Noname manuscript No.
(will be inserted by the editor)

Optimizing I/O Forwarding Techniques for Extreme-Scale Event Tracing

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

the date of receipt and acceptance should be inserted later

Abstract Programming development tools are a vital component for understanding the behavior of parallel applications. Event tracing is a principal ingredient to these tools, but new and serious challenges place event tracing at risk on extreme-scale machines. As the quantity of captured events increases with concurrency, the additional data can overload the parallel file system and perturb the application being observed. In this work we present a solution for event tracing on extreme-scale machines. We enhance an I/O forwarding software layer to aggregate and reorganize log data prior to writing to the storage system, significantly reducing the burden on the underlying file system. Furthermore, we introduce a sophisticated write buffering capability to limit the impact. To validate the approach, we employ the Vampir tracing toolset using these new capabilities. Our results demonstrate that the approach increases the maximum traced application size by a factor of 5x to more than 200,000 processes.

Keywords event tracing, I/O forwarding, atomic append

T. Ilsche · A. Knüpfer · W. E. Nagel
Technische Universität Dresden, ZIH
01062 Dresden, Germany
E-mail:
{thomas.ilsche,andreas.knuepfer,wolfgang.nagel}@tu-dresden.de

J. Cope · D. Kimpe · K. Iskra · R. Ross
Argonne National Laboratory, 9700 South Cass Avenue,
Argonne, IL 60439, USA
E-mail: {copej,dkimpe,iskra,rross}@mcs.anl.gov

J. Schuchart · T. Jones · S. Poole
Oak Ridge National Laboratory, Mailstop 5164
Oak Ridge, TN 37831, USA
E-mail: {schuchartj,trjones,spoole}@ornl.gov

1 Introduction

The rising levels of concurrency in large-scale computing systems present a number of challenges to parallel application programmers. Scaling to large number of cores introduces performance and correctness effects which do not always appear when running on smaller systems. At the same time, the performance analysis tools vital to the high-performance computing (HPC) software ecosystem often do not scale to these large systems. In addition, those that do frequently exhibit larger overheads, perturbing the runtime behavior of the analyzed application, destroying the very behavior the application developer intends to analyze. One challenge to scaling these tools is efficiently storing the recorded trace data. Tracing tools generate large amounts of data and must execute efficiently at full scale. Traditional access patterns for these tools, such as file per process, file per thread, and synchronous I/O, do not scale well past tens of thousands of processes. Such patterns require excessive use of file metadata operations and overwhelm large-scale storage systems. Synchronous I/O may cause unnecessary delays in trace data collection and skew application execution. Alternative access patterns, such as a shared file pattern, may alleviate metadata bottlenecks but inject artificial synchronization into the application. Therefore, a unique data organization is desired that exploits the loglike I/O behavior of performance analysis tools, allows for uncoordinated access to a shared file from multiple processes, and tolerates lazy I/O semantics.

To achieve and sustain full-size application event trace recording on large-scale systems, we investigated several I/O optimizations to support high-performance, scalable, and uncoordinated write access for event trace data generated by the Vampir toolset [25]. We observed that the uncoordinated I/O patterns generated by the VampirTrace and Open Trace Format (OTF) tools could be transparently optimized at an intermediate file I/O aggregation layer, known as the I/O forwarding layer. We integrated the I/O Forwarding Scalability Layer (IOFSL) [4, 33] with the VampirTrace/OTF toolset. Also, as part of contributions of this paper, we implemented optimizations and new capabilities within IOFSL to reorganize and optimize the captured VampirTrace/OTF I/O patterns while preserving the independent I/O requirements of these tracing tools. These new features include a distributed atomic file append capability and a write buffering capability. By taking advantage of the characteristics of the event trace workload and augmenting our HPC I/O stack to better support it, we have reduced the stress that the trace I/O workload places on HPC storage systems. Furthermore, we have reduced the impact of HPC I/O storage systems on the tracing tools and thus the perturbation of the analyzed application.

The work described in this paper led to a significant performance and scalability improvement of the VampirTrace and OTF software stack, enabling tracing of up to 5 times more cores (bringing the maximum number of traced cores from 40,000 to 200,000 on the system evaluated). The corresponding trace contained 941 billion events generated at an aggregate rate of 13.3 billion events per second, validating our approach of coupling the Vampir toolset and IOFSL. Overall, we have shown that the entire software stack including trace generation, middleware, postprocessing, and analysis can be utilized to analyze a parallel application consisting of more than 200,000 processes, yielding a performance analysis framework suitable for end users on large-scale computing systems.

The remainder of this paper is organized as follows. The general I/O requirements of performance analysis tools and the Vampir toolset I/O needs are described in Section 2. An overview of IOFSL and optimizations relevant to tracing is provided in Section 3. Section 4 describes the integration of I/O forwarding in the Vampir toolset and general scalability improvements for trace recording and processing. The proposed concepts are evaluated at scale, followed by an analysis of the results in Section 5. An overview of related work is given in Section 6. Conclusions and insights into future work are summarized in Section 7.

2 The Vampir Toolset

The Vampir toolset is a sophisticated infrastructure for performance analysis of parallel programs using combinations of MPI, OpenMP, Pthreads, CUDA, and OpenCL. It consists of the Vampir GUI for interactive post-mortem visualization, the VampirServer for parallel analysis, the VampirTrace instrumentation and runtime recording system, and the Open Trace Format as the file format and access library. The toolset relies on event trace recording, which allows

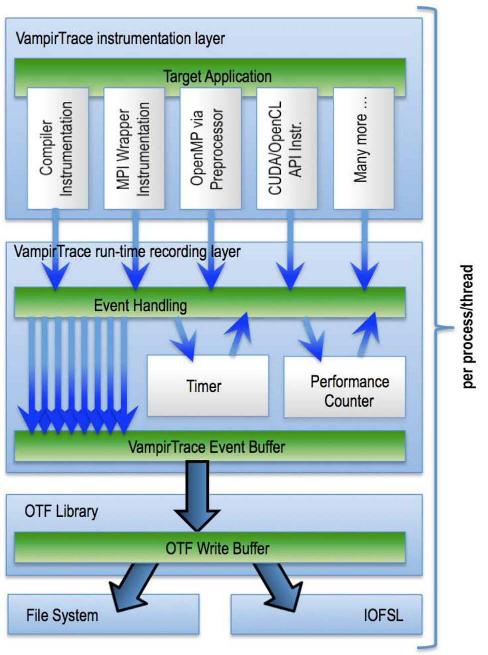

Fig. 1 The VampirTrace data flow.

the most detailed analysis of the parallel behavior of target applications. Figure 1 gives an overview of the data flow through VampirTrace's runtime infrastructure. In a first step, the target application is instrumented by VampirTrace using various techniques. The VampirTrace runtime recording layer collects the events generated by the instrumentation, thereby striving to impose minimal perturbation. Triggered by events of interest, this layer stores the event information together with vital properties (precise time stamp, event-specific properties, performance metric values if configured) to preallocated memory buffers. In order to avoid artificial synchronization of target applications, these buffers are never shared among processes and threads. The events of interest include entry and exit of user-code subroutines, message send and receive events, collective communication events, shared-memory synchronization, and I/O events.

Depending on its type, a single Vampir event requires approximately 10 to 50 bytes for its encoding in the buffer. With event frequencies ranging from 100 to 100,000 events per second (with proper settings), the amount of data for 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 27 TB[1]). In order to avoid severe application distortion, the sum of the size of the buffers and the memory required by the application should never exceed the local main memory size. Typical buffer sizes range from 10 MB to 1 GB per process or thread, depending on the projected amount of generated trace data. This event trace data is written to a set of OTF files from where it can be used for

[1] In this paper, we use $1\,\text{MB} = 2^{20}\text{B}$, $1\,\text{GB} = 2^{30}\text{B}$, and $1\,\text{TB} = 2^{40}\text{B}$.

post-mortem investigation with the Vampir GUI. Traditionally, VampirTrace and OTF use a file-per-thread I/O pattern for storing data to minimize coordination among processes and threads. Additional information about Vampir, VampirTrace, and OTF is provided in our prior work [25].

2.1 I/O Patterns in VampirTrace

VampirTrace supports several strategies for writing buffers to OTF files. The most general way is to flush and reuse a buffer as soon as the measurement library cannot ensure enough space for an event in the buffer. Therefore, the application is delayed until the data is written out through the OTF library. These buffer flush phases are clearly marked in the trace so that their effect is not mistaken for stray behavior of the target application. However, they can delay other processes waiting for messages or synchronization in the application, unless the buffer capacity is reached at the same time in the application for all processes (i.e., each process generated the same number of events). This becomes an issue for larger-scale and tightly coupled applications.

Alternatively, VampirTrace can use collective MPI operations to trigger synchronized buffer flushes by piggybacking work on application collective operations. For each collective operation, the measurement environment communicates the maximum buffer level. Once a threshold is reached, all processes enter a flush phase. All synchronous flushes and synchronizations are captured in the trace and clearly marked for analysis. An additional barrier after the flush makes sure the processes resume simultaneously, avoiding an indirect impact on the application behavior. The global collective operation is marked in the trace file to expose the overhead introduced by the measurement.

Preferably, only a single flush at the end of the recording is required (typically triggered by an `MPI_Finalize` wrapper). This removes trace I/O from the application execution but is only applicable as long as the event buffers are large enough to hold all events generated during the application execution. This can be achieved by reducing the total event count, e.g., by using filters or tracing only specific iterations of the application loops. In order to reduce the resulting file size, transparent compression using zlib in the OTF layer is applied to the written trace data. However, this is not applied to the in-memory buffers to keep the perturbation during trace recording minimal. If additional intermediate flushes are required, the number of buffer flushes per process can be limited in order to avoid uncontrolled use of storage space. Therefore, the overall storage space required per process or thread depends on the maximum number of flushes allowed, the size of the memory buffer, and the compression factor of the trace data.

General buffer flushes during MPI collective operations help largely reduce the pertubation due to I/O as compared to the general on-demand flushes. However, a collective operation cannot be guaranteed to trigger a buffer flush before the buffer capacity is exceeded as it depends on the number of tracing events recorded between two collective operations in the application and the buffer size. It is also difficult to choose appropriate settings that do not require buffer flushes. Therefore, VampirTrace may need to fall back to the general uncoordinated buffer flush that prevents loss of data at the cost of a performance impact. Traditional collective I/O optimizations and methods rely on implicit synchronization, which is not appropriate for VampirTrace as it distorts the measurement. Additionally, the support for dynamic process and thread creation in VampirTrace makes it impossible to predetermine the number of participants in collective I/O operations at any time.

2.2 I/O Challenges and Solutions

The original configuration of VampirTrace imposed two I/O challenges that prevented it from tracing applications running at full scale on large systems. First, the often simultaneous buffer flushes of many processes increase I/O bandwidth pressure on the I/O subsystem. This pressure can delay trace data storage and skew the application trace measurements. The storage targets can get overwhelmed with inefficient, nearly random workloads that can degrade their peak performance by over 60 %. Second, the metadata load for this configuration is high because of the creation of many individual files and the allocation of file system blocks for a large number of I/O operations. According to David Dillow, an Advanced Systems Architect & Developer at the Oak Ridge Leadership Computing Facility and co-chair of the Open Scalable File Systems Technical Working Group, ORNL's JaguarPF is in principle capable of opening one file per process even at the scale of 224,000 processes, taking around 45 seconds (David Dillow, personal communication, September 20, 2011). In production usage, however, such operations have been observed to take five minutes [43]. Parallel file creation requests at a high rate will impact all other users and jobs on the machine.

VampirTrace/OTF supports the use of node-local storage for the intermediate trace I/O and copying the files from local to global directories after the measurement. However, node-local storage is not available on most current large-scale systems. While research indicates that local storage based on solid-state drives or hybrid solutions could alleviate I/O bottlenecks on future systems [26], it is still unclear when this will be widely available for large-scale systems. Furthermore, in-memory file systems such as `tmpfs` [40] do not require physical node-local storage but their usage cannot be considered a solution for performance analysis since it would lead to a significant reduction of memory available to the target application. The effect would be analogous to massively increasing the size of trace buffers.

To address these challenges, we identified several opportunities that allow VampirTrace to store its trace data collections more efficiently while minimizing the overhead of the data accesses on the traced applications. First, we found that a shared-file access pattern may be better suited for large-scale applications than the file-per-thread access pattern. This approach significantly reduces the metadata load seen with the file-per-thread access pattern. Since writing to a shared file from many processes requires coordination among those processes, we opted to store trace data collections in multiple files, where the total number of files is far less than the number of traced processes or threads. Second, we identified an append-only streaming access pattern for storing the trace data collection. Such a pattern is easier for parallel file systems to handle than are random I/O patterns. Third, we recognized that a write buffering strategy can isolate the file system performance from the trace data collection storage, which further reduces the impact of the storage system on trace data collection.

The capabilities required by the improved VampirTrace I/O access pattern are not readily available or adequately supported by vendor-supplied I/O software stacks. The necessary capabilities missing from these stacks include transparent aggregation of uncoordinated I/O requests to a set of files, portable atomic append of file data, and write buffering. These capabilities can be implemented within I/O forwarding layers such as IOFSL. Our I/O forwarding approach provides a convenient solution that integrates with the existing VampirTrace/OTF infrastructure and promises to scale much farther than today's high-end systems.

3 IOFSL I/O Forwarding Layer

The goal of I/O forwarding is to bridge computation and storage components in large-scale systems. With this infrastructure, application file I/O requests are shipped to dedicated resources that aggregate and execute the requests. This approach enables I/O forwarding to bridge compute nodes, networks, and storage systems that are not directly connected, such as on the IBM Blue Gene systems [45]. I/O forwarding middleware aggregates file I/O requests from multiple distributed sources (I/O forwarding clients) to a smaller number of I/O handlers (the I/O forwarding servers). The I/O forwarding server delegates and executes the requests on behalf of the clients, as shown in Figure 2. Since the I/O forwarding layer has access to all the file I/O requests, one can implement file I/O optimizations on both coordinated and uncoordinated file access patterns. These optimizations include coalescing, merging, transforming, and buffering I/O requests. Furthermore, the middleware can incorporate expert knowledge about the underlying file system, thereby

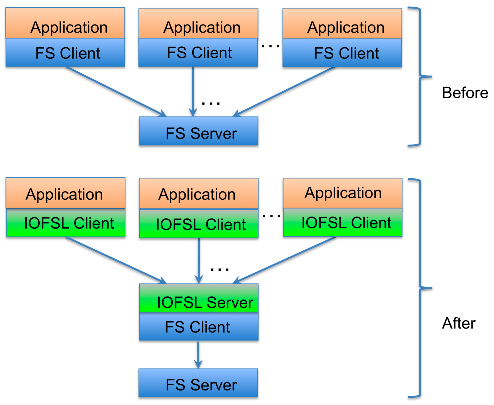

Fig. 2 The principle of I/O forwarding.

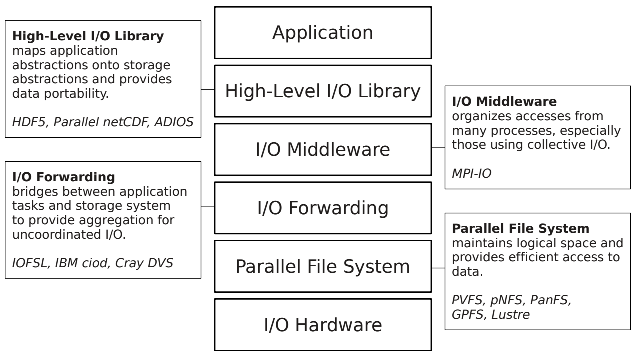
Application
High-Level I/O Library
I/O Middleware
I/O Forwarding
Parallel File System
I/O Hardware
High-Level I/O Library
maps application abstractions onto storage abstractions and provides data portability.
HDF5, Parallel netCDF, ADIOS
I/O Forwarding
bridges between application tasks and storage system to provide aggregation for uncoordinated I/O.
IOFSL, IBM ciod, Cray DVS
I/O Middleware
organizes accesses from many processes, especially those using collective I/O.
MPI-IO
Parallel File System
maintains logical space and provides efficient access to data.
PVFS, pNFS, PanFS, GPFS, Lustre

Fig. 3 Overview of the HPC I/O software stack.

providing an abstract interface for optimized I/O requests and relieving the application from having to deal with file system specific I/O optimizations.

The I/O forwarding layer is an ideal location to prototype and evaluate new or existing HPC I/O capabilities. Figure 3 illustrates a typical HPC software stack on large-scale systems and where I/O forwarding fits into it. I/O forwarding interacts directly with the parallel file system. Implementing the enhancements we present in this paper required no changes to vendor-supplied systems software; all our I/O enhancements were implemented in our I/O forwarding layer IOFSL. Since IOFSL is positioned just above the file system, our enhancements can affect all applications using the I/O software stack.

3.1 IOFSL Architecture

IOFSL [4, 33] is a high-performance, portable I/O forwarding layer for extreme-scale computing systems. It consists of two primary components: a set of IOFSL clients and a set of IOFSL servers, communicating with each other over the network.

The IOFSL client integrates with the compute-side I/O stack and is responsible for initiating I/O requests with the IOFSL server. An application can invoke the IOFSL client in several ways. IOFSL provides an implementation of the stateless ZOIDFS API that applications can use to directly communicate with IOFSL servers. The goal of the API is to reduce the amount of state required per client in an effort to improve scalability and resilience, and to minimize the number of individual API calls required to transfer complex HPC data structures. ZOIDFS has been specifically designed for HPC workloads and provides many useful features, including portable (location-independent) file handles and list I/O operations (i.e., the ability to operate on noncontiguous file regions within a single operation). Compatibility layers for POSIX and MPI-IO are available that translate standard I/O calls to ZOIDFS requests on the client side. For codes that use MPI-IO, a ZOIDFS driver for ROMIO is available. Codes using the POSIX file I/O API can use a FUSE client or a sysio [1] client. Neither of these options requires any modifications of existing POSIX I/O or MPI-IO calls within user code.

The role of the IOFSL server is to delegate I/O requests. Internally, the IOFSL server implements a number of optimizations in order 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 requests.

IOFSL provides several drivers for interaction with common HPC file systems, e.g., POSIX-based (Lustre, GPFS, etc), PVFS2 file systems [9] through a PVFS2 native driver, and GridFTP servers [13]. 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 that provides the server with information about the runtime environment and the IOFSL capabilities to enable.

For communication between clients and servers, IOFSL uses the Buffered Message Interface (BMI) library [8]. BMI is a portable communication layer that was originally used within the PVFS2 file system. It provides asynchronous, list I/O interfaces for network data transfers. Internally, BMI supports many common HPC networks using their native APIs; to date, this list includes native access to the SeaStar2+ network used on the Cray XT platforms using the Portals API, the IBM Blue Gene/P tree network using ZOID [22], 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.

3.2 I/O Forwarding Optimizations

In Section 2.2, we identified several I/O optimizations that can improve the performance and scalability of the trace data generation of VampirTrace. These improvements include a write buffering strategy to quickly offload trace data from the application compute nodes and an atomic file append capability to reduce random I/O workloads to the file system. In this section, we describe how we implemented these capabilities in IOFSL. Figure 4 provides an overview of both capabilities when writing to a single trace file.

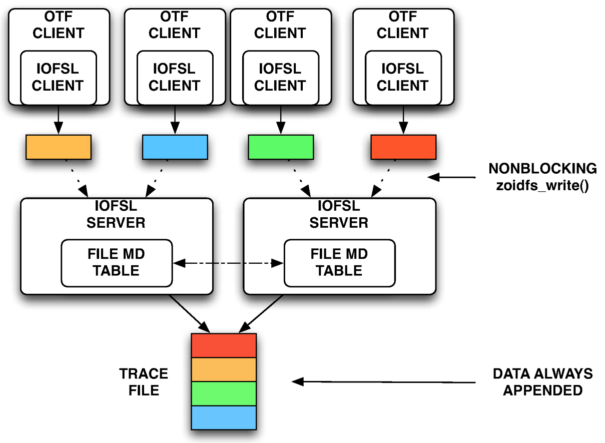

Fig. 4 IOFSL with nonblocking I/O and the atomic append feature for an Nx1 I/O pattern.

The IOFSL write buffering capability extends the IOFSL server and client with nonblocking file I/O enhancements. This capability transforms blocking I/O operations into nonblocking ones while requiring minimal changes to the application and no changes to the ZOIDFS API. It relaxes the data consistency semantics in order to achieve higher I/O throughputs for the application. To implement this, we modified the file write data path in the IOFSL server to signal operation completion to the client before initiating the file write operation on the server. Once the I/O forwarding server receives the client data for a nonblocking operation, the data is buffered within the server, and the client I/O operation is completed. The client is then free to release or reuse its transmitted data buffer since the server is now responsible for completing the I/O operation for the client. The I/O request will complete as soon as the server has resources to process the request or when the client forces all pending nonblocking I/O operations to complete by using a commit operation. This behavior allows the IOFSL server to transparently manage nonblocking I/O operations initiated by clients.

The atomic append capability of IOFSL allows multiple clients to share the same output file without client-side coordination and supports tools that exhibit loglike data access patterns. This file append capability is a distributed and atomic I/O operation. We developed several new IOFSL features to support distributed atomic file appends. IOFSL servers now provide a distributed hash table that is used to track the end-of-file offset for unique files. This data structure provides an atomic fetch-and-update operation. The distributed storage of file handles allows IOFSL to scatter and decentralize the file offset data. We also developed a mechanism for IOFSL servers to communicate with each other. Originally, IOFSL provided a client-server communication model only. The server-to-server communication capability allows IOFSL servers to query remote IOFSL servers and retrieve the end-of-file offset information. Consequently, the clients are not required to contact multiple IOFSL servers in order to obtain and update file offset information. Additionally, the IOFSL server coalesces multiple atomic append requests in order to limit the amount of server-to-server traffic. The atomic append capability can be used by any number of IOFSL servers, including a local server mode when data files are not shared between IOFSL servers and thus neither are end-of-file offsets.

Our atomic append approach has several benefits. Clients can append data to a file that is simultaneously being written to by other clients and I/O forwarding servers. IOFSL clients do not require prior knowledge of the end-of-file position and simply need to deliver to the server the data to be written into the file. This capability effectively allows applications to stream data to the IOFSL servers, which manage data placement within a file. The server returns to the client the file offset where the data will be written. This capability is similar to the O_APPEND mode provided by the POSIX I/O API. The novelty of our approach is in the support of a distributed and portable append functionality: O_APPEND does not work in a multinode environment like a parallel computing system. 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.

4 I/O and Scalability Improvements in the Vampir Toolset

The Vampir toolset has been improved in order to enable extreme-scale tracing. This work can be divided into two categories. First, since I/O was identified as a primary limiting factor, IOFSL support was integrated into trace recording. Second, some more general scalability improvements were implemented to address bottlenecks identified after the improved I/O scheme allowed the trace collection to scale beyond previous limits.

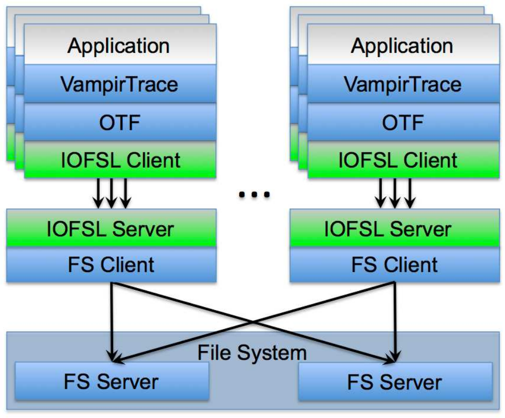

Fig. 5 I/O aggregation provided by IOFSL for VampirTrace / OTF. One forwarding server serves multiple clients; usually many servers are used to provide high capacity.

4.1 Integration of OTF and IOFSL

The OTF layer provides a single integration point between the VampirTrace stack and IOFSL. Since all trace I/O happens in this layer, it permits a portable solution that is usable for other applications based on OTF. We chose to use the ZOIDFS API because it provides additional capabilities that are not present in the IOFSL POSIX translation layers.

The primary integration goal was the reduction of the number of files generated by the Vampir tracing infrastructure. Instead of storing n OTF event streams in n files, the streams are now aggregated into m files, where $m \ll n$; m can vary based on the tracing configuration. For example, m could be equal to the number of IOFSL servers (one file per IOFSL server) or be smaller (files shared between the servers). Figure 5 illustrates the file aggregation and integration of the used software layers.

To accomplish this integration, all OTF write operations use the ZOIDFS 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, which enables additional optimizations by the server.

The coordination of the blocks and their positions in the shared file is managed by IOFSL, which reports the result of this activity to OTF where it is stored in memory individually for every OTF stream[2]. As a final step, OTF writes a list of blocks and their file positions together with the identifier of the stream those blocks belong to. This is sufficient to later extract all blocks of a stream in correct order during reading. The mapping is stored in a shared index file, which is also written via IOFSL using the atomic append capabili-

[2] An OTF stream abstracts the events from a single process or thread.

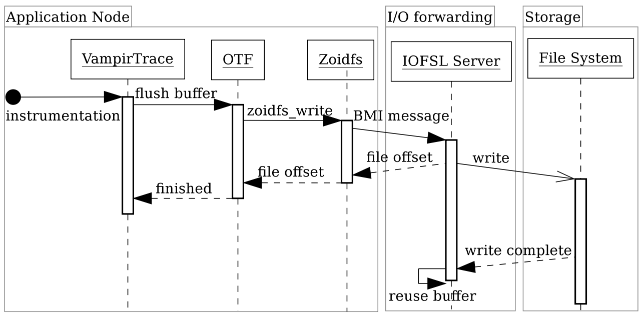

Fig. 6 Sequence diagram of a flush that utilizes the buffered I/O capability of IOFSL. Vertical axis is time, increasing downward, not true to scale.

ties. The trace data files and index files can be read with or without the use of IOFSL.

The traditional OTF write scheme uses synchronous I/O calls to ensure that all I/O activities happen during the buffer flush phases, which are explicitly marked in the trace. While the flush phases are blocking and event trace buffers are reused, there can still be buffering by the operating system, the standard library, or the file system itself, which cannot be eliminated easily by the tracing infrastructure. However, other optimizations that just hide the transfer time can negatively affect the application when resources such as I/O bandwidth are used after the flush.

In contrast, IOFSL's write buffering capabilities can decrease the time spent in buffer flushes without affecting the application. Unlike local optimizations, the trace data will have been transferred from the application node to the I/O forwarding node by the time the flush completes. The file I/O is then initiated on the forwarding node and no local resources are used after that, thus minimizing the application perturbation. The effects are similar to other jobs utilizing the shared network and I/O subsystem. In cases where this is undesirable – 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-blocking I/O capability may not be appropriate and can be disabled. Figure 6 displays the interaction between application, trace library, IOFSL, and the file system in a sequence diagram.

4.2 Scalability Improvements for VampirTrace

In addition to the improvements necessary to efficiently write the trace output, a number of other optimizations were performed to address scalability bottlenecks within the Vampir toolset. Trace postprocessing with `vtunify` was previously parallelized by using OpenMP and MPI. The master `vtunify` process serves as a global instance to unify the trace definitions (metadata about processes, functions, etc). In order to enable the handling of even larger traces, the serial workload in the master process has been significantly reduced. The remaining serial workload was optimized in time complexity with respect to the total number of application processes. A merge option was implemented in `vtunify` that causes each unification worker to only write a single output file instead of one output file per processed stream. This can help avoiding the metadata issues caused by the creation of too many files while it still generates OTF files that are compatible with legacy OTF applications. With these improvements, trace postprocessing became feasible for large scales, as is documented in Section 5. A hierarchical unification scheme for definitions could further improve the scalability and eliminate the master process as a bottleneck.

As described in Section 2.1, a synchronized buffer flush is beneficial for large-scale tracing scenarios. The therefore required threshold check is implemented in terms of injecting a call to `MPI_Allreduce` into each global collective operation. At large scales, this can result in more significant overhead because the `MPI_Allreduce` operation is particularly prone to high variability caused by operating system noise [24]. In order to reduce the total overhead, a configuration option has been introduced that specifies that the threshold of the buffers should be checked only every *n*th collective operation. This mitigates the overhead while still being able to reliably trigger collective synchronized flushes. The additional time used by the measurement library for the threshold check is still marked in the generated trace.

A further enhancement was the improvement of OTF's zlib compression capability. During the OTF and IOFSL integration, the compression capability was updated to ensure that full compression output buffers were written to the file system. This modification ensured that most OTF writes have a fixed size and are stripe aligned, presenting a more efficient pattern to file systems. Unaligned OTF accesses can now only occur at the end of the application's execution when the remaining contents of the compression buffer are flushed to the file system.

VampirTrace and the IOFSL integration presented in this paper were designed and tested with hybrid applications that use MPI in combination with OpenMP, threads, CUDA, or other node-local parallel paradigms. No restriction is imposed on when new threads can be created or when buffer flushes may happen.

5 Evaluation and Analysis

JaguarPF [6] is a 2.3 petaflop Cray XT5 large-scale HPC system deployed at the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory (ORNL). Data storage is provided by a Lustre-based centerwide file system [39].

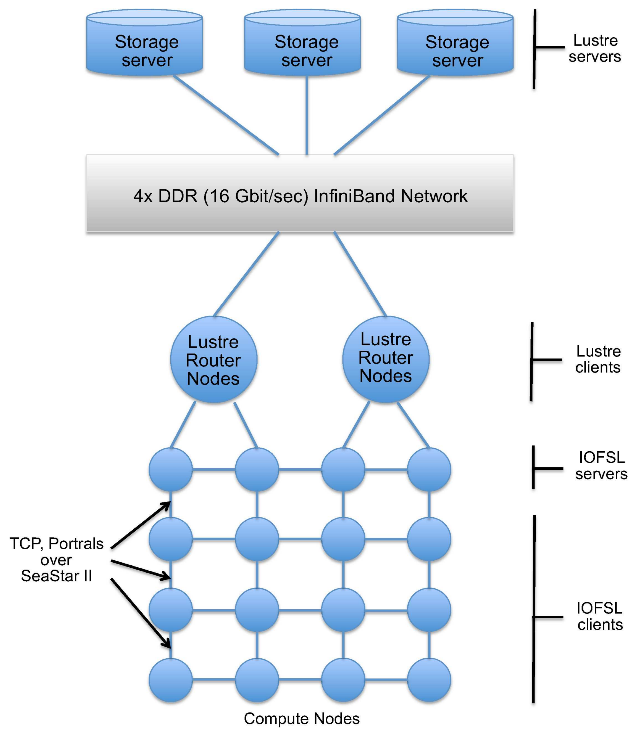

Fig. 7 Deployment of application processes and IOFSL servers on JaguarPF.

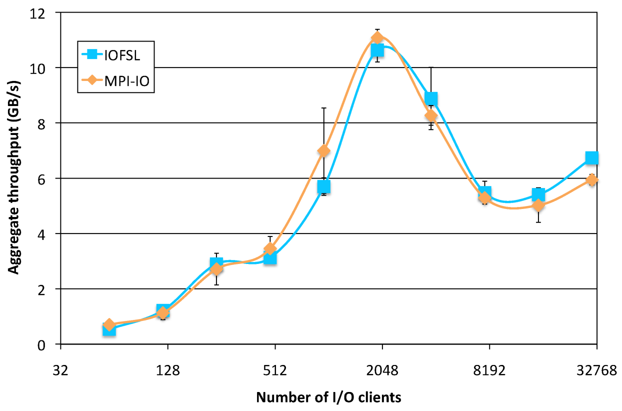

Fig. 8 IOR performance on JaguarPF using a shared-file I/O pattern with MPI-IO and IOFSL. In this experiment we kept the ratio of IOFSL clients to servers constant (12 clients for every server).

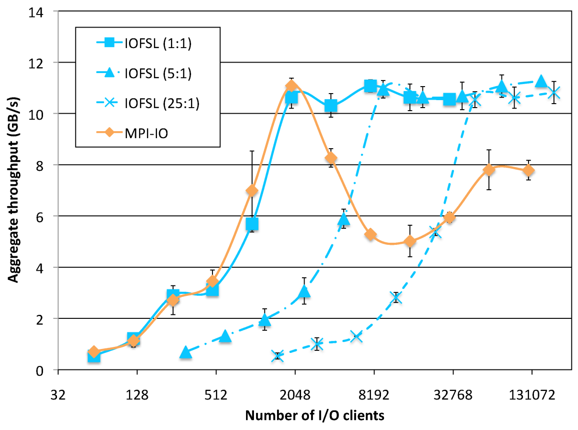

Fig. 9 IOR performance on JaguarPF using a shared-file I/O pattern with MPI-IO and IOFSL. In this experiment we kept the ratio of IOFSL client nodes to server nodes constant (indicated in the legend) up to a maximum of 160 IOFSL servers.

User-level access to I/O nodes or Lustre router nodes, which would be optimal locations for the deployment of IOFSL servers, is not possible on JaguarPF due to administrative policies. Therefore, additional compute nodes are allocated with each application launch and the IOFSL servers are spawned on these extra nodes, through which all application I/O requests are proxied. Figure 7 illustrates this deployment strategy. On JaguarPF, the BMI Portals driver is used to leverage the performance of the XT5 SeaStar 2+ interconnect.

5.1 IOFSL Performance on JaguarPF

For a general understanding of I/O performance on JaguarPF, we ran a series of experiments using the IOR benchmark [21] that measures the write throughput of bulk data transfers (4 MB). The results of these experiments are illustrated in Figures 8, 9, and 10. In these experiments, the native version used IOR's MPI-IO driver while the IOFSL tests employed a custom IOFSL driver that uses the ZOIDFS API. The results illustrated in Figure 8 depict the expected performance of IOFSL when increasing the number of servers in a shared-file I/O pattern. In general, IOFSL matches the observed native performance when using MPI-IO. The results in Figure 9 depict the performance of IOFSL when the number of IOFSL servers is limited to at most 160 and the number of clients to each IOFSL server varies when using a shared-file I/O pattern. We observe that IOFSL outperforms the native case at larger scales. We have identified two reasons for this improvement: the number of writers (IOFSL servers) is limited, and the I/O pattern and layout mitigated lock contention when forwarded through the IOFSL servers. The event-driven architecture of the IOFSL servers allows each server at the larger-scale experiments to accommodate 1,200 clients while providing roughly a total of 11 GB/s write throughput. The results illustrated in Figure 10 highlight the performance of IOFSL when using a file-per-process I/O pattern. IOFSL performance follows the general pattern observed in the native use case.

With the default IOFSL configuration (unbuffered I/O mode) and the IOR benchmark on JaguarPF when the system was in normal operation, 10.8 GB/s to 11.5 GB/s aggregate sustained bandwidth was observed when writing to a single shared file for 1,920 to 192,000 IOFSL clients and when using at most 160 IOFSL servers. Furthermore, an aggregate sustained bandwidth of 17.9 GB/s for 2,880 clients,

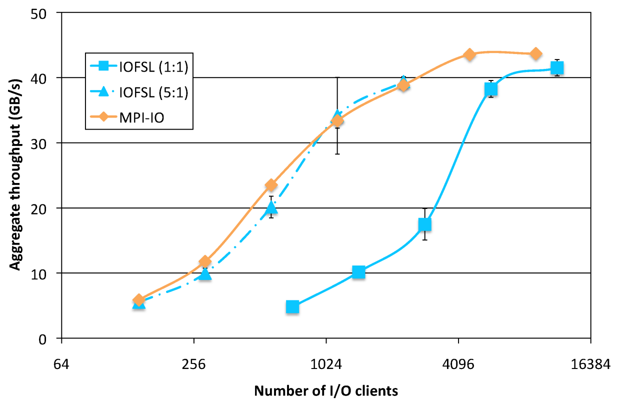

Fig. 10 IOR performance on JaguarPF using a file-per-process I/O pattern with MPI-IO and IOFSL. In this experiment we kept the ratio of IOFSL client nodes to server nodes constant.

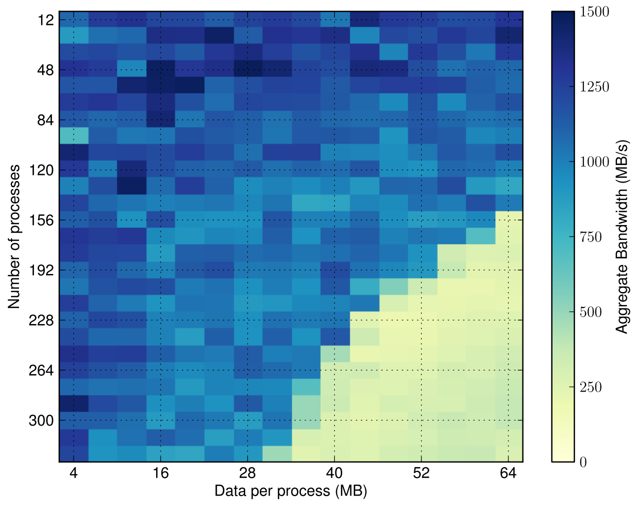

Fig. 11 IOFSL write buffering performance on Cray XT5 system (OLCF's JaguarPF). For this experiment, we used a single IOFSL server, 12 to 324 IOFSL clients, and 4 MB to 64 MB of data per IOFSL client.

39.2 GB/s for 5,760 clients, and 42.5 GB/s for 11,520 clients was when observed using 60 IOFSL clients per IOFSL server and when writing to unique files (one file per process).

To better understand the performance of IOFSL's new write buffering capability when compared with the original IOFSL synchronous write behavior, we measured the performance of the new capability using a modified version of the IOR benchmark that invoked write buffering operations. These experiments focused on identifying the I/O throughput observed by the application, and the results ignore the cost of application-initiated flushes. Figure 11 illustrates the performance on JaguarPF. This data clearly indicates that IOFSL can significantly accelerate the sustained storage system bandwidth perceived by the application when sufficient buffer space is available at the IOFSL server. The drop in performance at the bottom right corner of this figure occurs when the IOFSL server exceeds its write buffer space and blocks any additional buffered I/O operations until buffer space becomes available. Therefore, the usefulness of this capability is constrained by the size of write buffer available to an IOFSL server, the frequency of write buffering operations initiated by IOFSL clients, and the sustained bandwidth the IOFSL server can realize when transferring the buffered data to the storage system.

5.2 Recording traces of S3D with VampirTrace

To demonstrate tracing at large scale, we instrumented the petascale application S3D with VampirTrace. S3D is a parallel direct numerical simulation code developed at Sandia National Laboratories [11]. We used a problem set that scales to the full JaguarPF system. It uses weak scaling to allow a wide range of process counts, from 768 to 200,448. The processes are single-threaded, and we ran one process per core. In its role as an early petascale code, S3D is well understood and has been analyzed with TAU and Vampir at smaller scales [23]. The purpose of our experiment was to investigate the scaling of trace recording rather than an analysis of the application. S3D provides a real-world instrumentation target for the measurement environment. In addition, the large number of MPI messages generated by S3D creates a high frequency of events (approximately 7,700 events per second per process). Complete information about all messages is essential for the analysis of parallel applications, so these events should not be filtered. This provides a challenging workload for the measurement environment. We have traced 60 application time steps during our experiments using a basic online function filter. Further improvements of the instrumentation, such as selective function instrumentation or manually tracing a limited number of time steps, were deliberately not applied in order to keep the workload for the measurement environment high. Showing that the measurement environment handles this workload well suggests that it will also work for applications with lower number of MPI messages or more specific instrumentation. The synchronous flush feature in VampirTrace was used with a total of three flushes during the application execution in addition to the final flush during the application shutdown.

Prior to our successful demonstration, the largest scale trace for VampirTrace was approximately 40,000 processes using POSIX I/O. In practice, achieving this level of parallelism is already difficult because of substantial overhead during file generation and the impact on other users of the file system.

In our demonstration we utilized 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 file system. We have conducted multiple experiments tracing up to 200,448 application processes running S3D and using a set of 672 I/O forwarding nodes resulting in 2,688 files. The largest generated trace size was 4.2 TB of compressed data containing 941 billion events. The total time spent on trace I/O, including connection setup to the forwarding server, file creation, open, sync, and close, was 71 seconds with write buffering I/O, for a total application run time of 22 minutes. Trace I/O was synchronized among the MPI processes so this time includes the time spent in barriers waiting for other processes to complete their I/O operations. It therefore represents the total extension of application run time due to trace I/O. On average it took 5.5 seconds for each process to establish the connection to the forwarding server and to open the four shared output files (definitions file and events file, plus an index file for each). For some processes the connection setup took up to 32 seconds because of the massive stress on I/O forwarding nodes resulting from write operations from other processes. The intermediate buffer flushes are not affected by connection initialization, file open times, and final commit and therefore show significantly improved individual performance. With write buffering enabled, aggregated write rates of up to 154 GB/s or 33.5 billion events per second were observed, as recorded by the tracing measurement environment during individual flushes. We observed this high bandwidth because all trace data fit into the IOFSL servers' buffers. The time for the client to flush this data was limited by the IOFSL server performance. This bandwidth result also includes the synchronization of all processes as well as the overhead of OTF and trace data compression.

For comparison, we ran a full-size experiment using the IOFSL enhancements and unbuffered I/O. The total trace I/O time was 122 seconds, yielding a sustained aggregate bandwidth of 35.3 GB/s. This further indicates that write buffering reduces the I/O overhead observed by the tracing infrastructure. The IOFSL capability buffers trace data at the IOFSL server and overlaps application tracing with trace data storage. In this test series, the trace size per process remains almost constant.

The postprocessing (`vtunify`) for such a trace requires approximately 27 minutes but only a fraction of the resources of the application (10,752 workers). This is a required step, regardless of the use of IOFSL. However, IOFSL was not used because only 10,754 files[3] are created in this process.

This demonstration shows that full-size trace recording on large-scale systems can be done with a well-manageable overhead, even with trace I/O phases during the application execution. We investigated the scaling behavior of our solution with a series of experiments in different configurations. Figure 12 shows the total application run times at different scales with and without tracing. While the overhead of both trace I/O and tracing in general increases with the number of processors, it remains below 15% even at full scale.

[3] 10,752 event files, one definitions file, and one control file.

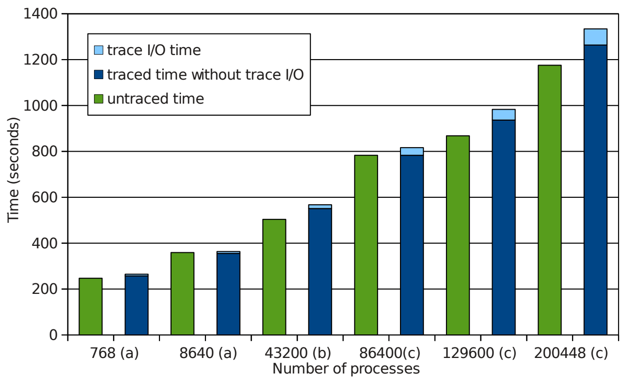

Fig. 12 Run times of S3D with and without tracing for different process counts: (a) average of 11 experiments, (b) average of 7 experiments, (c) single test run during dedicated reservation.

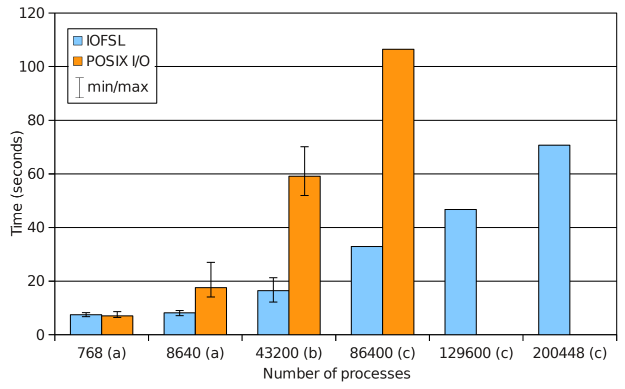

Fig. 13 Total trace I/O times for different process counts: (a) average of 11 experiments with min/max, (b) average of 7 experiments with min/max, (c) single test run during exclusive system reservation; no POSIX I/O data for 129600, 200448.

A comparison with POSIX I/O at different scales is presented in Figures 13 and 14. The POSIX I/O experiment with 86,400 processes was conducted during a dedicated system reservation. To avoid any potential impact to file system stability, we did not scale the POSIX I/O tests further. For all tests, the same software versions were used; hence, POSIX I/O tests also benefit from the improvements described in Section 4.2 that are not directly related to IOFSL. The POSIX I/O event rate is limited by the rate of file creation. This limitation is removed by utilizing IOFSL and allows to saturate the I/O bandwidth of the forwarding servers for 40k to 200k client processes. The impact of file creation depends on total time, which in turn depends on trace size per process; it will be even more dominant with lower numbers of events.

The experiments with lower process counts were run at different times during production use of the system. I/O in a shared system is always prone to variability, especially with a single metadata server being the bottleneck for any file metadata operation.

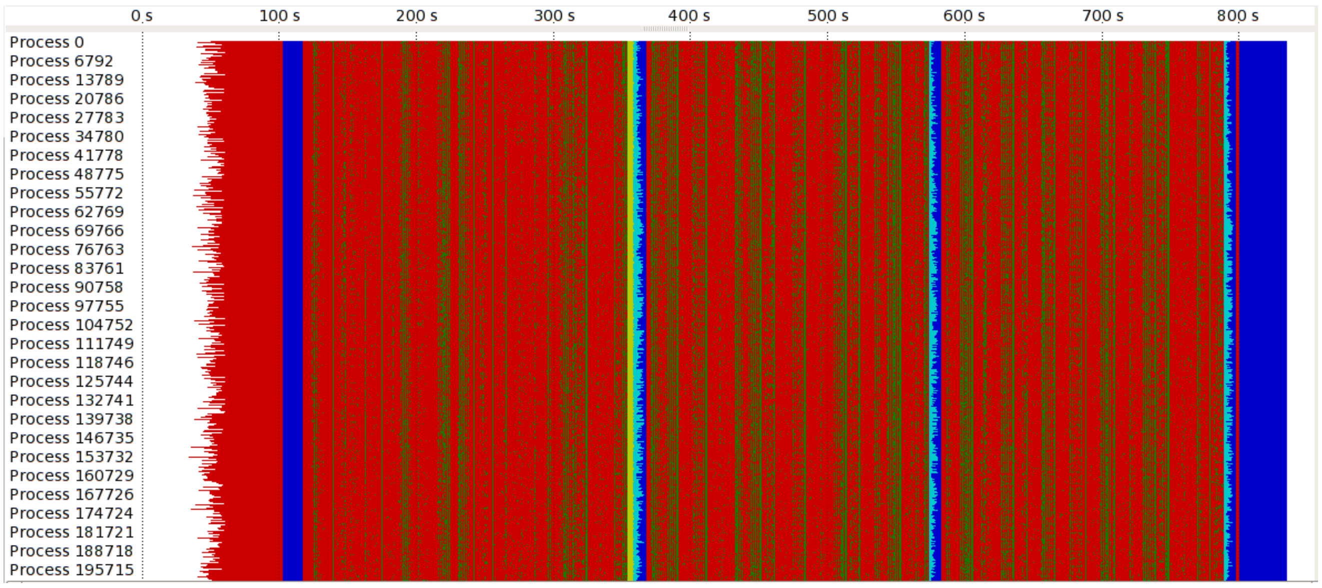

Fig. 15 Screenshot of Vampir visualizing a trace of the S3D application using 200,448 processes on JaguarPF. User functions are shown in green, MPI operations in red, and activities of the measurement environment in yellow (file open), light blue (trace I/O), and dark blue (synchronization).

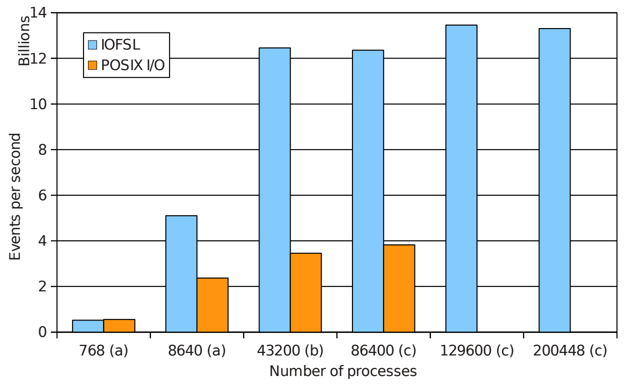

Fig. 14 Aggregate event write rates for different process counts. Averages used as in Figure 12. The data includes the overheads of establishing the connection, file open, synchronization, commit, and close.

The trace files generated with IOFSL were validated by using the post-mortem analysis tool Vampir. Vampir was able to read valid trace files and display detailed graphics of measured events versus a timeline and various other displays. Figure 15 shows a trace of S3D with 200,448 processes opened in Vampir, using 21,516 processes for interactive visualization. All processes are visible in an overview showing user functions in green, MPI operations in red, and phases in which the application was suspended by the measurement environment in blue. In addition, file open operations in the first flush are presented in yellow, followed by the actual trace I/O colored light blue as in subsequent flushes. Dark blue represents synchronization phases at the end of each flush. The three flushes take place around 360 s, 580 s, and 790 s of program run time. While being clearly visible, they are acceptably short in relation to the overall run time of the application. The figure also shows VampirTrace's internal time synchronizations that extend the first and the last synchronization phase after 100 s and 800 s. This total overview serves as a starting point to further investigate the details by zooming into both the time and process axes.

At such large scales, visualization for analysis purposes becomes more challenging as the ratio between available pixels and displayed processes decreases. Vampir's ability to smoothly zoom and scroll into both the process and time dimension helps navigation even in such large traces. However, new ways to highlight performance anomalies are required to help the user at those scales find the right spots to focus at. These topics are the subject of ongoing research; our solution lays a foundation for a comprehensive analysis at full scale by providing a feasible way to store event trace data.

5.3 Portability to Other Systems

We have also evaluated the integrated IOFSL and Vampir toolset on the Intrepid IBM Blue Gene/P (BG/P) large-scale computing system that is deployed at the Argonne Leadership Computing Facility (ALCF). The purpose of this evaluation was to demonstrate the portability of our solution to other large-scale computing platforms, runtime environments, storage systems, and applications.

Intrepid is a 557 teraflop IBM BG/P, consisting of 40,960 compute nodes and 640 I/O nodes. I/O nodes communicate via 10 Gigabit Ethernet with one another and with Intrepid's two high-performance storage systems: a 3 PB GPFS file system and a 512 TB PVFS file system. Intrepid's system administration policies permit users to customize the runtime environment of the system. For our evaluation of IOFSL and VampirTrace/OTF, these policies allowed us to deploy the IOFSL servers on the I/O nodes. We had to replace IBM's ciod I/O forwarding software with the ZOID [22] BG/P tree network driver in order to facilitate high-throughput and low-latency communication between user-space processes on the compute and I/O nodes, and boot the ZeptoOS [44] operating system on the BG/P compute nodes (replacing IBM's CNK). Additional information on the deployment of IOFSL on BG/P systems is provided in our prior work [33].

We successfully traced the Radix-k [36] image compositing algorithm on Intrepid at a variety of scales using the integrated IOFSL and VampirTrace/OTF toolset. Aside from small adjustments to the infrastructure deployment, the software stack required no additional changes to run on the system. Our experience on Intrepid demonstrates that we can trace additional applications, run our toolsets in different runtime environments and systems, and interact with different storage systems. Since our initial target platform was JaguarPF, assessing the performance and scalability of these tools on IBM BG/P systems is a work in progress.

6 Related Work

An early version of this work was presented at HPDC 2012 [20]. Major improvements in this extended version include an in-depth discussion of the architecture of IOFSL (Section 3) and new, extensive measurements of "raw" performance of IOFSL on JaguarPF (Section 5.1) to provide a context for subsequent results with VampirTrace; furthermore, the paper has been extened with a comparison to other established tracing frameworks and a discussion of new developments in standard IO APIs, specifically pNFS and the recently ratified MPI version 3.0, and how these might affect the work described in this paper (Section 6).

The performance analysis toolset Scalasca faced problems similar to ours when handling large numbers of trace files. Recently, the scalability of Scalasca was improved up to $300,000$ cores [43]. For tests on a large IBM BG/P system, the SIONlib library was used. It uses a special multifile format that contains additional metadata managing chunks of data from different processes within one file [15]. With SIONlib, multifile creation is a collective operation. This would pose a significant limitation to VampirTrace with respect to the dynamic threading model.

Instead of relying on function call instrumentation like VampirTrace, HPCToolkit employs a sampling strategy to generate call path profiles and traces [3]. While this reduces the amount of data generated and allows a rough estimation of the required storage space, it comes with its own sampling-related problems, for example, a loss of details for very short events. For storing the trace information, HPCToolkit also requires using at least one file per process. While this requirement does not constitute a problem for the maximum reported $8,184$ core executions [41], it will inevitably lead to similar problems for larger scales.

The TAU framework focuses on profiling, which produces only a fraction of the amount of data seen in tracing applications [38]. However, since TAU creates at least one file per process for storing the profiling data, it is also prone to the scalability issues related to metadata operations during the creation of files.

The POSIX I/O standard was designed before the advent of wide-scale parallelism. Hence, it suffers from many fundamental characteristics that preclude it from scenarios such as multiple writers updating the same file – a common need for parallel I/O oriented activity [19].

New I/O research efforts within standards-oriented activities have recognized this fact and are actively working on APIs appropriate for extreme-scale parallelism [19, 35]. One such API is pNFS [18], an extension to NFSv4 designed to overcome NFS scalability and performance barriers. Like IOFSL, it is based on a stateless protocol. However, it does not provide the "n-to-m" client to forwarding-server architecture fundamental to our design 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 pNFS for I/O accesses.

MPI-IO [28] provides a more sophisticated I/O abstraction than does POSIX. It includes collective operations and file views, which enable coordinated and concurrent access without locking [12]. It does not directly provide an "n-to-m" mapping from clients to output files. It does support distributed atomic append operations. For this capability to work, however, all processes that want to append must open the file using a collective, synchronizing operation, introducing unnatural synchronization into the application being traced.

In addition, the shared file pointer operations used to implement distributed atomic append functionality do not return the offset at which data has been written – information required to efficiently build OTF indices. MPI does offer a method to query the position of the shared file pointer; but as pointed out by Chaarawi et al. [10], doing so in the presence of concurrent updates creates a race condition. In order to resolve this race condition, a synchronizing collective would have to be used, making buffer flushes effectively collective.

While MPI-IO also supports asynchronous write operations, these differ from the write-buffering capability in IOFSL in that the operation returns before the data has been transferred off the node. Instead, the transfer continues in the background. Since this would perturb the application, this functionality cannot be used for event tracing.

The version 3 of the MPI standard (which includes MPI-IO) was recently released. Unfortunately, the updates to MPI-IO are mainly limited to the introduction of nonblocking collective I/O operations, leaving the issues highlighted in this section unresolved.

The I/O Delegate Cache System (IODC) [32] is a caching mechanism for MPI-IO that resolves cache coherence issues and alleviates the lock contention of I/O servers. IOFSL offers similar capabilities but is positioned 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 in the Sandia Cplant project [34], 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.

DataStager [2] and Decoupled and Asynchronous Remote Transfers (DART) [14] achieve high-performance transfers on Cray XT5 using dedicated data-staging nodes. Unlike our approach, which is transparent to the applications that use POSIX and MPI-IO interfaces, DART requires applications to use a custom API.

Similarly, Adaptable I/O System (ADIOS) [27] provides performance improvements through strategies such as prefetch and write-behind, based on application-specific configuration files read at startup; this information also helps ADIOS minimize the memory footprint during the course of the application run. In contrast, our approach requires no knowledge of the application behavior in advance and is situated at a lower level in the I/O software stack.

PLFS [5] 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, it interposes middleware between the client application and the underlying file system through the use of FUSE. The solution, aimed at checkpointing and similar activities for architectures such as Los Alamos National Laboratory's Roadrunner (3,060 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. Since 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. Furthermore, our IOFSL-based solution focuses on transforming uncoordinated file accesses to many unique files, such as a file-per-process I/O pattern, into a shared-file per group of processes I/O pattern. Our solution reduces file system resource contention generated by shared-file access patterns (such as file stripe lock contention or false sharing) and eliminates file system metadata overheads generated by I/O patterns with one file per process (such as frequent file creation or attribute access operations) at extreme scales.

IOFSL work extends the earlier ZOID efforts [22]. ZOID is a Blue Gene-specific function call forwarding infrastructure that is part of the ZeptoOS project. The I/O forwarding protocol used by IOFSL was first prototyped in ZeptoOS. IOFSL is a mature, portable implementation that integrates with common HPC file systems and also works on the Cray XT series and Linux clusters.

While recent work has addressed the use of nonblocking I/O at the I/O forwarding layer [42], our work focuses on providing a portable and transparent-to-applications, write-buffering-based, and high-performance nonblocking I/O capability in HPC environments. Furthermore, nonblocking file I/O capabilities are not provided by existing I/O forwarding solutions, including IBM's ciod or Cray's DVS.

In information technology in general, big data is a major new research topic. Technologies such as Apache Hadoop allow users to handle dozens of petabyes of data [7]. The challenge of big data is adjacent to HPC; however, scientific data is often tightly coupled and structured. Similarly to what we have proposed, big data uses augmentations to existing I/O software that take advantage of specific workload characteristics and have been shown effective in improving performance for important workloads. For example, the Google File System provides specialized append operations that allow many tasks to contribute to an output file in an uncoordinated manner [16].

7 Conclusions and Future Work

In this paper we have addressed the challenge of massively parallel I/O operations by utilizing I/O forwarding middleware. Tests on a large-scale system show that aggregated performance for a high number of parallel write operations can be improved by using I/O forwarding, when compared to MPI-IO. The I/O forwarding middleware IOFSL also provides a new atomic append capability that is the key to aggregating uncoordinated parallel data writes. Through an integration of IOFSL into the OTF library, the Vampir performance analysis toolset benefits from this capability. Additional improvements of performance result from utilizing write buffering, which, thanks to being implemented on separate I/O forwarding nodes, does not perturb the application processes.

In conclusion, it is now feasible to trace performance data of full-size application runs on large-scale systems (over 200,000 processes). Further scalability improvements of the Vampir toolset leverage the entire performance analysis workflow, including post-processing and visualization, for these large application traces.

While these results show, that I/O is no longer a severe bottleneck, there are still further lines of inquiry to be considered. Ongoing work investigates advanced filtering, selective tracing, and semantic runtime compression to provide additional benefits for tracing large application runs. We show that even at medium scales, tracing overhead can be significantly reduced with our solution. The benefit for scalability results from reducing the massive amount of metadata file system requests from all application processes to a much lower number.

We will pursue more advanced aggregate memory footprint optimizations to yield more available memory to user applications. While we have addressed the data collection challenges 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 will 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 event and trace data collections [29, 30].

The capabilities described in this paper are also applicable to other use cases beyond improving VampirTrace's I/O and can be implemented within other I/O forwarding tools. The new IOFSL capabilities can be used to improve the I/O performance of tools that generate per-process logs. Thus, these capabilities are applicable to massively parallel applications that exhibit loglike data storage patterns (such as Qbox's [17] shared file pointer object capability), data-intensive stream-processing tools (such as LOFAR's real-time signal-processing pipeline [37]), and high-level I/O libraries that allow unlimited dimensionality or enlargement of variable data structures (such as chunked data storage in HDF5 [31]). While we limited our demonstration of these capabilities to IOFSL, they are sufficiently generic and can be implemented in other production-quality I/O forwarding layers, such as IBM's ciod and Cray's DVS. If these capabilities are implemented in these production tools, they can substantially improve the HPC community's ability to understand applications running on large-scale systems.

Since the experiments were conducted, the OLCF and ALCF have upgraded their large-scale HPC computing resources. The new Titan Cray XK6 supercomputer at OLCF consists of 299,008 CPU cores and 18,688 GPUs; Mira, an 786,432 CPU core IBM Blue Gene/Q system, is now the leading system at ALCF. Both centers also upgraded the storage systems that serve their large-scale computing resources. While we are confident that our toolsets will scale on these systems, we will re-evaluate the scalability and performance of our tools on these new platforms as they are deployed. Moreover, we plan to further investigate the IOFSL and OTF/VampirTrace configuration space on these systems so that we can identify optimal infrastructure configurations for performance analysis I/O workloads.

Acknowledgements We thank Ramanan Sankaran (ORNL) for providing a working version of S3D as well as a benchmark problem set for JaguarPF. We are grateful to Matthias Jurenz for his assistance on VampirTrace as well as Matthias Weber and Ronald Geisler for their support for Vampir. 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. This work was supported in part by the National Science Foundation (NSF) through NSF-0937928 and NSF-0724599. This work is supported in a part by the German Research Foundation (DFG) in the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing".

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

References

1. SYSIO. `http://sourceforge.net/projects/libsysio`.
2. Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F. DataStager: Scalable data staging services for petascale applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC)* (2009), pp. 39–48.
3. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurr. Comput. : Pract. Exper. 22*, 6 (Apr. 2010), 685–701.
4. Ali, N., Carns, P., Iskra, K., Kimpe, D., Lang, S., Latham, R., Ross, R., Ward, L., and Sadayappan, P. Scalable I/O forwarding framework for high-performance computing systems. In *Proceedings of the 11th IEEE International Conference on Cluster Computing (CLUSTER)* (2009).
5. Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. PLFS: A checkpoint filesystem for parallel applications. In *Proceedings of 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2009).
6. Bland, A., Kendall, R., Kothe, D., Rogers, J., and Shipman, G. Jaguar: The world's most powerful computer. In *Proceedings of the 51st Cray User Group Meeting (CUG)* (2009).
7. Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., and Aiyer, A. Apache Hadoop goes realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), pp. 1071–1080.
8. Carns, P., Ligon III, W., Ross, R., and Wyckoff, P. 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)* (2005).
9. Carns, P. H., Ligon III, W. B., Ross, R. B., and Thakur, R. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference (ALS)* (2000), pp. 317–327.

10. Chaarawi, M., Dinan, J., and Kimpe, D. On the usability of the MPI shared file pointer routines. In *Proceedings of the 19th European MPI Users' Group Meeting (EuroMPI)* (2012).
11. Chen, J. H., Choudhary, A., de Supinski, B., DeVries, M., Hawkes, E. R., Klasky, S., Liao, W. K., Ma, K. L., Mellor-Crummey, J., Podhorszki, N., Sankaran, R., Shende, S., and Yoo, C. S. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery 2*, 1 (2009), 015001.
12. Ching, A., Choudhary, A., Coloma, K., Liao, W., Ross, R., and Gropp, W. Noncontiguous I/O access through MPI-IO. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)* (2003), pp. 104–111.
13. Cope, J., Iskra, K., Kimpe, D., and Ross, R. Bridging HPC and Grid file I/O with IOFSL. In *Proceedings of the Workshop on State of the Art in Scientific and Parallel Computing (PARA'10)* (2011).
14. Docan, C., Parashar, M., and Klasky, S. DART: A substrate for high speed asynchronous data IO. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC)* (2008).
15. Frings, W., Wolf, F., and Petkov, V. Scalable massively parallel I/O to task-local files. In *Proceedings of 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2009).
16. Ghemawat, S., Gobioff, H., and Leung, S. The Google File System. *SIGOPS Operating Systems Review 37* (Oct. 2003), 29–43.
17. Gygi, F., Duchemin, I., Donadio, D., and Galli, G. Practical algorithms to facilitate large-scale first-principles molecular dynamics. *Journal of Physics: Conference Series 180*, 1 (2009).
18. Hildebrand, D., and Honeyman, P. 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)* (2005), pp. 18–27.
19. IEEE POSIX Standard 1003.1 2004 Edition. `http://www.opengroup.org/onlinepubs/000095399/functions/write.html`.
20. Ilsche, T., Schuchart, J., Cope, J., Kimpe, D., Jones, T., Knüpfer, A., Iskra, K., Ross, R., Nagel, W. E., and Poole, S. Enabling event tracing at leadership-class scale through I/O forwarding middleware. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2012), HPDC '12, ACM, pp. 49–60.
21. IOR HPC Benchmark. `http://sourceforge.net/projects/ior-sio/`.
22. Iskra, K., Romein, J. W., Yoshii, K., and Beckman, P. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2008), pp. 153–162.
23. Jagode, H., Dongarra, J., Alam, S., Vetter, J., Spear, W., and Malony, A. D. A holistic approach for performance measurement and analysis for petascale applications. In *Proceedings of the 9th International Conference on Computational Science (ICCS)* (2009), vol. 2, pp. 686–695.
24. Jones, T., Dawson, S., Neely, R., Tuel, W., Brenner, L., Fier, J., Blackmore, R., Caffrey, P., and Maskell, B. Improving the scalability of parallel jobs by adding parallel awareness. In *Proceedings of the 15th ACM/IEEE International Conference on High Performance Networking and Computing (SC)* (2003).
25. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M. S., and Nagel, W. E. The Vampir performance analysis tool-set. In *Tools for High Performance Computing* (July 2008), M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds., Springer Verlag, pp. 139–155.
26. Liu, N., Cope, J., Carns, P. H., Carothers, C. D., Ross, R. B., Grider, G., Crume, A., and Maltzahn, C. On the role of burst buffers in leadership-class storage systems. In *MSST* (2012), IEEE.
27. Lofstead, J. F., Klasky, S., Schwan, K., Podhorszki, N., and Jin, C. 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)* (2008), pp. 15–24.
28. MPI Forum. MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/docs.html`, 1997.
29. Muelder, C., Gygi, F., and Ma, K.-L. Visual analysis of inter-process communication for large-scale parallel computing. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1129–1136.
30. Muelder, C., Sigovan, C., Ma, K.-L., Cope, J., Lang, S., Iskra, K., Beckman, P., and Ross, R. Visual analysis of I/O system behavior for high-end computing. In *Proceedings of the 3rd International Workshop on Large-Scale System and Application Performance (LSAP)* (2011).
31. NCSA. HDF5. `http://hdf.ncsa.uiuc.edu/HDF5/`.
32. Nisar, A., Liao, W., and Choudhary, A. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of 20th ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2008).
33. Ohta, K., Kimpe, D., Cope, J., Iskra, K., Ross, R., and Ishikawa, Y. Optimization techniques at the I/O forwarding layer. In *Proceedings of the 12th IEEE International Conference on Cluster Computing (CLUSTER)* (2010).
34. Pedretti, K., Brightwell, R., and Williams, J. Cplant^{TM} runtime system support for multi-processor and heterogeneous compute nodes. In *Proceedings of the 4th IEEE International Conference on Cluster Computing (CLUSTER)* (2002), pp. 207–214.
35. Petascale Data Storage Institute. `http://www.pdsi-scidac.org/`.
36. Peterka, T., Goodell, D., Ross, R., Shen, H.-W., and Thakur, R. A configurable algorithm for parallel image-compositing applications. In *Proceedings of 21st ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2009).
37. Romein, J. FCNP: Fast I/O on the Blue Gene/P. In *Parallel and Distributed Processing Techniques and Applications (PDPTA)* (2009).
38. Shende, S. S., and Malony, A. D. The TAU parallel performance system. *Int. J. High Perform. Comput. Appl. 20*, 2 (May 2006), 287–311.
39. Shipman, G., Dillow, D., Oral, S., and Wang, F. The Spider center wide file system; from concept to reality. In *Proceedings of the 51st Cray User Group Meeting (CUG)* (2009).
40. Snyder, P. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 European UNIX Users Group Conference* (1990), pp. 241–248.
41. Tallent, N. R., Mellor-Crummey, J., Franco, M., Landrum, R., and Adhianto, L. Scalable fine-grained call path tracing. In *Proceedings of the international conference on Supercomputing* (New York, NY, USA, 2011), ICS '11, ACM, pp. 63–74.
42. Vishwanath, V., Hereld, M., Iskra, K., Kimpe, D., Morozov, V., Papka, M., Ross, R., and Yoshii, K. Accelerating I/O forwarding in IBM Blue Gene/P systems. In *Proceedings of 22nd ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2010).

43. Wylie, B. J. N., Geimer, M., Mohr, B., Böhme, D., Szebenyi, Z., and Wolf, F. Large-scale performance analysis of Sweep3D with the Scalasca toolset. *Parallel Processing Letters 20*, 4 (2010), 397–414.
44. Yoshii, K., Iskra, K., Naik, H., Beckman, P., and Broekema, P. C. Performance and scalability evaluation of "Big Memory" on Blue Gene Linux. *International Journal of High Performance Computing Applications 25*, 2 (2011), 148–160.
45. Yu, H., Sahoo, R. K., Howson, C., Almási, G., Castaños, J. G., Gupta, M., Moreira, J. E., Parker, J. J., Engelsiepen, T. E., Ross, R. B., Thakur, R., Latham, R., and Gropp, W. D. High performance file I/O for the Blue Gene/L supercomputer. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)* (2006), pp. 187–196.