}
@zih.tu-dresden.de

ABSTRACT

As concurrency increases in leadership-class computing systems, the large number of concurrent client data and metadata accesses can overload parallel file systems and reduce application I/O performance dramatically. One class of applications impacted by such issues are trace-based performance analysis and debugging tools. These tools strive to minimize any perturbation to the application being traced—a goal which is hard to accomplish given the unpredictable response times of parallel file systems. We present and evaluate the forwarding middleware IOFSL which is able to transparently aggregate and reorganize application I/O requests. The investigation resulted in novel ways to reduce the burden on the underlying file system in the cases where coordination at the application level is not feasible. As a demonstration case of our technique we present the integration into the Vampir tools for trace-based performance analysis. The paper contains results at large scale with a scalable target application (S3D) and the Vampir monitoring infrastructure on a leadership-class machine using more than 200,000 processes.

1. INTRODUCTION

The future of high performance computing is expected to rely upon extreme levels of concurrency. Today's leading parallel systems consist of hundreds of thousands of processing elements; machines consisting of a million or more processing elements are predicted in this decade [19]. As the level of concurrency reaches 100,000 clients and beyond, new stress points emerge in existing parallel file systems. The management of write locks becomes an issue—especially when all processes write to a shared file. In many parallel file systems, metadata accesses to the same directory are effectively serialized, so one million file create operations in the same directory become problematic. Moreover, such a large client-server discrepancy can result in many contention-related problems and brings about new issues with access coordination and concurrency management.

Existing parallel file systems provide unsatisfactory support for access patterns common in HPC applications. Using individual files per process/thread cannot be relied upon as a robust strategy at a scale of 10,000 clients or even less because of metadata contention. Such workloads may halt or even crash the whole file system. On the other hand, using one shared file causes excessive locking at the order of 100,000 clients or less. Concurrency not only affects basic I/O functionality; tremendous impacts to I/O performance, waiting time, and I/O load balance are also the norm. Additional complexities result from inhomogeneous I/O subsystems, e.g., specialized I/O nodes next to pure compute nodes.

Suitable solutions for extreme scale I/O are interesting technical challenges. They are very difficult to tackle at the application level because they require complex solutions which are usually outside of the application team's focus. Our approach relies on the novel I/O Forwarding and Scalability Layer (IOFSL) middleware to provide portable and scalable file I/O aggregation as well as metadata aggregation to bridge the gap between extreme scale applications and the constraints of parallel file systems.

1.1 Peta-Scale Event-Trace Recording

Performance analysis tools in general are a vital part of the HPC software ecosystem. They provide insight into the run-time behavior of parallel applications and enable performance assessments of what is adequate and what is insufficient. Furthermore, these tools guide performance optimization activities towards the most promising or most urgent aspects.

Performance analysis tools are vital to application development at very large scale. It is insufficient to move a target application to the highest degree of parallelism after the performance analysis on a lower scale, because this would ignore important emerging effects. One example is the very I/O behavior addressed in this paper which fundamentally changes for certain levels of parallelism.

While being essential for the performance optimization pro-

cess for target applications, the performance measurement tools face the same portability and scalability challenges as the target applications. Furthermore, the tools are needed most during the early phase of deployment for investigating new effects and for adapting applications for maximum performance.

As a demonstration case of our work, Vampir event-trace recording at full leadership-class scale was selected which is able to provide very detailed insight into peta-scale parallel execution at the cost of large amounts of measurement data produced. Without the presented integration of IOFSL this would be impossible due to the previously mentioned limitations.

1.2 Contributions

We present our recent research on scalable event management and request processing at the I/O forwarding layer named "I/O Forwarding Scalability Layer" (IOFSL). Our contributions include:

1. The design and demonstration of a scalable and portable event-driven architecture for the I/O forwarding layer.
2. Scalable and portable request processing, coordination, and aggregation techniques within the I/O forwarding layer.
3. The application to the portable Vampir performance analysis tools and demonstration on the leadership-class Cray XT5 installation Jaguar at ORNL.

The remainder of this paper is organized as follows: We describe the I/O requirements of performance analysis tools in general and the needs of the Vampir toolchain in Section 2. The main design components are outlined in Section 3. In Section 4, we describe the demonstration of our concepts on a leadership-class machine. Section 5 presents an overview of related work. Finally, Section 6 provides the conclusions and an outlook to future work.

2. THE VAMPIR TOOL-SET

The Vampir tool-set is a sophisticated performance analysis infrastructure for parallel programs using MPI, OpenMP, pthreads, CUDA, OpenCL, or combinations of them. It consists of the Vampir GUI for interactive post-mortem visualization, the VampirTrace instrumentation and run-time recording system, and the Open Trace Format (OTF) as the file format. The Vampir tools rely on event-trace recording which allows the most detailed analysis of the parallel behavior of target applications. In particular, it is more expressive than profiling techniques by design. As the first step, it performs automatic instrumentation of the target application using various techniques. During run time, the monitoring component collects the instrumented events together with significant properties. This includes for example enter/leave events for user code subroutines, message send/receive events, collective communication events, shared memory synchronization, I/O events, and many more. The event-trace data is written to a set of OTF files eventually and is then ready for post-mortem investigation with the Vampir GUI. Figure 1 gives an overview of VampirTrace's data flow. For more details about Vampir, VampirTrace, and OTF see [24, 18, 17].

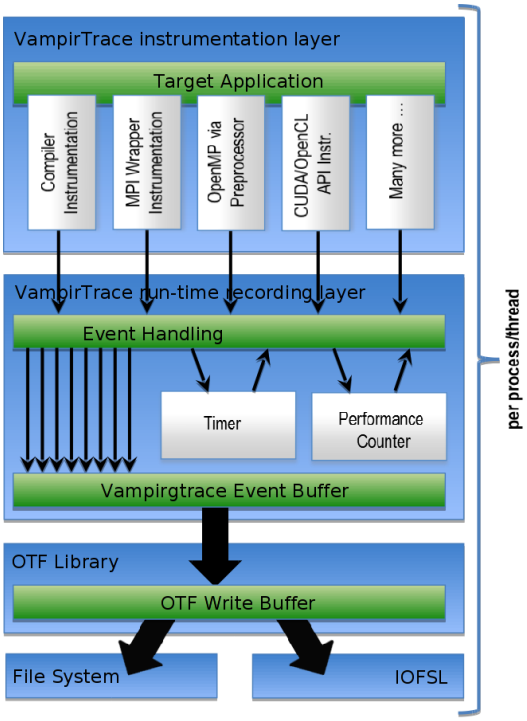

Figure 1: The VampirTrace data flow

When the VampirTrace monitoring component captures the parallel run-time behavior of the target application, it pays utmost attention to impose minimal perturbation. On occurrence of events of interest, the run-time monitoring component stores the event information together with vital properties (precise time stamp, all event specific properties, performance counter values if configured) to a pre-allocated memory buffer. The recording is performed independently for every process/thread into local buffers. Buffers are never shared to avoid artificial synchronization of the target application. This buffer sharing restriction applies to both processes and threads executing within the same address space.

The normal program execution is interrupted to write out all data to a file (through a buffer flush) when the buffer becomes full. The buffer flush phases are clearly marked for analysis if recording is continued afterwards, such that its effect is not mistaken for stray behavior of the target application. However, having a single flush at the end of the recording should be preferred whenever possible. In this scheme, every application process/thread produces a private output file. The buffer flushes across parallel processes/threads can either be asynchronous or synchronized using a high-watermark heuristic. Unfortunately, due to the SPMD nature of most parallel software, buffer flushes tend to happen almost simultaneously.

2.1 I/O Challenges

The I/O scheme of VampirTrace leads to occasional heavy I/O write phases induced by the event trace collection, which are independent of the I/O behavior of the target application. The data volume is directly proportional to the event frequency and the degree of parallelism. The former can be influenced by the configuration of the instrumentation and

the run-time monitoring system, the latter is fixed. A single event needs approximately 10 to 50 bytes for its encoding in the buffer. Typically, event frequencies range from 100 to 100,000 per second (with proper settings). A parallel run with 10,000 processes or threads for 10 minutes results in data sizes of $6{\cdot}10^9$ to $3{\cdot}10^{13}$ bytes (approx. 5.6 GB to 28 TB), as an example. The trace buffer size should not exceed the local main memory size minus the memory required by the target application, otherwise the application behaviour will be severely distorted. Typical sizes are 10 MB to 1 GB per process/thread.

During the buffer flush phases, the trace collection induces three challenges on the I/O subsystem. First, trace collection requires significant amounts of disk space for storing huge trace files. Second, the nearly simultaneous buffer flushes of many processes or threads increases I/O bandwidth pressure on the I/O subsystem. Finally, the metadata load for the trace collection I/O patterns is high due to the creation of many individual files and the allocation of file system blocks for a large number of I/O operations.

High end parallel file systems for machines with 50,000 to 300,000 CPU cores are usually well equipped to handle the first one, equipped to handle the second one, but not the third. Creating one file per process or thread, which usually corresponds to one file per CPU core or GPGPU card, at almost coordinated points in time is completely infeasible with all existing parallel file systems. Parallel file creation request above 4,000 per second[1] will affect all other users and jobs on the machine. This is a very dissatisfying situation, especially since (non-parallel) desktop file systems handle such demands gracefully. It is a serious limitation at a surprisingly low degree of parallelism for scaling up parallel codes whose I/O behavior is otherwise uncritical.

The current state-of-the-art solution is to avoid the creation of too many files altogether. For example, the demand for a full system run of the VampirTrace monitoring infrastructure (with any target application) on the Jaguar leadership-class machine at ORNL is to reliably create less than 1,000 files per second.

For any parallel application in general, this can only be provided by a modified parallelization scheme for the I/O parts. This necessarily requires internal synchronization and communication between the processes/threads, which may not always be practical for existing codes. For event trace generation however, this conflicts with the important goal not to superimpose additional synchronization on the target application. Adequate support from the underlying parallel file systems is not available and not foreseeable in the near future. The I/O forwarding approach presented next provides a convenient solution which integrates well with the existing VampirTrace/OTF infrastructure and which promises to scale much further than today's highest-end parallel systems.

3. DESIGN AND IMPLEMENTATION

[1] 4,000 per second is the suggested maximum file creation frequency for Jaguar's Spider file system at ORNL.

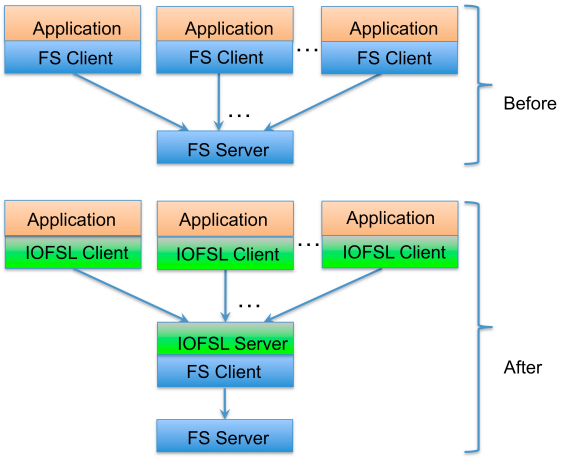

Figure 2: The principle I/O forwarding approach

In this section, we describe our I/O forwarding middleware used to address the previously outlined I/O challenges. In addition, we describe the implementation of our scalable trace collection tools and the integration of these tools in VampirTrace.

3.1 IOFSL Overview

The basis for our approach is to interject a new data management layer in the I/O stack that yields new capabilities as depicted in Figure 2. IOFSL [26, 3] provides a portable and scalable version of this I/O forwarding layer for HPC systems. Here, the I/O forwarding layer aggregates data from multiple clients at the I/O forwarding server (or multiple servers) and delegates the I/O requests on behalf of the clients. This I/O management approach is appropriate for many HPC I/O workloads that exhibit intense I/O demands at extreme scales, including general massively parallel applications using checkpoint-restart mechanisms as well as performance analysis and debugging tools. IOFSL can act as a bridge between application compute nodes and storage systems that are physically disconnected, as is the case on, e.g., IBM Blue Gene systems [31]. This bridge is also an ideal place to optimize file I/O traffic.

Besides simply rerouting I/O operations, data aggregation and coalescing capabilities are an important aspect of I/O forwarding. Parallel file systems yield much higher performance provided that read and write accesses are in full multiples of certain fundamental sizes (e.g., the file system block size in GPFS or the file stripe size in Lustre). For example, write accesses that are of insufficient size or alignment may incur costly performance penalties due to read-modify-write operations. Other benefits derived from the client reduction aspect of an I/O forwarding layer include important optimizations in file locking and other file system resources that scale with the number of clients. The I/O forwarding layer is also a convenient place to implement file caching or prefetching.

Without an I/O forwarding layer, data in a large machine may proceed from compute node memory to I/O storage node memory as depicted in Figure 3. The Figure demonstrates two separate data paths: (a) on the left, the data

from 12 clients proceed to the lower level file system as 12 separate accesses; (b) on the right, a collective I/O introduced at the I/O middle-ware layer performs a 4-to-1 aggregation and the lower level file system receives only 3 separate accesses. This type of optimization, which is available in the MPI-IO interface, can provide substantial benefit. However, incorporating aggregation in the I/O middleware depends on collective operations for best results, potentially requiring a reorganization of the I/O related portions of the application. This may not always be possible in situations where performance analysis and debugging tools are involved.

The inclusion of an I/O forwarding layer provides a helpful coalescing capability, as depicted in Figure 4. The I/O forwarding layer is able to perform aggregation and coalescing optimizations without any modifications inside the application. In the event that the application is already written with MPI-IO collective operations, no performance penalty is incurred. While MPI-IO is able to coalesce data accesses if collective calls are used, it does not have the ability to coalesce or aggregate metadata accesses, which is critical at extreme scale.

With the extra IOFSL layer, additional file system capabilities can be added that provide modified I/O semantics better suited for HPC. We will present one such capability, the atomic append, below.

3.2 IOFSL Clients and Servers

IOFSL consists of two primary components: an I/O forwarding client and an I/O forwarding server. These components and their interactions are illustrated in Fig. 5.

The IOFSL client integrates with the application I/O stack and is responsible for initiating I/O requests with the IOFSL server. There are several ways in which an application can invoke the IOFSL client. IOFSL provides an implementation of the stateless ZOIDFS API that applications can use to directly communicate with IOFSL servers. The ZOIDFS API provides many useful features for HPC applications, including portable file handles and list I/O operations. For applications that would like to use a traditional file I/O API, there are several tools that integrate the IOFSL client and translate I/O requests into IOFSL compatible I/O requests. For applications that use MPI-IO, a ZOIDFS driver for ROMIO is available. Applications using the POSIX file I/O API can use a FUSE client or a sysio [1] client. The ROMIO, FUSE, and sysio clients for IOFSL require no modifications of existing POSIX I/O or MPI-IO calls within applications.

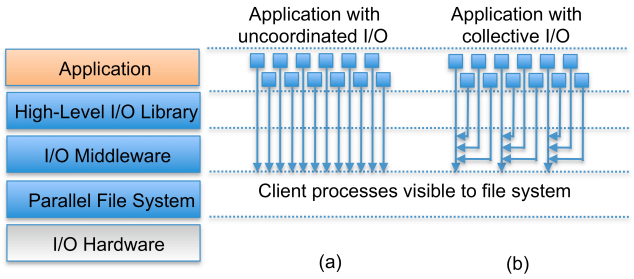

Figure 3: A typical I/O stack relies on collective I/O

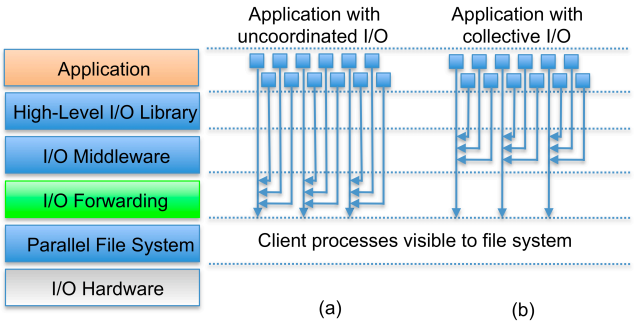

Figure 4: Aggregation provided by an I/O Forwarding layer

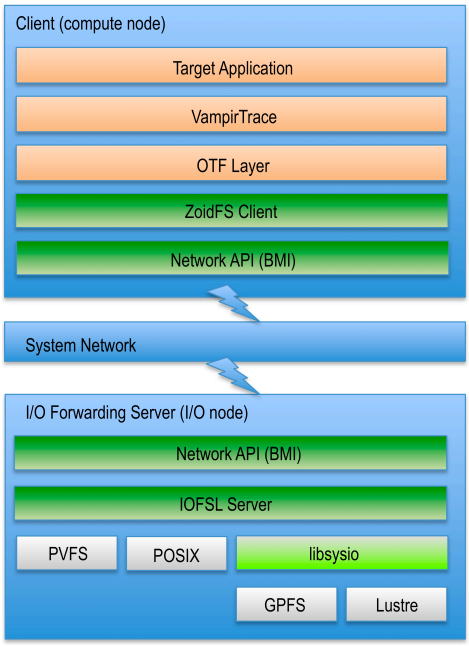

Figure 5: IOFSL Software Architecture

The role of the IOFSL server is to delegate I/O requests. Internally, the IOFSL server implements many optimizations to achieve scalable and efficient file I/O from many concurrent IOFSL clients. IOFSL implements an event-driven architecture that is built on top of asynchronous network, file, and computation resources. All client operations are translated into state machines that use these resources to execute application I/O operations.

IOFSL provides several drivers for interaction with common HPC file systems, for example to POSIX-based file systems using a native driver or a sysio-based driver, to PVFS2 file systems through a PVFS2 native driver, and to GridFTP servers [9]. When possible, these file system drivers take advantage of tunable parameters provided by the file systems, such as ioctls for tuning specific file or file system configurations. The IOFSL server is configured at initialization through a text-based configuration file which provides the server with information about the runtime environment and the IOFSL capabilities to enable.

For communication between the clients and servers, IOFSL uses the Buffered Message Interface (BMI) library [6]. BMI is a portable communication layer that was originally used within the PVFS2 file systems. It provides asynchronous and list I/O interfaces for network data transfers. Internally, BMI supports many common HPC networks using their native APIs. To date, BMI supports native access to the Portals networks used on the Cray XT platforms, the IBM Blue Gene/P tree network using ZOID, Myrinet Express (MX), InfiniBand, and TCP/IP. These network-specific drivers allow IOFSL to take advantage of the asynchronous, low-latency, and high-throughput characteristics of common HPC networks through an abstract and portable interface.

A recent addition to IOFSL provides Remote Procedure Call (RPC) capabilities for IOFSL clients and servers. This capability allows servers to register additional operations and for clients to invoke these operations. Furthermore, this RPC layer provides a mechanism for inter-server communication. It is possible for servers to issue RPC requests to remote servers so that they can work together in a coordinated way or share data. The RPC layer can function over a loopback device for local communication (intra-server) or over BMI for distributed communication (inter-server).

3.3 IOFSL Scalable Logging Tools

To enable uncoordinated, scalable support for log-structured access patterns, we implemented two optimizations within IOFSL. First, we provided a server-managed, non-blocking I/O capability. Second, we implemented an atomic file append capability.

Atomic append allows multiple clients to share the same output file without any client-side coordination. This file append capability is a distributed and atomic I/O operation; clients can append data to a file that is simultaneously being written to by other I/O forwarding servers. IOFSL clients do not require prior knowledge of the end-of-file and are required only to deliver the data to be written into the file to the server. The server returns the file offset at which the data was written to the the client. While this capability is similar to MPI shared file pointers and the O_APPEND mode provided by the POSIX I/O API, the novelty of our approach is that it is implemented in our portable I/O forwarding layer and is limited by file systems that do not provide adequate support for this mode. For example, support for O_APPEND is not provided by all file systems, such as NFS. Since this capability is implemented within IOFSL, it can be used on any system where IOFSL can run, regardless of the underlying file system, operating system, or network constraints of that system.

The IOFSL non-blocking I/O capability provides asynchronous file I/O enhancements to the IOFSL server and client. This capability transforms blocking IOFSL I/O operations into non-blocking I/O operations while requiring minimal changes to the application and no changes to the ZOIDFS API. When using this capability, a IOFSL server will complete an I/O operation for a client after the server has received all necessary data from the client. This behavior allows the IOFSL server to transparently manage asynchronous I/O operations initiated by clients. While recent work has addressed the use of asynchronous I/O at the I/O forwarding layer [30], our work focuses on providing a portable, asynchronous I/O capability in HPC environments.

The IOFSL atomic append and non-blocking I/O capability can be used simultaneously by multiple processes within a parallel application and for any ZOIDFS write operation within a specific process. This effectively allows applications to stream data to the IOFSL servers and allow the IOFSL servers to manage data storage within a file. Sufficient information is returned from each of these capabilities so that users can construct an index of their data accesses within the file and determine the state of non-blocking operations after they were submitted to the IOFSL server.

To support both of these new IOFSL features, we added hooks into the IOFSL server and client to change the IOFSL server data path when either of these features are initiated by a user. These hooks allow the IOFSL server to function normally (bypassing these optimizations) unless the user of the IOFSL client indicates that an optimization should be enabled. To enable these features, the user of the IOFSL client adds hints to the IOFSL I/O request that describe the optimization. Hints are a generic mechanism that allows clients and servers to pass additional information or context about a ZOIDFS operation. The hints are also used to transport results from the IOFSL server back to the IOFSL client. This allows the server to pass to the client additional information about the operation or the enabled optimization that is not easily represented in the existing ZOIDFS API. For example, the hint passed back from the server during an atomic append operation contains the file offset where the IOFSL server stored the block of data.

Upon receiving a ZOIDFS request with an enabled atomic append or non-blocking I/O optimization, the IOFSL server alters the I/O requests' data path. Normally, the server receives an IOFSL write request from the client, allocates the necessary network buffers to receive the data from the client, issues the file system I/O operation, and then returns the result of this operation to the IOFSL client. Upon receipt of a non-blocking I/O request, the server will transfer the request's file data to a dedicated memory pool and complete the client request. Internally, the IOFSL server manages this memory pool and continues to process pending write requests stored in the pool. If there is no memory available in this pool, the server will overide the users non-blocing I/O request and fallback to the normal blocking I/O mode. The non-blocking I/O capability provides a server-side completion mechanism for IOFSL clients. No request tracking, testing, or waiting is necessary on the client side with this optimization, however, it requires the use of the ZOIDFS commit operation to flush any pending non-blocking I/O requests before the application or IOFSL server terminates. The commit operation will indicate if there were any failures for the non-blocking I/O operations.

For atomic append operations, the IOFSL server will determine the end of the file and reserve space for a pending I/O request at the end of the file. IOFSL tracks the end of the file in a hash table that is globally accessible between all the servers. Information stored in this table is keyed on the file path and IOFSL file handle. To prevent a sin-

gle source of failure and potential bottlenecks, the keys are hashed and the values for each key are distributed among all active servers. Access to this table is both distributed and atomic. The servers execute RPC requests between each other to access this table (if the information is not locally available) and update the atomic offset value for a file. Additionally, the server can manage this table in a stand-alone mode. In this mode, the RPC mechanism is bypassed and the server responds only to local access requests to the hash table. This mode is useful for situations where the RPC cost is prohibitive to good file I/O performance or for situations where globally accessible information is not necessary.

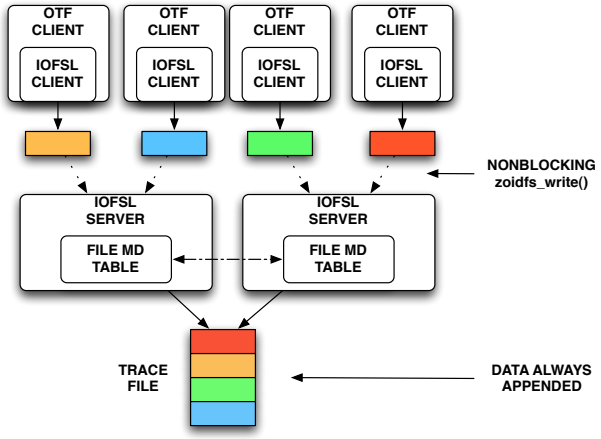

Figure 6: IOFSL with non-blocking I/O and the atomic append feature for an Nx1 I/O pattern

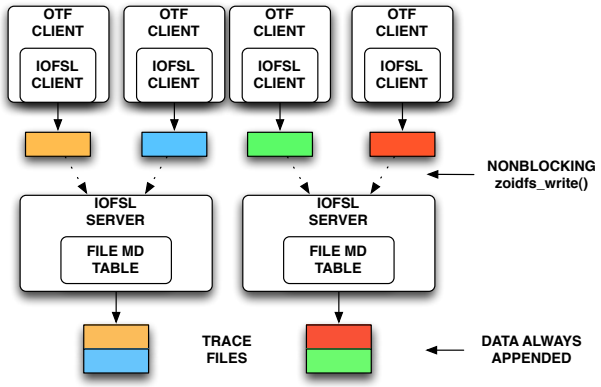

Figure 7: IOFSL with non-blocking I/O and the atomic append feature for an NxM I/O pattern

There are two possible I/O patterns using the atomic append capability. The first pattern aggregates and appends I/O operations to a file shared by all clients interacting with a single IOFSL server. This mode allows reductions in the number of files to be on the order of the number of active IOFSL servers. This I/O pattern is depicted in Figure 7. The second I/O pattern using the atomic append mode is to perform a distributed, atomic append I/O operation to a single file shared by all processes and I/O forwarding servers. This pattern is depicted in Figure 6. This pattern requires communication between the IOFSL servers to track the end-of-file position for the shared file. It is also possible to build more complex I/O patterns from a combination of these modes so that the file system can be best utilized. For example, IOFSL servers could be separated into groups and each group performs a distributed atomic append operation to a unique file shared only within the group of IOFSL servers.

Since both of these features are implemented within IOFSL and do not rely on specific file system or operating system features, the optimizations are portable to any system that can run IOFSL. To date, we have evaluated these features on Linux clusters and Cray XT systems.

3.4 Integrating IOFSL with OTF

To integrate the I/O Forwarding and Scalability Layer into the VampirTrace stack, we decided to choose the OTF layer as a single integration point. Since all I/O happens in this layer, this permits a portable solution that is also usable for other applications based on OTF. We chose to use the ZOIDFS APIs directly instead of transparently intercepting the existing POSIX I/O calls, because OTF's file management was altered.

The main goal of the modification was write to n OTF event streams (originating from n processes/threads of the target application) to m shared files, where $m \ll n$, instead of n private files. Either, m coordinated IOFSL servers write to a single file or each server uses a separate file to save coordination costs, see also Figures 6 and 7.

To accomplish this, all ZOIDFS write operations use the novel atomic-append feature of IOFSL. This allows arbitrary subsets of event trace streams to share the same file without any coordination on the OTF side. IOFSL ensures that blocks from the same source stay in their original order, but makes no guarantees with respect to global ordering; this enables additional optimizations.

The coordination of blocks and their positions in the shared file is done by OTF. Every OTF stream collects the file positions of its blocks individually in memory. As a final new step, all OTF streams write a list of file position for the own blocks together with the shared file ID and their stream ID. This is sufficient to extract all blocks from this stream in original order later during reading[2]. This mapping is stored in a single shared index file, which is also written via IOFSL for convenience.

The traditional OTF write scheme uses synchronous I/O calls in order to guarantee that all actual I/O activities happen during the buffer flush phases which is explicitly marked in the trace itself. IOFSL provides an easy option to use non-blocking I/O instead. Thus the I/O calls return much faster but the actual offloading of trace data may happen after the buffer flush phase. In cases where I/O perturbations from non-blocking I/O can not be tolerated, this feature should be avoided. For example, if the target application's I/O is the subject of the analysis or if the machine's I/O network is not separate from the communication network, the non-

[2]Reading can be done via IOFSL or traditional POSIX I/O.

blocking I/O capability may not be appropriate and it's impact on tracing results should be analyzed. Otherwise, the non-blocking should be enabled. Enabling or disabling this feature is controlled via an environment variable.

4. DEMONSTRATION AND DISCUSSION

JaguarPF [5] is a 2.3 Petaflop Cray XT5 Supercomputer deployed at the Leadership Computing Facility (LCF) at Oak Ridge National Laboratory (ORNL). It consists of 18688 nodes, each node with 2 processor sockets, each socket hexcore. Meeting the I/O needs of JaguarPF is a Lustre-based center-wide file system named Spider is deployed [29]. Spider is designed to provide over 240 GB/s of aggregate throughput and over 10 petabytes of formatted capacity spread over three separate file systems. The storage devices are hosted by 192 Cray service I/O (SIO) nodes.

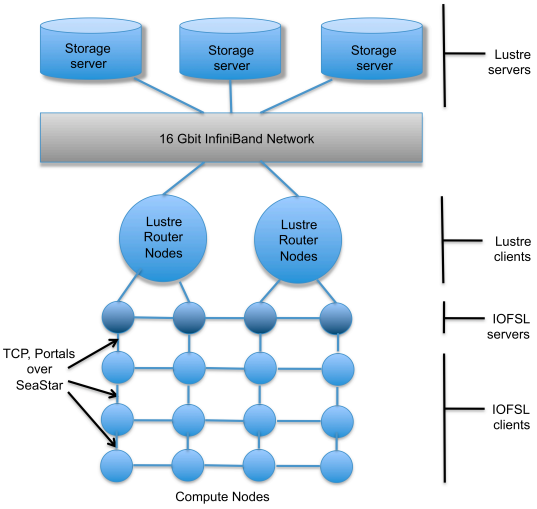

Figure 8: Deployment of application processes and IOFSL servers on JaguarPF

User level access to the I/O nodes or Lustre router nodes, which would be optimal locations for running IOFSL servers, is not possible on JaguarPF due to local policies. Therefore, we allocate additional compute nodes with each application launch, spawn IOFSL servers on these extra nodes, and proxy all application I/O requests through these nodes. Figure 8 illustrates this deployment strategy.

To achieve this setup, we provide additional functions for the PBS job script that selects a set of nodes for hosting the IOFSL servers and determines the network address of these nodes. In the background, the script then launches one IOFSL server per node (a 1:12 processor core ratio on the 12-core Jaguar XT5 nodes) using the Cray *aprun* application launcher command. Once all IOFSL servers are active, a directory of the servers is passed to the application. The OTF layer then parses the directory and determines which IOFSL each OTF-enabled process will utilize. The application issues I/O requests through the IOFSL client using either the BMI TCP/IP or Portals drivers.

To utilize tracing at a large scale we instrumented the Petascale application S3D with VampirTrace. S3D is a parallel direct numerical simulation code developed at Sandia National Laboratories [7]. We utilized a problem set that scales to the full Jaguar system. In its leadership role as a Petascale code, S3D is well understood and has been previously analyzed with TAU and Vampir on lower scales [16].

Prior to our successful demonstration, the largest scale trace for VampirTrace was approximately 30,000 processes using POSIX I/O. In practice, this level of parallelism is already difficult due to substantial overhead during file generation. We use a S3D run with 8640 processes to compare the impact of regular tracing I/O with our forwarding solution. In our demonstration we utilize the full stack that is involved in trace generation: application (S3D), VampirTrace, OTF, IOFSL, the BMI Portals driver for network transfers, and Lustre as a target filesystem.

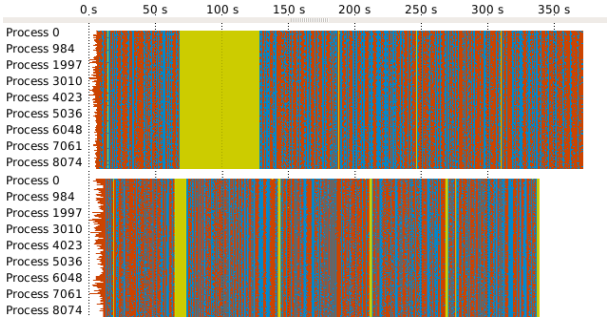

Figure 9: Vampir process timeline for S3D with 8640 processes: (top) POSIX I/O (bottom) IOFSL

Figure 9 shows the effect of using our I/O forwarding solution to manage the trace I/O during the application execution (the horizontal time axes are aligned). In the displayed trace timelines, yellow areas show the interruption of the application during a synchronous buffer flush in VampirTrace. The first buffer flush involves the creation of all output files and therefore takes a much longer time to complete than the following ones. Our forwarding solution is able to significantly reduce the impact of the file creation during the first flush, reducing the impact on the application execution and the overall runtime.

The POSIX version cannot be scaled much further because of the previously mentioned I/O constraints. The IOFSL version has been successfully scaled up to 200,448 cores running S3D using a set of 672 I/O forwarding nodes. In multiple experiments traces up to 2.4 TiB have been generated. In a setup with a single buffer flush at the application end, it took 178 seconds to generate the trace data files with a total volume of 415 GiB. Besides the actual I/O, this measurement also includes the time for creating the target files, establishing connections to the forwarding servers, event data compression, VampirTrace shutdown and MPI finalize, but not the execution of the VampirTrace post-processing tools.

Validation of the trace files generated using IOFSL was performed with the use of the post-mortem analysis tool Vampir. Vampir is able to read valid trace files and display detailed graphics of measured events versus a timeline and various other displays.

Our success in scaling the trace generation revealed new bottlenecks in post-processing. Work is ongoing to improve the ability of the post-mortem (post-processing) tools to deal with this new era of multi-terabyte trace files (particularly at scales above 86,400). Furthermore, we have noticed new "non I/O" perturbation of the target program while it is analyzed at scale. One possible cause are inserted `MPI_Allreduce` calls, that are used by VampirTrace for synchronizing the buffer thresholds. Also the visualization for analysis becomes more challenging at the large scales—the ratio between available pixels and displayed processes increases even further. New ways to highlight performance anomalies are required to support the user at those scales. All of this is subject of ongoing research; our solution lays a foundation for a comprehensive analysis at full scale by providing a feasible way to persist event data collected from an instrumented application.

Note to SC'11 Reviewers from Paper Authors:

Additional scaling results showing wallclock runtimes for S3D in three modes will be incorporated into the final paper:

(a) without tracing up to full machine,
(b) with tracing through POSIX I/O up to the scales that work,
(c) with tracing through IOFSL I/O up to full machine.

If the reviewers feel shepherding the paper after the inclusion of the additional data is helpful, we would welcome that.

5. RELATED WORK

The POSIX I/O standard was designed before the advent of wide-scale parallelism. As such, it suffers from many fundamental characteristics which preclude it from scenarios such as multiple writers updating the same file—a common need for parallel I/O oriented activity [14].

Important new I/O research efforts with standards-oriented activities have recognized this fact and are actively working on APIs appropriate for extreme scale parallelism [14, 28]. One such API is pNFS [13], an extension to NFSv4 designed to overcome NFS scalability and performance barriers. Like IOFSL, it is based on a stateless protocol. It does not provide the "n-to-m" client to forwarding-server architecture fundamental to our design, however, and is unable to coalesce independent accesses to improve performance. We plan to incorporate a direct connection from IOFSL to pNFS as an alternative lower layer for platforms using it.

MPI-IO [21] provides a more advanced I/O abstraction than POSIX, including collective operations and file views, which enable coordinated and concurrent access without locking [8]. It does not directly provide an "n-to-m" mapping from clients to output files. OTF's management of mixed blocks in shared files would be difficult to implement on top of MPI-IO, because most implementations (including the popular ROMIO implementation), do not return accurate current file sizes unless a synchronizing collective is used.

The I/O Delegate Cache System (IODC) [25] is a caching mechanism for MPI-IO which resolves cache coherence issues and hence alleviates the lock contention of I/O servers. IOFSL offers similar capabilities, but it sits below MPI-IO in the I/O software stack, providing a dedicated abstract device driver enabling unmodified applications to take full advantage of its optimizations.

The I/O forwarding concept was introduced with the Sandia Cplant project [27], which used a forwarding framework based on an extended NFS protocol. IOFSL extends the target environment imagined by Cplant to much larger scales and higher performance through a more sophisticated protocol permitting additional optimizations.

Functionally, our work is most closely related to current proprietary approaches for I/O forwarding available from IBM and Cray. IBM's Control and I/O Daemon (ciod) is the Blue Gene I/O-forwarding infrastructure [31]. Each process in the compute node partition forwards I/O requests to the ciod daemon running on an I/O node. The requests are handled by I/O proxy processes. Cray XT's Data Virtualization Service (DVS) is a distributed network service that provides transparent access to file systems on the service I/O (SIO) nodes from the compute nodes [10]. In contrast to ciod and DVS, IOFSL is a portable, open source I/O forwarding stack. In addition, IOFSL realizes its performance through its aggregation and coordination protocol and also incorporates optional data compression.

Decoupled and Asynchronous Remote Transfers (DART) [11] and DataStager [2] achieve high-performance transfers on Cray XT5 using dedicated data staging nodes. Unlike IOFSL, which is transparent to the applications that use POSIX and MPI-IO interfaces, DART requires applications to use a custom API. With this, however, DART provides the building blocks for higher-level data services such as filtering.

Similarly, Adaptable I/O System (ADIOS) [20] provides application-specific performance improvements through prefetch and write-behind strategies based on application-specific configuration files read at startup; this information also helps ADIOS to minimize the memory footprint during the course of the application run. It provides a high-level I/O API that can be used in place of data formatting libraries like HDF5 and netCDF to provide much more aggressive write-behind and log-like reordering of data location within a data file. This technique requires application modification. In contrast, IOFSL requires no knowledge of the application behavior in advance and it is situated at a lower level in the I/O software stack.

The SION library [12] intercepts POSIX I/O, not unlike IOFSL, to emulate a large number of logical independent files (e.g., one per rank/process/thread) and map them to a lesser number of physical files (from the file system perspective). It requires no server processes but causes unforeseeable synchronization to the parallel execution, which

would be critical for VampirTrace. Furthermore, it always requires modification of the application source code and depends upon MPI for internal coordination, which is infeasible for monitoring non-MPI parallel applications.

PLFS [4] is a file system translation layer developed for HPC environments to alleviate scaling problems associated with large numbers of clients writing to a single file. Like our solution, they interpose middleware between the client application and the underlying file system through the use of FUSE. Their solution, which is aimed at checkpointing and similar activities for architectures like LANL's Roadrunner (3060 nodes), transparently creates a container structure consisting of subdirectories for each writer as well as index information and other metadata for each corresponding data file. As our solution is focused on supporting hundreds of thousands of clients or more, we have chosen to aggregate I/O operations in the middleware thus resulting in fewer metadata operations in the underlying parallel file system.

IOFSL work extends our earlier ZOID efforts [15]. ZOID is a Blue Gene-specific function call forwarding infrastructure that is part of the ZeptoOS project; our I/O forwarding protocol was first prototyped there. IOFSL is a mature, portable implementation that integrates with common HPC file systems and also works on the Cray XT series and Linux clusters.

6. CONCLUSIONS AND FUTURE WORK

This paper described a new I/O forwarding layer designed to provide advanced I/O request aggregation and reorganization. Our scheme is designed to impose minimal restrictions on the target platform while achieving the fullest benefit of data aggregation and reorganization. We have demonstrated that our I/O solution enables software tracing on full-scale leadership class systems (200,448 cores). A comprehensive trace-based analysis is now feasible for pattern recognition, post-processing and visualization systems.

We attribute our improved scalability of bandwidth and metadata operations to our novel design. Our scheme utilizes I/O forwarding for data as well as metadata aggregation. We incorporated further optimizations specific to scalable logging tools, including a specialized atomic append, event compression at selectable transfer sizes (e.g., Lustre stripe width), and various tuning parameters including non-blocking I/O strategies.

While these results meet our immediate needs and objectives, this effort has led us to consider still further related lines of inquiry. We would like to pursue more advanced aggregate memory footprint optimizations to yield more available memory to user applications. While we have addressed the data collection challenges in this paper and presented a solution to this problem, we do not address how to effectively visualize trace data for applications running at extreme scales. This information visualization challenge must be addressed as our work progresses. We plan to couple the data collection tools and techniques presented in this paper with recent MPI and I/O visualization tools that focus on extreme scale HPC event and trace data collections [23, 22]. Furthermore, we plan to optimize the ratio of IOFSL servers to clients and to provide a more convenient automatic deployment of IOFSL server processes. Finally, we have plans to pursue data collection for multiple leadership class machines.

Acknowledgements

The authors would like to thank Ramanan Sankaran (ORNL) for providing a working version of S3D and a benchmark problem set for JaguarPF. The IOFSL project is supported by the DOE Office of Science and National Nuclear Security Administration (NNSA). This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory and the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory, which are supported by the Office of Science of the U.S. Department of Energy under contracts DE-AC02-06CH11357 and DE-AC05-00OR22725 respectively. Computational resources were provided by NSF-MRI Grant CNS-0821794, MRI-Consortium: Acquisition of a Supercomputer by the Front Range Computing Consortium (FRCC), with additional support from the University of Colorado and NSF sponsorship of the National Center for Atmospheric Research.

The general enhancement of the VampirTrace and Vampir tools at TU Dresden for full-size runs on leadership-class HPC systems is supported with funding and cooperation by ORLN and UT Battelle.

7. REFERENCES

[1] SYSIO. `http://sourceforge.net/projects/libsysio`.

[2] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. DataStager: Scalable data staging services for petascale applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC '09)*, pages 39–48, New York, NY, USA, 2009. ACM.

[3] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable I/O Forwarding Framework for High–Performance Computing Systems. In *IEEE International Conference on Cluster Computing 2009*, 2009.

[4] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, New York, NY, USA, 2009. ACM.

[5] A. Bland, R. Kendall, D. Kothe, J. Rogers, and G. Shipman. Jaguar: The world's most powerful computer. In *Proceedings of the Cray User's Group*, 2009.

[6] P. Carns, W. Ligon III, R. Ross, and P. Wyckoff. BMI: A network abstraction layer for parallel I/O. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, Workshop on Communication Architecture for Clusters (CAC '05)*, 2005.

[7] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale

direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):015001, 2009.

[8] A. Ching, A. Choudhary, K. Coloma, W. Liao, R. Ross, and W. Gropp. Noncontiguous I/O access through MPI-IO. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '03)*, pages 104–111, 2003.

[9] J. Cope, K. Iskra, D. Kimpe, and R. Ross. Bridging HPC and Grid file I/O with IOFSL. In *Para 2010: State of the Art in Scientific and Parallel Computing*, 2010.

[10] Cray XT system software 2.1 release overview.

[11] C. Docan, M. Parashar, and S. Klasky. DART: A substrate for high speed asynchronous data IO. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC '08)*, pages 219–220, New York, NY, USA, 2008. ACM.

[12] W. Frings, F. Wolf, and V. Petkov. Scalable massively parallel I/O to task-local files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 17:1–17:11, New York, NY, USA, 2009. ACM.

[13] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '05)*, pages 18–27, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[14] IEEE POSIX Standard 1003.1 2004 Edition. `http://www.opengroup.org/onlinepubs/000095399/functions/write.html`.

[15] K. Iskra, J. W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 153–162, New York, NY, USA, 2008. ACM.

[16] H. Jagode, J. Dongarra, S. Alam, J. Vetter, W. Spear, and A. D. Malony. A holistic approach for performance measurement and analysis for petascale applications. In *Proceedings of the International Conference on Computational Science, part II (ICCS '09)*, volume 5545 of *Lecture Notes in Computer Science*, pages 686–695. Springer-Verlag, 2009.

[17] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the Open Trace Format (OTF). In V. N. Alxandrov, G. D. Albada, P. M. A. Sloot, and J. J. Dongarra, editors, *6th International Conference on Computational Science (ICCS)*, volume 2, pages 526–533, Reading, UK, 2006. Springer.

[18] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel. The vampir performance analysis tool-set. In M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer Verlag, July 2008.

[19] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[20] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE '08)*, pages 15–24, New York, NY, USA, 2008. ACM.

[21] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/docs.html`, 1997.

[22] C. Muelder, F. Gygi, and K.-L. Ma. Visual Analysis of Inter-Process Communication for Large-Scale Parallel Computing. *IEEE Transaction on Visualization and Computer Graphics*, 15(6):1129–1136, November/December 2009.

[23] C. Muelder, C. Sigovan, K.-L. Ma, J. Cope, S. Lang, K. Iskra, P. Beckman, and R. Ross. Visual Analysis of I/O System Behavior for High–End Computing. In *Workshop on Large-Scale System and Application Performance (LSAP2011), in conjunction with the 20th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2011)*, 2010 (to appear).

[24] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel. Developing scalable applications with vampir, vampirserver and vampirtrace. In C. Bischof, M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications*, volume 15 of *Advances in Parallel Computing*, pages 637–644. IOS Press, 2008.

[25] A. Nisar, W. Liao, and A. Choudhary. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC '08)*, 2008.

[26] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa. Optimization Techniques at the I/O Forwarding Layer. In *IEEE International Conference on Cluster Computing 2010*, 2010.

[27] K. Pedretti, R. Brightwell, and J. Williams. Cplant™ runtime system support for multi-processor and heterogeneous compute nodes. In *Proceedings of the 4th IEEE International Conference on Cluster Computing (CLUSTER '02)*, pages 207–214, 2002.

[28] Petascale Data Storage Iinstitute. `http://www.pdsi-scidac.org/`.

[29] G. Shipman, D. Dillow, S. Oral, and F. Wang. The Spider center wide file system; from concept to reality. In *Proceedings of the Cray User's Group*, 2009.

[30] V. Vishwanath, M. Hereld, K. Iskra, D. Kimpe, V. Morozov, M. Papka, R. Ross, and K. Yoshii. Accelerating I/O Forwarding in IBM Blue Gene/P Systems. In *SC 2010*, 2010.

[31] H. Yu, R. K. Sahoo, C. Howson, G. Almási, J. G.

Castaños, M. Gupta, J. E. Moreira, J. J. Parker, T. E. Engelsiepen, R. B. Ross, R. Thakur, R. Latham, and W. D. Gropp. High performance file I/O for the Blue Gene/L supercomputer. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA '06)*, pages 187–196, 2006.