CLARISSE: a middleware for data-staging coordination and control on large-scale HPC platforms.

Florin Isaila and Jesus Carretero
University Carlos III (Spain)
Email: {fisaila, jcarrete}@arcos.inf.uc3m.es

Rob Ross
Argonne National Laboratory (USA)
Email: rross@mcs.anl.gov

***Abstract*—On current large-scale HPC platforms the data path from compute nodes to final storage passes through several networks interconnecting a distributed hierarchy of nodes serving as compute nodes, I/O nodes, and file system servers. Although applications compete for resources at various system levels, the current system software offers no mechanisms for globally coordinating the data flow for attaining optimal resource usage and for reacting to overload or interference.**

In this paper we describe CLARISSE, a middleware designed to enhance data-staging coordination and control in the HPC software storage I/O stack. CLARISSE exposes the parallel data flows to a higher-level hierarchy of controllers, thereby opening up the possibility of developing novel cross-layer optimizations, based on the run-time information. To the best of our knowledge, CLARISSE is the first middleware that decouples the policy, control, and data layers of the software I/O stack in order to simplify the task of globally coordinating the data staging on large-scale HPC platforms. To demonstrate how CLARISSE can be used for performance enhancement, we present two case studies: an elastic load-aware collective I/O and a cross-application parallel I/O scheduling policy. The evaluation illustrates how coordination can bring a significant performance benefit with low overheads by adapting to load conditions and interference.

***Index Terms*—HPC; storage; data staging; parallel I/O; collective I/O; I/O scheduling;**

I. Introduction

The past several years have brought a significant growth in the amount of data generated in scientific domains such as astrophysics, climate, high-energy physics, biology, and medicine. Managing this data profligacy on large-scale HPC platforms requires a sustained effort in both hardware and software development. One of the most critical challenges is understanding the limitations of the storage I/O software stack in petascale systems and proposing novel solutions to address these limitations for larger data sets and larger scale [1], [2], [3].

The software I/O stack employed by many simulations on todays HPC platforms, shown in Figure 1, consists of scientific libraries (e.g., HDF5, parallel NetCDF), middleware (e.g., MPI-IO), I/O forwarding (e.g., IOFSL), and file systems (e.g., GPFS, Lustre). Scaling this I/O stack is challenging because the functionality involved in storage access is distributed over several types of nodes (compute nodes, I/O nodes, file system servers). Additionally, the current uncoordinated development model of independently applying optimizations at each layer of the system software I/O software stack is not expected to scale to the new levels of concurrency, storage hierarchy, and capacity [3]. Radically new approaches to reforming the I/O software stack are needed in order to enable holistic system software optimizations that can address cross-cutting issues such as performance, resiliency, and power.

The main contribution of this paper is CLARISSE, a middleware designed to improve the scalability of the software I/O stack on large-scale HPC infrastructures. To the best of our knowledge, CLARISSE is the first middleware that decouples the policy, control, and data layers of software I/O stack in order to simplify the task of globally coordinating the data staging on large-scale HPC platforms. CLARISSE exposes the parallel data flows from a supercomputer to a higher-level hierarchy of controllers, thereby opening up the possibility of developing novel cross-layer optimizations, based on the run-time information. In comparison, today's MPI-IO implementation does not use information such as network and server load in order to adapt the data paths for avoiding congestion or ensuring resilience. This paper proposes a novel model for building global control that can be used for designing system-wide data staging optimizations. To demonstrate how CLARISSE can be used for performance enhancement, we present two novel implementations of an elastic load-aware collective I/O and of a parallel I/O scheduling policy.

The remainder of the paper is organized as follows. Section II presents an overview of CLARISSE. Section III discusses the design and implementation of the CLARISSE middleware. Section IV presents two CLARISSE applications: an elastic

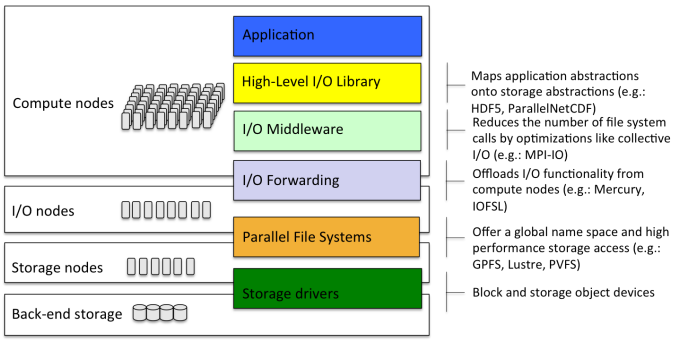


Fig. 1. Mapping of the I/O software stack (right-hand side) on the architecture of current large-scale HPC systems (left-hand side).

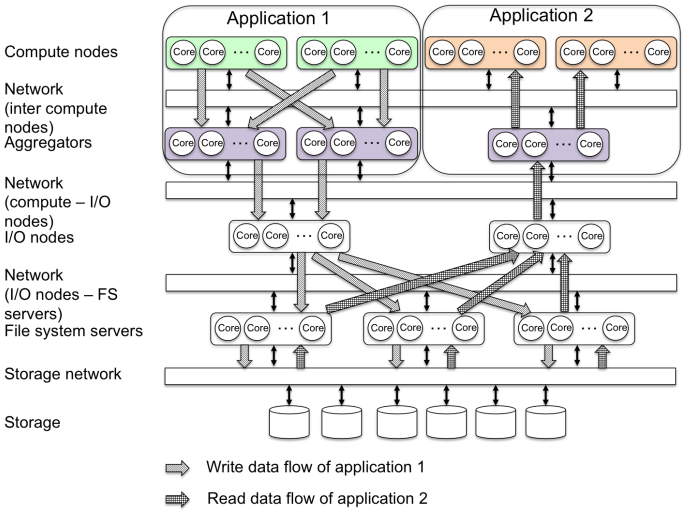


Fig. 2. Data path of two applications in current large-scale platforms.

collective I/O implementation and parallel I/O scheduling. Section V presents the experimental results. Section VI compares and contrasts CLARISSE with related work. Section VII presents our conclusions and discusses current and future work.

II. Overview

On current large-scale HPC platforms such as Blue Gene/Q or most Cray systems, the data path from compute nodes to final storage passes through several networks interconnecting a distributed hierarchy of nodes serving as compute nodes, I/O nodes, and file system servers. Figure 2 shows two parallel applications writing and reading from the external storage. Despite the fact that these applications compete for resources at various system levels, the current system software offers no mechanisms for globally coordinating the data flow for optimal resource usage and for reacting to overload or interference.

The main goal of the CLARISSE middleware is to offer runtime cross-layer coordination of data staging on large-scale HPC platforms. In order to achieve this goal,, the middleware functionality is separated into a control plane, data plane, and policy layer in a fashion similar to that in the software defined networking approach [4], as shown in Figure 3. The data plane includes mechanisms for transferring the data from the compute nodes to the storage nodes either collectively or independently and allows the building of data flows such as those in Figure 2. The control backplane offers mechanisms for coordinating and controlling the data staging based on a publish/subscribe API. Using control backplane mechanisms, one can implement various policies for controlling the data plane for aspects such as elastic collective I/O, parallel I/O scheduling, load balancing, resilience, and routing.

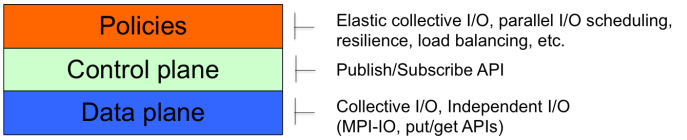


Fig. 3. Separation of data, control, and policy in CLARISSE:

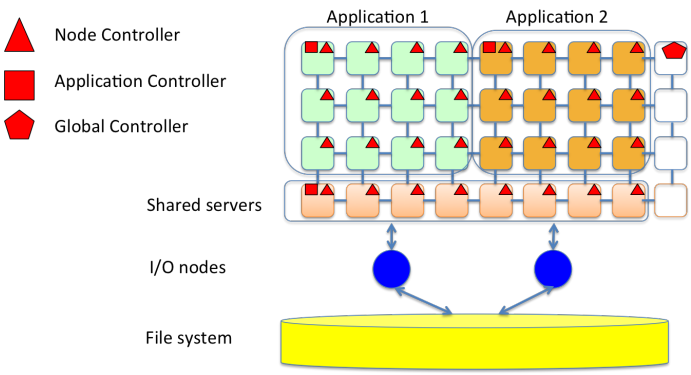


Fig. 4. Control backplane example. Node controllers reside on all compute nodes. Each application has an application controller. A global controller is used for systemwide coordination.

III. Design and Implementation

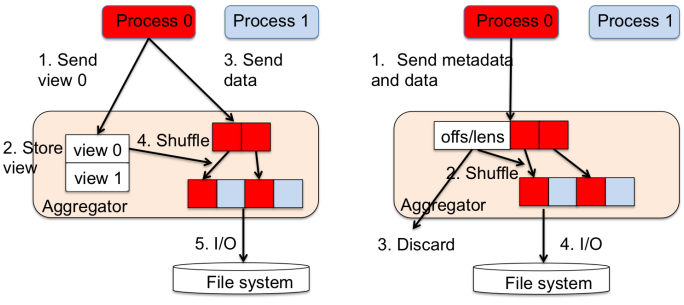
This section discusses the design and implementation of the CLARISSE middleware. The section is organized in three parts corresponding to data, control, and policy layers.

A. Data plane

The CLARISSE data plane is responsible for staging data between applications and storage or between applications. Applications can access data using a put/get interface or an MPI-IO interface. If the data source/destination is storage, the current design assumes the existence of a global file name space. For interapplication communication, data can be exchanged through a virtual name space. The two applications need only to agree on the name of the data set to be exchanged. The data exchange is performed through a shared data space (e.g., virtual file), which CLARISSE data plane maps on the data model of each application through MPI data types or offset-length lists.

For both put/get and MPI-IO interfaces, there are independent and collective I/O versions of the data access functions. The main difference between the two is that collective I/O involves the merging of small requests from several processes into larger ones in order to reduce I/O contention. The merging process is performed at processes called *aggregators*. Since high-performance scalable I/O for either network transfer or file system access involves some form of aggregation, we will focus in the remainder of this section on the collective I/O operations.

CLARISSE currently offers two collective I/O implementations: view-based I/O (see Figure 5a) and list-based I/O



a. View-based collective I/O b. List-based collective I/O

Fig. 5. Collective I/O implementations in CLARISSE: (a) view-based collective I/O; (b) list-based collective I/O.

(see Figure 5b). View-based I/O was described in detail elsewhere [6]. In summary a file is mapped on processes based on views, which are contiguous windows mapped on noncontiguous file regions. The views are either sent once to the aggregators and saved there for future use or sent with each access request. View-based I/O works well for small and moderately fragmented files. List-based I/O is a new collective I/O implementation designed for highly fragmented files, for addressing the high memory footprint of view-based I/O in these cases. Instead of transferring full views, list-based I/O optimally packs in a network buffer the maximum amount of data and pairs of offset-lengths representing the mapping of the access patterns to file. Unlike views, the offset-lengths pairs are ephemeral: they are discarded by the aggregators as soon as the data are aggregated into the collective buffers.

The data access operations described above can work in the absence of a control plane, for instance in the same way as the collective I/O operations from any existing MPI distribution. However, the main strength of our approach and difference from other work is that CLARISSE operations are controllable through the actions of the control plane, which is the subject of the next section. This approach opens up the possibility of implementing a large range of policies addressing, for instance, load and failure conditions at various places in the data-staging flow.

B. Control plane

The CLARISSE control backplane acts as a coordination framework designed to support the global improvement of key aspects of data staging, including load balancing, I/O scheduling, and resilience. The control backplane is constructed from a set of minimally invasive control agents. A control agent runs on every node and monitors the occurrence of predefined or dynamically defined events and reactively executes the associated actions.

The control plane offers the following classical publish/-subscribe API [7] that can be used for implementing control policies:

- `subscribe(event_properties)` registers for an event, which could either trigger a callback or be placed in an event queue on the calling node.
- `publish(event_type, event_data)` publishes an event, which causes the event data to be forwarded to the subscribed nodes.
- `event_data wait(event_type)` blocks waiting for the event to be received and returns the event data.
- `int test(event_type)` nonblockingly tests whether the event has been received and, if it has been received, returns the event data, otherwise NULL.

Figure 4 shows an example of a hierarchical control infrastructure, which has been implemented for this work. In this example a control agent can play the role of a node controller, an application controller, or a global controller. Control policies are implemented in an event-driven manner by the interaction among these types of controllers. The node controller is responsible for the node-level events associated with application or server processes (e.g., server load). An application controller is in charge of monitoring the nodes where the application is running and detect events related to any individual node (e.g., server node failure). A global controller monitors running applications and can take system-level decisions (e.g., I/O scheduling).

C. Policy layer

In this section we describe the steps involved in the development of a CLARISSE policy. These steps do not necessary have to be applied in the order described below. Rather, the implementation should be iteratively refined until a satisfactory result is obtained. In the next section we show how these steps are implemented for two policy examples.

First, the developer of a control policy has to identify relevant *control variables*, a set of variables to be used for implementing the policy. These can be existing variables that control the data flow or new variables. For instance, a file can be declustered over a set of servers represented by a server map. An existing server map can be chosen as a control variable if the control policy acts upon it and thereby changes the data flow. The identifier of a loaded server dynamically discovered is an example of a new variable introduced for implementing a policy.

Second, the developer has to decide on the proper place for inserting *control points* in the logic of the data staging implementation. A codesign of data staging and control algorithms can produce more efficient implementations. However, one can also add control to any existing data staging implementation.

Third, the developer needs to identify the distributed *entities* involved in the control algorithm. For instance, these can be the controller processes (e.g., node, application or global controllers), an external performance or fault monitor, or the processes of an end-user application.

The *control orchestration* is implemented through control actions using the control plane API described in Section III-B or other operations of the entities involved in control. The other operations can be, for example, communication operations specific to the platform where CLARISSE is deployed. The control actions from this step are placed only at the control points. Examples of control actions include waiting for an event to occur, querying the system state, and generating an event.

D. Status

A prototype of the CLARISSE middleware has been implemented in approximately 25K lines of C code [1]. In the current version the communication is MPI-based. The data plane collective I/O methods from Figure 5, view-based and list-based collective I/O can be used though put/get and MPI-IO interfaces. The control plane from Figure 4 has been implemented as as a publish/subscribe layer in MPI. The policies described in the following sections also have been fully implemented, and their evaluation is discussed in Section V.

[1]The code is available for download at https://bitbucket.org/fisaila/clarisse.

	Elastic Collective I/O	**Parallel I/O Scheduling**
Control variables	Server map Epoch Loaded server	Waiting queue
Control points	Before data shuffle	Before data shuffle After data shuffle
Entities	Application processes Performance monitor Controllers	Application processes Controllers
Orchestration	Figure 6	Figure 7

TABLE I
STEPS INVOLVED IN THE DEVELOPMENT OF TWO CLARISSE APPLICATIONS.

IV. CLARISSE POLICIES

This section illustrates the types of policies that CLARISSE enables. We discuss two policies: an elastic collective I/O and a parallel I/O scheduling policy. Table 1 shows the information relevant to the steps involved in developing a new CLARISSE policy as discussed in Section III-C. The details are discussed below.

A. Elastic collective I/O

The current collective I/O implementation from ROMIO, two-phase I/O [8] does not leverage information such as load or faults in order to adapt to run-time conditions and improve performance or avoid failures. In this section we present the implementation of a collective I/O operation that adapts to the load conditions at data aggregation servers (aggregators) in order to improve the performance. In particular, this implementation leverages CLARISSE control for dynamically removing a loaded server from the data path and continuing operation.

The implementation of elastic collective I/O uses three control variables. The first variable is the *server map*, the list of servers that are used for aggregating small file system requests into larger ones (as discussed in Section III-A). A newly defined variable is used for identifying a currently loaded server. A newly defined *epoch* is the time interval in which the server map value does not change. For instance, after the removal of a server from the system map has been disseminated to all application processes, a new epoch starts.

We use one control point for each collective I/O operation called by an application process before the data shuffling starts (see Figure 5).

Three entities are involved in control: end-user application processes calling the collective I/O operations, a performance monitor providing run-time information about system load, and the controllers managing the run-time event handling.

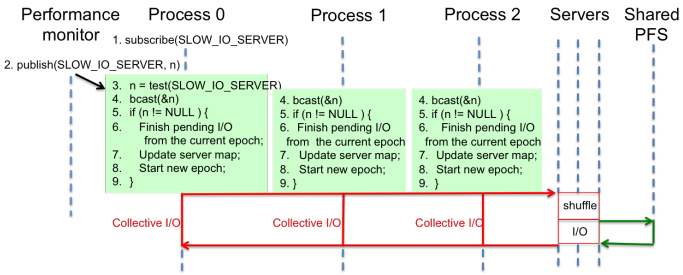


Fig. 6. Elastic collective I/O protocol. The green rectangles represent control points.

Figure 6 shows the control orchestration used by the elastic collective I/O method. For simplicity we do not show the controllers that are in charge of providing the publish/subscribe infrastructure. Our implementation assumes the existence of a large-scale system on-line performance monitor that detects a loaded server based on some criteria and publishes a message about it. The implementation of such a performance monitor is a complex task in itself [9] and is outside the scope of our work.

Our implementation dynamically removes a loaded server from a data-staging flow. Beginning on one node, the controller subscribes to `SLOW_IO_SERVER` events (step 1). Whenever the performance monitor detects a slow server, it publishes a `SLOW_IO_SERVER` event (step 2). When reaching a control point, the subscribed application process checks for the arrival of a `SLOW_IO_SERVER` event (step 3). Subsequently, a broadcast operation with the semantics of an MPI blocking collective operation [10] is used for broadcasting the loaded server (or none) and for enforcing synchronization between the processes participating in the collective I/O (step 4). This operation ensures that either none or all processes receive the information about a loaded server. If there is a loaded server (step 5), all processes finish pending independent I/O operations (step 6), update the server map by removing the loaded server (step 7), and start a new epoch (step 8).

The policy discussed in this section dynamically removes one server from the server map. A similar protocol can be used for adding a new server to the server map. A slightly more complex protocol can be used for adding/removing several servers in the same control iteration. A similar approach can be used for removing a server that failed. However, additional actions need to be taken into consideration for ensuring the correctness of data such as restarting of collective operations that are in progress. The implementation of these type of more complex policies is the subject of future work.

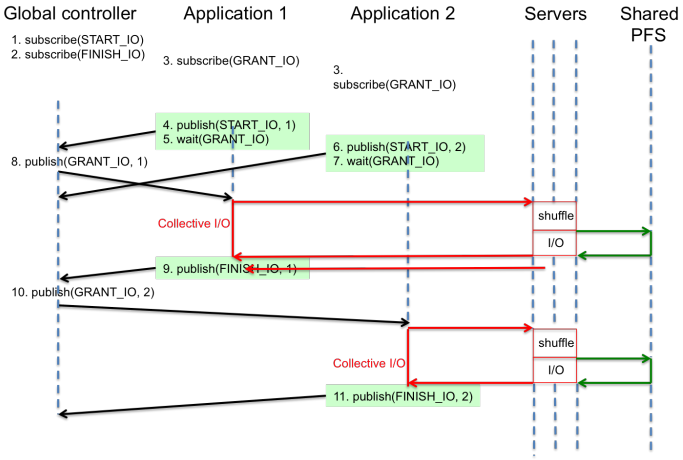


Fig. 7. FCFS parallel I/O scheduling. The green rectangles represent control points.

B. Parallel I/O scheduling

The flexible implementation of parallel I/O scheduling policies between applications for data accesses to shared resources is also a capability that is notably missing from the current software I/O stack, despite the fact that the benefits of such capability have been studied and empirically demonstrated [11]. CLARISSE enables the implementation of a large spectrum of parallel I/O scheduling policies. In this paper, we present an example of a simple policy implementation for collective I/O operations. An extensive study of parallel I/O scheduling policies is beyond the scope of this paper.

In this example we address a simple instance of the parallel I/O scheduling in Figure 4. Assume that two parallel applications are concurrently issuing collective I/O requests that involve the same set of aggregators at the same time. The lack of a parallel I/O scheduling strategy may cause server contention and substantially impact the performance of the parallel applications.

The control policy for first-come first-served (FCFS) requires a waiting queue as a control variable. The control points are before and after the data shuffling. The control policy involves application processes and controllers. The control orchestration is shown in Figure 7. A global controller subscribes to `START_IO` and `FINISH_IO` events (steps 1 and 2), and the application controllers subscribe to `GRANT_IO` events (step 3). Application 1 publishes a `START_IO` event containing a system-wide unique application identifier (step 4) and blocks waiting for a `GRANT_IO` event (step 5). Application 2 does the same (steps 6 and 7). The global scheduler first receives a `START_IO` event from application 1 and, given that no other application is currently scheduled, publishes a `GRANT_IO` event for the application identifier (step 8). Subsequently, the global scheduler receives the `START_IO` event from application 2 and saves it in the waiting queue, given that the application 1 has been scheduled. The application controller receives the `GRANT_IO` event, executes the collective I/O operation, and publishes a `FINISH_IO` event (step 9). The global controller receives this event and schedules the next application by retrieving the next application from the waiting queue and publishing a `GRANT_IO` event (step 10). Application 2 receives this event, schedules the shuffle operation, and publishes a `FINISH_IO` event (step 11).

This FCFS implementation schedules the access to aggregators. More complex time-sharing and space-sharing policies and multistage scheduling of aggregators and file system access are the subject of future work.

V. Experimental results

In this section we present an evaluation of the two CLARISSE policies proposed in this paper: elastic collective I/O and parallel I/O scheduling. We target the following questions: What is the performance benefit of these policies compared with the case when they are not used? What is the cost incurred by CLARISSE? Do the benefits outweigh the costs?

This section first describes the experimental setup and then presents the results.

A. Experimental setup

The experiments for our study are run on the Vesta BG/Qsupercomputer at Argonne National Laboratory. Vesta has 2,048 compute nodes (4 racks of 512 compute nodes each) with PowerPC A2 cores (1.6 GHz, 16 cores/node, and 16 GB RAM). The compute nodes are interconnected in a 5D torus network and do not have persistent storage. Each compute node has 11 network links of 2 GB/s and can concurrently receive/send an aggregate bandwidth of 44 GB/s. While 10 of these links are used by the torus interconnect, the 11th link provides connection to the I/O nodes. On Vesta, a set of 32 compute nodes (known as a pset) has one I/O node acting as an I/O proxy. For every I/O node there are two network links of 2 GB/s toward two distinct compute nodes acting as bridges. Therefore, for every 128-node partition, there are $n_b = 4 \times 2 = 8$ bridges. The I/O traffic from compute nodes passes through these bridge nodes on the way to the I/O node. The I/O nodes are connected to the storage servers through Quad-data-rate (QDR) InfiniBand links. The file system on Vesta is GPFS 3.5. The data are stored on 40 NSD SATA drives with a 250 MB/s maximum throughput per disk; the block size is 8 MB. The file system blocks are distributed by GPFS in a round-robin fashion over several NSDs, with the goal of balancing the space utilization of all system NSDs. The I/O nodes are file system clients, and the size of the client cache on each I/O node is 4 GB. The MPI distribution used in all experiments is MPICH 3.1.4.

In all experiments all the clients write to a shared file using list-based collective I/O (described in Section III-A). The ratio of the number of aggregators to number of clients was chosen based on the default ratio in the MPI-IO driver for GPFS, roughly 16:1 with the constraint of having a power of 2 number of total cores. The aggregators were placed on the Blue Gene/Q topology on the nodes close to the bridge nodes with a placement policy similar to the one from the MPI-IO driver for GPFS: first, aggregators were placed on nodes one hop away from the bridge nodes, followed by nodes two-hops away from the bridge nodes, and so on until the desired number of aggregators was reached. For making a reasonable use of resources, we always used the maximum number of cores of a batch scheduler allocation. For instance, for 128 nodes, we run an experiment with 128 x 16 = 2048 processes, of which 128 were aggregators; that is, 2048 - 128 = 1920 processes were dedicated to the applications. This explains the lack of powers of 2 in the number of processes in the experiments.

In our evaluation we used a self-crafted version of the IOR benchmark and two application kernels (VPICIO and VORPALIO). The IOR benchmark is one of the most popular parallel I/O benchmarks [12]. Our self-crafted version of the IOR benchmark (which we will call S-IOR in the remainder of the paper) generates the same pattern as the original IOR benchmark with the following differences meant to better

No. of Client Processes	No. of Server Processes	Access Size /Process	File Size
1920	128	16 MB	300 GB
3840	256	8 MB	300 GB
7680	512	4 MB	300 GB
15760	1024	2 MB	300 GB

TABLE II
BENCHMARK PARAMETERS.

reproduce the behavior of a significant class of real applications: it allows insertion of a pseudo-computation between two consecutive I/O operations and execution of a number of phases with consecutive I/O operations instead of repetitions of the same operation. For S-IOR we used the MPI-IO interface.

VPICIO and VORPALIO are two I/O kernels extracted from real scalable applications at LBL [13]. Both of these I/O kernels perform storage I/O through the H5Part library, which can store and access time-varying, multivariate data sets through the HDF5 library. For collective I/O the HDF5 library employs MPI-IO. In this evaluation we use the implementation of MPI-IO on top of CLARISSE.

VPICIO is an I/O kernel of VPIC, a scalable 3D electromagnetic relativistic kinetic plasma simulation developed by Los Alamos National Laboratory [14]. VPICIO receives as parameters the number of particles and a file name, generates a 1D array of particles, and writes them to a file. VPICIO was extended by us to write the array over a number of time steps.

VORPALIO is an I/O kernel of VORPAL, a parallel code simulating the dynamics of electromagnetic systems and plasmas [15]. The relevant parameters of VORPALIO are 3D block dimensions (x, y, and z), a 3D decomposition over p processes (p_x, p_y, and p_z where $p_x \times p_y \times p_z = p$), and the number of time steps. In each step VORPALIO creates a 3D partition of blocks and writes it to a file.

B. Elastic collective I/O

For the elastic collective I/O implementation we first evaluate the S-IOR benchmark in more detail and then present the results for VPICIO and VOPRALIO.

S-IOR is run as one application in the model shown in Figure 4 with four configurations shown in Table 2. The total data written to the file system in all cases is 30 GB/operation (strong scaling), that is, 300 GB when 10 consecutive file write operations were used.

Before we evaluate the benefits and costs of elastic collective I/O, we present two motivating experiments that answer the following two questions. What is the impact of the load of one aggregating server on the file access performance? What is the impact of removing an arbitrary number of servers on the file access performance?

In the first experiment we inject a delay in response representing a server load ranging from 0 μs to 512 μs to one server. Figure 8 shows the results. For values up to 8 μs the impact is not noticeable because the performance is dominated by other I/O operations. Starting at 16 μs, the load significantly impacts the performance. For 512 μs delay the performance degradation is as large as 4x.

In the second experiment we evaluate the impact of running S-IOR with fewer aggregating servers. We varied the number

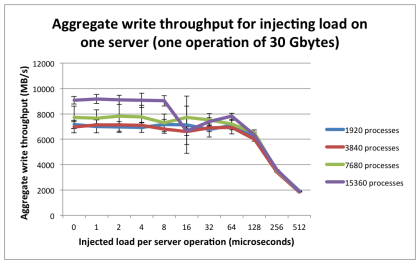


Fig. 8. Server load injection.

of servers for the four cases and plotted the results in Figure 9. In all cases the removal of a small number of servers does not have a significant impact on performance. For larger number of client processes the performance even improves with a smaller number of aggregating servers. This apparently paradoxical result is explained by the increased contention on the file system that is caused by a large number of servers. This suggests that the default parameter used in the MPI-IO driver for GPFS is not ideal for the experimental platform.

The empirical answers to the previously posed questions suggest that the load on a single aggregating server can significantly impact performance (not surprisingly, according to Amdahl's law), but that removing the loaded server can restore the performance. The next question is whether this dynamic removal can be performed with low overhead. Figure 10 shows an evaluation of the dynamic removal of a loaded server for 3,840 and 15,360 processes per application (i.e., using the policy described in Section III-C).

In this experiment we run 10 consecutive write operations writing a total of 300 GB to a shared file. A permanent load of 512 μs is injected into each file write operation of exactly one aggregating server right before the third operation. In the upper timeline of each graph we note that the aggregate write performance significantly deteriorates and remains low for the lifetime of the benchmark. In the lower timeline of each graph, after paying a high cost of performing the third access, the detection of load triggers the server removal control protocol, which removes the server during the fourth operation and starts a new epoch with fewer servers with the fifth operation. The control protocol is fully overlapped with the fourth operation, and the application perceives significantly better performance

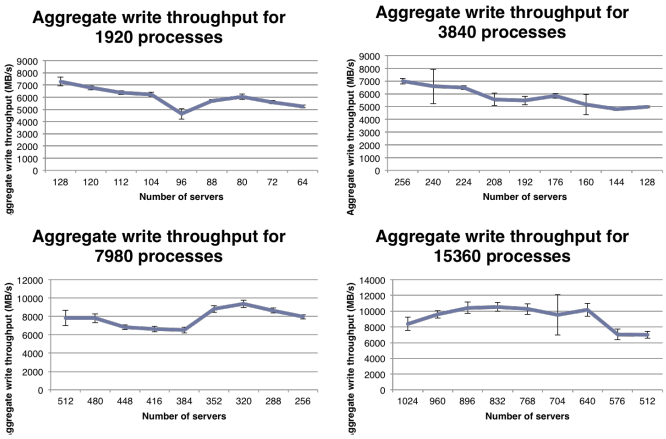


Fig. 9. Impact of server removal on aggregate write throughput.

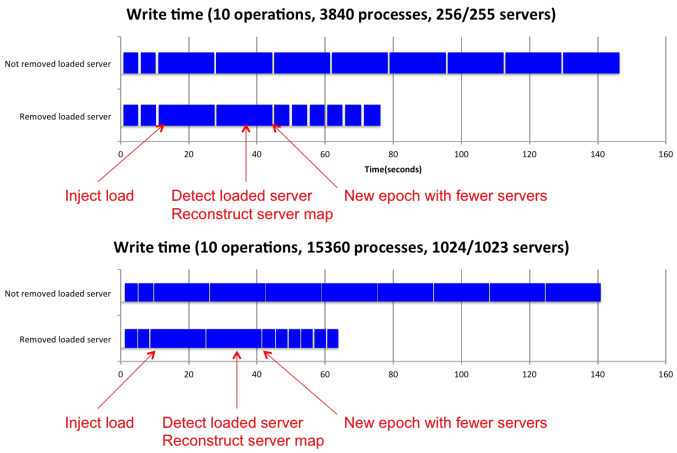


Fig. 10. Dynamic server removal results.

only starting from the fifth operations.

The left hand side of Figure 11 displays the speedup values for individual operations after the loaded server has been removed and the speedup of the overall benchmark time including the operations with the loaded server. The dynamic server removal offers a large speedup for the individual operations ranging on average between 359% and 473%. The overall benchmark speedup ranges between 188% and 220%.

We compute the cost incurred by the elastic collective I/O policy for one operation as the ratio of the maximum over all processes of the time spent performing control operations (at a control point) and the maximum over all the processes of write operation time. The right hand side of Figure 11 shows the results in percentages. In all cases the mean cost is under 0.3% of the total operation time. We consider this cost to be low compared with the performance benefits that such a policy brings.

These results demonstrate the need for dynamic load detection and avoidance policies in the software I/O stack, given the dramatic impact that they can have on the performance of a single loaded node in the system.

1) Application kernels: We repeated the load injection/detection experiment with VPICIO and VORPAL kernels using a server load of 512 μs per operation in a weak scaling scenario. VPICIO was run for 131,032 particles per process and 10 steps, which for p processes generated total data-set sizes of $p \times 5$ MB (i.e., 75 GB for 15,360 processes). For VORPALIO, we used block dimensions of sizes $x = 50$, $y = 50$, and $z = 30$; decompositions of sizes $p_x = p/15$, $p_y = 5$, and $p_z = 3$; and 10 time steps, which for p processes generated total data set sizes of $p \times 17$ MB (i.e., 257 GB for 15,360

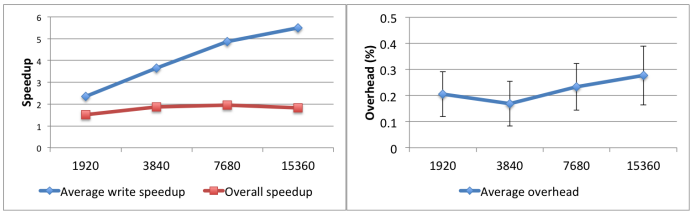


Fig. 11. Left-hand side: Speedup of elastic collective I/O for S-IOR. Right-hand side: CLARISSE overhead for elastic collective I/O in percentage from the write operation.

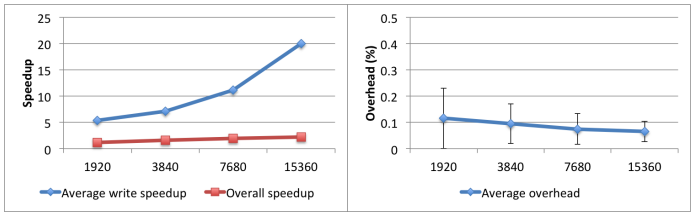


Fig. 12. Left-hand side: Speedup of elastic collective I/O for VPICIO. Right-hand side: CLARISSE overhead for elastic collective I/O in percentage from the write operation.

processes).

The left-hand sides of Figures 12 and 13 show the speedups obtained by the elastic collective I/O implementation over the version that continues with the loaded server. Both average write speedup and whole application speedup are shown. In all cases the improvement is substantial. For VPICIO there is a one order of magnitude improvement in the collective write operations for 7,680 and 15,360 processes. This improvement is due to the dynamic removal of the loaded server during the fourth iteration of the application. In the right-hand sides of the figures we can see that the policy cost is under 0.2% of the write operation time. As in the case of S-IOR, this cost is low compared with the benefit that this policy brings.

C. Parallel I/O scheduling

In this experiment we evaluate the performance of the FCFS parallel I/O scheduling described in Section III-C. In the evaluation we use three metrics: the interference factor I, the scheduling cost factor C, and the scheduling overhead. The interference factor was defined in [11] as

$$I = \frac{T_{nosched}}{T_{alone}} \tag{1}$$

, where $T_{nosched}$ is the total time of the storage I/O, when applications are running concurrently without scheduling and T_{alone} is the time of the application running alone. We define the scheduling cost factor C in similar fashion:

$$C = \frac{T_{sched}}{T_{alone}} \tag{2}$$

, where T_{sched} is the time the storage I/O requires when scheduling is used. Intuitively, the interference factor reflects the degree of overlap in time of I/O operations when no scheduling is performed. For no overlap the theoretical value of I is 1. The scheduling cost factor C reflects the contribution of two main components: the amount of waiting due to mutual exclusion and the overhead of implementing the scheduling

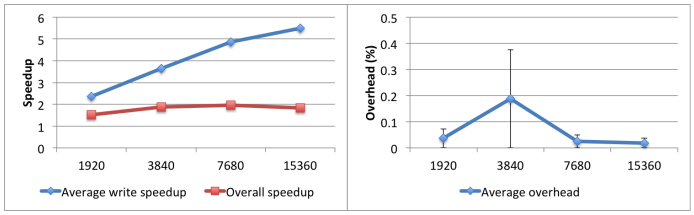


Fig. 13. Left hand side: Speedup of elastic collective I/O for VORPALIO. Right hand side: CLARISSE overhead for elastic collective I/O in percentage from the write operation.

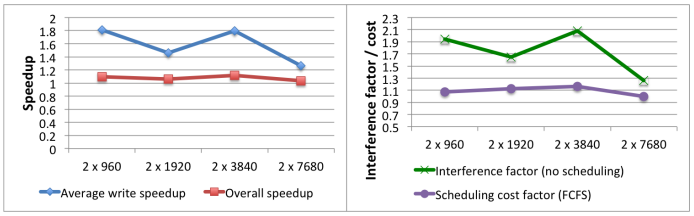


Fig. 14. Left-hand side: Speedup of FCFS parallel scheduling I/O for two concurrent instances of S-IOR. Right-hand side: Interference and scheduling cost factors.

algorithm in CLARISSE. Intuitively, the I/O scheduling improves the performance if $I > C$.

We first evaluate four cases of two concurrent instances of S-IOR, each running 960, 1,920, 3,840, and 7,680 client processes. Each S-IOR instance has 10 phases alternating a write operation to a shared file with a computation operation of 20 seconds, in a fashion similar to many scientific application patterns. Each benchmark instance writes data to its own file. Figure 16 shows the results for 960 and 3,840 clients. The performance without I/O scheduling is depicted in the upper part of each graph and the performance with FCFS scheduling in the lower part.

When no I/O scheduling is employed, the contention at servers significantly degrades the performance of write operations. Figure 14 shows the speedup that can be obtained though the FCFS policy for all four cases. The performance of individual operations improves between 132% and 198%. Overall the benchmark speedup is between 103% and 112%. This value is not as large because it includes a large fraction of computation, more than 82% in all cases. If we consider only I/O, the speedup is between 127% and 190%.

To better understand the performance, in the right-hand side of Figure 14 we plot the average over the two applications of the interference factor I and scheduling cost factor C. As expected, the speedup appears to be correlated with the difference $I - C$. The more efficient is the scheduling, the larger is the average write speedup. The values of C are significantly lower than those of I, indicating that the I/O scheduling is effective. This fact is confirmed by the obtained speedup.

We estimated the scheduling overhead for operation instances, which are chosen for scheduling without waiting. We computed the overhead as a ratio of the time required for scheduling over the total operation time. The results are plotted in Figure 15. The mean overhead is less than 0.02%, which is many orders of magnitude less than the total time, even when it shows some variability. These results demonstrate the potential beneficial impact that a simple I/O scheduling policy can have on the file write performance at a low cost in the presence of contention.

1) Application kernels: We evaluated the parallel I/O scheduling with VORPALIO and VPICIO kernels in a strong scaling experiment. VPICIO was run for 4,194,4304 / 2,097,152 / 1,048,576 / 524,288 particles per process and 10 steps, which for each run of 960, 1,920, 3,840 and 7,680 processes generated a total data set size of 150 GB per application. For VORPALIO, we used block dimensions of sizes $x = 256/(p/960)$, $y = 64$, and $z = 32$; decompositions of sizes $p_x = p/15$, $p_y = 5$, and $p_z = 3$; and 10 time steps, which for $p = 960$, 1,920, 3,840 and 7,680 processes generated a total data-set size of 112.5 GB per application.

We evaluated three scenarios: (1) two concurrent instances of VPICIO, (2) two concurrent instances of VORPALIO, and (3) two concurrent instances, one of VPICIO and one of VORPALIO. The instances were all started at the same time. Figures 17, 18, and 19 show the speedup for both average write time and overall application.

In 9 of 12 cases parallel I/O scheduling brings a performance benefit of upto 84% for average write time and upto 25% for the whole application. For VPICIO+VORPALIO, however, there was practically no speedup for 960, 1920 and 3840 processes. To better understand the performance, on the right-hand side of each figure we plot the average over the two applications of the interference factor I and scheduling cost C as we did in Section V-C for S-IOR. As expected, the speedup appears to be correlated with the difference $I - C$. A larger positive difference corresponds to a higher speedup, and a small or negative difference corresponds to no speedup. For VPICIO+VORPALIO and 960, 1,920, and 3,840 processes the lack of speedup is due to a larger than usual scheduling cost factor, which does not counterbalance the interference cost. Based on these results, we analyzed the execution traces and noted that for these cases the ratio of I/O times to computation times was high: more than 1 for 960 processes and between 0.5 and 1 for 1,920 and 3,840 processes. This substantially increased the waiting time for scheduling and therefore resulted in low or no speedup for these cases. The overhead of scheduling excluding waiting is similar to the one for S-IOR (the same operations are involved) and it is not shown here. A more extensive analysis of this trade-off between interference and I/O scheduling is a subject of future work.

VI. Related work

This section discusses related work in three areas: HPC software storage I/O stack, scalable on-line monitoring and run-time systems, and in situ and in-transit computation.

A. HPC software storage I/O stack

In the past several years increasing efforts have been made to improve the scalability and the performance of the HPC storage I/O stack. The goal of the Fast Forward I/O and Storage program [16] is to redesign the storage I/O stack for

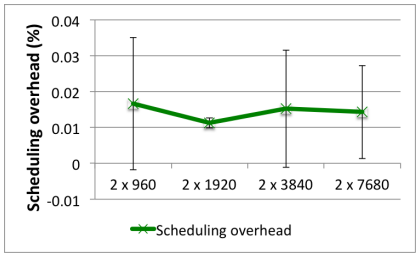


Fig. 15. Overhead of FCFS parallel scheduling I/O for two concurrent instances of S-IOR.

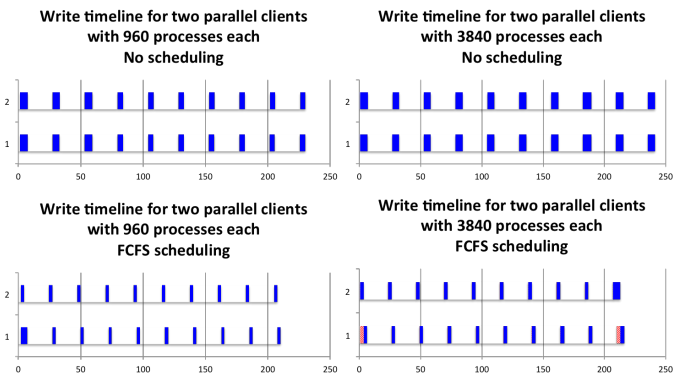


Fig. 16. FCFS parallel I/O scheduling. The blue bars correspond to write time, and the red hashed bars correspond to waiting time.

addressing the scalability requirements of exaflop systems. In turn, we target building cross-layer control abstractions that could be used for global optimization of existing or future I/O stacks. Our approach is close in spirit to IOFlow [5], a software-defined storage architecture that uses a logically centralized controller for managing the data flows between virtual machines. Unlike our approach, IOFlow targets storage in virtualized data centers and distributed applications with different requirements and APIs from those of the HPC platforms.

Most research in this area has been dedicated to improve what we call the data plane of the HPC software storage I/O stack. For instance, researchers have proposed several collective I/O implementations [8], [17], [18]. In all these approaches, however, the coordination is intrinsic; and none of them are systemwide optimizations taking into account external factors such as interference and system load.

A few studies advocate for the need for improving coordination in the HPC storage I/O stack. Song et al. [19] proposed a coordination approach based on server-side scheduling of one application at a time in order to reduce the completion time while maintaining the server utilization and fairness. Two recent studies [11], [20] address the growing impact on performance of the interference of multiple applications

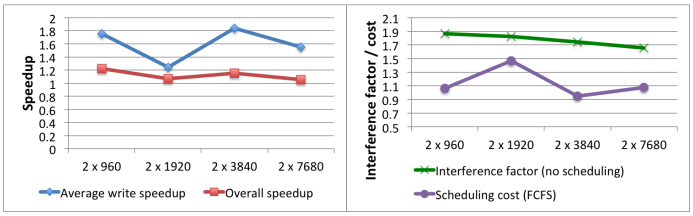


Fig. 17. Left-hand side: Speedup of FCFS parallel scheduling I/O for two concurrent instances of VPICIO. Right-hand side: Interference and scheduling

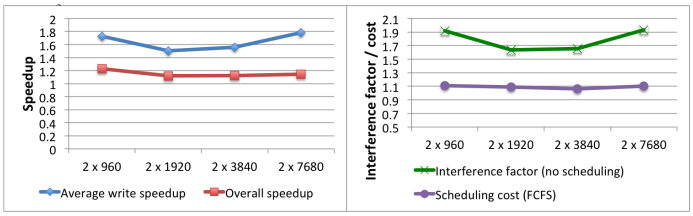


Fig. 18. Left-hand side: Speedup of FCFS parallel scheduling I/O for two concurrent instances of VORPALIO. Right-hand side: Interference and scheduling cost factors.

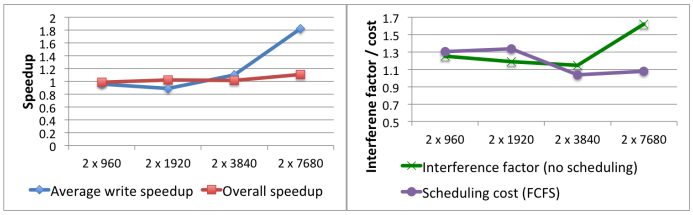


Fig. 19. Left-hand side: Speedup of FCFS parallel scheduling I/O for concurrent instances of VPICIO and VORPALIO. Right-hand side: Interference and scheduling cost factors.

accessing a shared file system by client-side scheduling of application accesses to the file systems. CLARISSE can be used to implement such one-level policies, while opening up the space of implementing a much higher range of data-staging coordination policies including multiple-level I/O scheduling.

B. Scalable on-line monitoring and run-time systems

Traditionally, the global monitoring of the HPC infrastructures has been done by system administrators. However, as the HPC infrastructure scales are continuously growing, there is an increasing need for scalable high-performance monitoring libraries that can be used by system software and library developers for implementing adaptive algorithms in the face of increasing probability of congestion and failure. LDMS [9] provides a distributed metric service that can be used for on-line monitoring. However, LDMS is currently a research effort, and this kind of library is not available on the current platforms. The CIFTS infrastructure [21] provides a fault-tolerant backplane for global dissemination of fault information. The Argo [22] and Hobbes [23] projects investigate operating systems and run times for future exascale systems. They both use a global information bus for the dissemination of run-time information about events such as faults or congestion to a hierarchy of enclaves (logical partitions of the system into groups of nodes). The CLARISSE project complements these approaches by focusing on coordination of data staging.

C. In situ and in-transit computation

As shared file systems are currently reaching their scalability limit under increasing parallelisms and data requirements, in situ and in-transit locality exploitation has become key for pushing the scalability beyond the current level. Aspects currently addressed include data staging [24], in situ and in-transit data analysis and visualization [25], [26], [27], publish-subscribe paradigms for coupling large-scale analytics [28], and flexible analytics placement tools [29]. These techniques require coordination between the data staging and the data consumers in the data path. In most of these approaches the control is embedded in the frameworks, and designing novel coordination approaches is complex. CLARISSE seeks to alleviate this problem by separating control and data paths and facilitating the development of novel coordination policies based on the control backplane.

VII. Conclusions

In this paper we presented CLARISSE, a framework designed to improve the data-staging coordination on scalable

HPC platforms. The CLARISSE design consists of data, control, and policy layers. This approach offers a significant degree of flexibility. The CLARISSE data plane offers independent and collective I/O operations. The CLARISSE control plane is generic and fully decoupled from the data plane. Ideally, the control plane and data plane should be codesigned, but CLARISSE allows the control data plane to be used with any existing data plane. CLARISSE opens up a large space of implementing various data-staging coordination policies for cross-layer distributed coordination of the software I/O stack. In this paper we presented two case studies, an elastic collective I/O and a parallel I/O scheduling implementation. We demonstrated empirically that CLARISSE can bring a significant performance benefit at a low cost for elastic collective I/O and parallel I/O scheduling.

We are currently investigating several research directions based on the foundations presented in this paper. First, we plan to design and implement adaptive policies for data aggregation and staging targeting high-performance and high resource utilization. In particular, we will extend the elastic collective I/O policy to address more complex load patterns that occur on HPC platforms. Second, we will actively research novel parallel I/O scheduling policies that reduce the noise perceived by the applications. In particular, we plan to look at policies at various layers including aggregation, burst buffers, and file systems. Third, we will investigate how CLARISSE can be used to improve the resilience of the software I/O stack. Fourth, CLARISSE offers proper mechanisms for supporting the necessary coordination for data sharing and staging for complex workflows of applications. We plan to explore how these mechanisms can be applied in real scientific workflows consisting of multiple simulations and combinations of simulation and analysis/visualization. Fifth, the CLARISSE run time will highly benefit from an on-line scalable and high-performance monitoring framework that offers a dynamic low-latency view of a large-scale system, including fault notification, aggregation of metrics, and congestion detection. Several efforts in this direction are promising [9], [22], [23], and we plan to capitalize on these in order to significantly improve the scalability and performance of the software I/O stack on future platforms.

REFERENCES

[1] R. Ross, G. Grider, E. Felix, M. Gary, S. Klasky, R. Oldfield, G. Shipman, and J. Wu, "Storage Systems and Input/Output to Support Extreme Scale Science," Department of Energy, Tech. Rep., 2015.

[2] F. Isaila, J. Garcia, J. Carretero, R. Ross, and D. Kimpe, "Making the Case for Reforming the I/O Software Stack of Extreme-Scale Systems," *Elsevier's Journal Advances in Engineering Software*, 2015.

[3] M. Bancroft, J. Bent, E. Felix, G. Grinder, J. Nunez, S. Poole, R. Ross, E. Salmon, and L. Ward, "HEC File Systems and I/O Workshop Document. http://institute.lanl.gov/hec-fsio/docs/." Tech. Rep., 2011.

[4] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN," *Queue*, vol. 11, no. 12, p. 20, 2013.

[5] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A Software-defined Storage Architecture," in *Proceedings of the Twenty-Fourth ACM SOSP*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 182–196.

[6] F. J. G. Blas, F. Isaila, D. E. Singh, and J. Carretero, "View-Based Collective I/O for MPI-IO," in *8th IEEE CCGrid 2008*, 2008, pp. 409–416.

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[8] R. Thakur, W. Gropp, and E. Lusk, "Data Sieving And Collective I/O In ROMIO," in *Proceedings of FRONTIERS '99*. IEEE Computer Society, 1999, pp. 182–189.

[9] A. Agelastos and et al., "The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications," in *Proceedings of SC '14*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 154–165.

[10] "MPI: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[11] M. Dorier, G. Antoniu, R. B. Ross, D. Kimpe, and S. Ibrahim, "CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination," in *28th IEEE IPDPS*, Phoenix, AZ, 2014.

[12] H. Shan, K. Antypas, and J. Shalf, "Characterizing and Predicting the I/O Performance of HPC Applications Using a Parameterized Synthetic Benchmark," in *Proceedings of SC '08*, 2008, pp. 42:1–42:12.

[13] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming Parallel I/O Complexity with Auto-tuning," in *Proceedings of SC '13*, 2013, pp. 68:1–68:12.

[14] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh Performance Three-Dimensional Electromagnetic Relativistic Kinetic Plasma Simulationa)," *Physics of Plasmas*, vol. 15, no. 5, p. 055703, May 2008.

[15] C. Nieter and J. R. Cary, "VORPAL: A Versatile Plasma Simulation Code," *J. Comput. Phys.*, vol. 196, no. 2, pp. 448–473, May 2004.

[16] *The Fast Forward Storage and I/O Program. Available at https://wiki.hpdd.intel.com/.*

[17] F. Isaila, G. Malpohl, V. Olaru, G. Szeder, and W. Tichy, "Integrating Collective I/O and Cooperative Caching into the 'Clusterfile' Parallel File System," in *Proceedings of ACM ICS*, 2004, pp. 58–67.

[18] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, "Server-Directed Collective I/O in Panda," in *Proceedings of SC '95*, ser. Supercomputing '95. New York, NY, USA: ACM, 1995.

[19] H. Song, Y. Yin, X.-H. Sun, R. Thakur, and S. Lang, "Server-Side I/O Coordination for Parallel File Systems," in *Proceedings of SC '11*, pp. 17:1–17:11.

[20] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir, "Scheduling the I/O of HPC Applications Under Congestion," in *In IPDPS 2015*, 2015, pp. 1013–1022.

[21] R. Gupta, P. Beckman, B.-H. Park, E. Lusk, P. Hargrove, A. Geist, D. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems," *2014 43rd ICPP*, pp. 237–245, 2009.

[22] S. Perarnau et al., "Distributed Monitoring and Management of Exascale Systems in the Argo Project," in *Distributed Applications and Interoperable Systems*, 2015, pp. 173–178.

[23] R. Brightwell, R. Oldfield, A. B. Maccabe, and D. E. Bernholdt, "Hobbes: Composition and Virtualization As the Foundations of an Extreme-Scale OS/R," in *Proceedings of ROSS '13*, pp. 2:1–2:8.

[24] T. Jin, F. Zhang, Q. Sun, H. Bui, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager, C. Chang, and M. Parashar, "Leveraging Deep Memory Hierarchies for Data Staging in Coupled Data Intensive Simulation Workflows," in *IEEE Cluster 2014*, 2014.

[25] M. Dreher and B. Raffin, "A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Chicago, IL, USA, May 26-29, 2014*, 2014, pp. 277–286.

[26] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward Simulation-Time Data Analysis and I/O Acceleration on Leadership-Class Systems." in *LDAV*, D. Rogers and C. T. Silva, Eds. IEEE, 2011, pp. 9–14.

[27] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O," in *IEEE CLUSTER*, 2012.

[28] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-Based Publish/Subscribe System for Large-Scale Science Analytics." in *CCGRID*, 2014, pp. 246–255.

[29] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu, "FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics," in *27th IEEE IPDPS 2013*, 2013, pp. 320–331.