ZOID: I/O-Forwarding Infrastructure for Petascale Architectures *

Kamil Iskra[1] John W. Romein[2] Kazutomo Yoshii[1] Pete Beckman[1]

[1] Argonne National Laboratory
Mathematics and Computer Science Division
9700 South Cass Avenue
Argonne, IL 60439, USA
{iskra,kazutomo,beckman}@mcs.anl.gov

[2] Stichting ASTRON (Netherlands Foundation for Research in Astronomy)
Oude Hoogeveensedijk 4
7991 PD Dwingeloo, The Netherlands
romein@astron.nl

Abstract

The ZeptoOS project is developing an open-source alternative to the proprietary software stacks available on contemporary massively parallel architectures. The aim is to enable computer science research on these architectures, enhance community collaboration, and foster innovation. In this paper, we introduce a component of ZeptoOS called ZOID—an I/O-forwarding infrastructure for architectures such as IBM Blue Gene that decouple file and socket I/O from the compute nodes, shipping those functions to dedicated I/O nodes. Through the use of optimized network protocols and data paths, as well as a multithreaded daemon running on I/O nodes, ZOID provides greater performance than does the stock infrastructure. We present a set of benchmark results that highlight the improvements. Crucially, the flexibility of our infrastructure is a vast improvement over the stock infrastructure, allowing users to forward data using custom-designed application interfaces, through an easy-to-use plug-in mechanism. This capability is used for real-time telescope data transfers, extensively discussed in the paper. Plug-in–specific threads implement prefetching of data obtained over sockets from an input cluster and merge results from individual compute nodes before sending them out, significantly reducing required network bandwidth. This approach allows a ZOID version of the application to handle a larger number of subbands per I/O node, or even to bypass the input cluster altogether, plugging the input from remote receiver stations directly into the I/O nodes. Using the resources more efficiently can result in considerable savings.

Categories and Subject Descriptors D.4.4 [*Operating Systems*]: Communications Management—Input/output; D.4.8 [*Operating Systems*]: Performance—Measurements; C.5.1 [*Computer System Implementation*]: Large and Medium ("Mainframe") Computers—Super (very large) computers; J.2 [*Physical Sciences and Engineering*]: Astronomy

General Terms Design, Experimentation, Measurement, Performance

Keywords I/O forwarding, petascale

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.
LOFAR is funded by the Dutch government in the BSIK programme for interdisciplinary research for improvements of the knowledge infrastructure. Additional funding is provided by the European Union, European Regional Development Fund (EFRO) and by the "Samenwerkingsverband Noord-Nederland," EZ/KOMPAS.

PPoPP'08, February 20–23, 2008, Salt Lake City, Utah, USA.

1. Introduction

A supercomputer is a device for turning compute-bound problems into I/O-bound problems—this half-humorous definition, attributed to either Seymour Cray or Ken Batcher, holds true today more than ever.

File I/O bandwidth available in current-generation supercomputers varies greatly per installation. One of the strongest machines, the ASC Purple [1], with a computational power of about 100 TF, has file I/O bandwidth of about 100 GB/s—1 GB/s per 1 TF. The consensus for contemporary machines appears to be that at least 0.5 GB/s per 1 TF is desirable. The upcoming IBM Blue Gene/P machine, to be hosted at Argonne National Laboratory, will have a peak computational power of around 500 TF. Even though Argonne is planning to spend as much as 20% of the cost of the whole system on the I/O, the per-TF I/O bandwidth will be significantly lower than on current machines, possibly by as much as an order of magnitude. The increase in the computational power, primarily due to a larger number of cores per node, is just too large to keep up with. Instead of binding the I/O throughput to the computational power, therefore, Argonne decided to bind it to the system memory size. The design goal is to be able to perform a full memory dump in no more than 30 minutes. For comparison, a similar action on a laptop would take no more than 1–2 minutes.

Since I/O is clearly one of the most precious resources in a supercomputer, every effort must be made to ensure that it is used to its full capacity. We would like to have control over the complete I/O path, from application processes running on compute nodes to the fileservers. This would allow us to, for example, instrument I/O operations in the application and individually track those instrumented operations through each level of the I/O stack. We would then have much-needed insight into where bottlenecks occur and by fixing them would improve the performance.

Unfortunately, contemporary massively parallel architectures, such as the IBM Blue Gene/L [13] or the Cray XT3 [8], have seriously limited flexibility in this area. They employ a number of closed-source, proprietary technologies, ranging from special-purpose compute node kernels to interconnection protocols, which by their very nature are unamenable to external modification or in-

strumentation. This situation limits the scope of independent computer science research that can be performed on these state-of-the-art architectures. It can also affect ordinary execution of applications. Some codes will not run at full speed, because the high-performance interfaces required are simply not available, such as the parallel I/O API between the compute nodes and the fileservers. Some codes cannot be run at all, as they expect a much richer runtime environment on the compute nodes than the proprietary components provide; a good example is LLNL's KULL [12], which cannot be run on LLNL's largest supercomputer.

The shortcomings of available software stacks and the lack of a community-owned toolkit for exploring and experimenting with optimizations for large-scale machines prompted us to launch the development of the ZeptoOS project [16]. ZeptoOS uses widely available open-source components such as the Linux kernel to develop an alternative, fully open software stack on large-scale parallel systems. The aim is to enable computer science research on these architectures, enhance community collaboration, and foster innovation.

The focus of this paper is on ZOID—an I/O-forwarding component of ZeptoOS. ZOID aspires to become the I/O-forwarding infrastructure of choice on the upcoming petascale platforms, especially the IBM Blue Gene/P and the Cray XT5 (see Section 5). The current implementation, as described in this paper, works on the IBM Blue Gene/L and is freely downloadable from the ZeptoOS project website [16].

ZOID has been highly optimized. Through the use of optimized network protocols and data paths, as well as multithreading, it frequently offers a higher performance than does the stock IBM infrastructure. Even more important, ZOID is far more flexible: it can be easily extended with custom application interfaces through the use of plug-ins, allowing application or middleware writers to transfer data in and out of the machine conveniently.

As an example, we elaborate on a plug-in that is used to communicate real-time telescope data from the LOFAR (LOw Frequency ARray) radio telescope. This telescope is currently being constructed and is operational at a small scale. A dedicated Blue Gene/L system provides the computational power to process the data centrally, but meeting the external bandwidth requirements turned out to be hard. ZOID provides the high throughput required to handle the large amounts of data. Moreover, its flexibility enabled a redesign of the system that uses the resources much more efficiently.

Massively parallel systems typically run a stripped-down operating system lacking the capability to perform file I/O (see, e.g., BG/L CNK [13] and XT3 Catamount [9]). This design is motivated by a desire to reduce memory usage, system complexity, and operating system noise (jitter) [2–4]. Naturally, applications still expect file I/O to be available; at least two solutions to that problem have emerged. On the Cray XT [8], applications need to be linked with the SYSIO library [9] and the client part of the Lustre [10] filesystem protocol. This approach is flexible but has a major disadvantage: if the application process crashes, the heavyweight Lustre client, which runs in the same context, goes down as well, potentially resulting in state inconsistencies that are difficult to recover from gracefully. The filesystem client code can also be a significant source of noise. On the IBM Blue Gene [13], an additional layer of intermediary *I/O nodes* is used instead. These nodes act as the filesystem clients, and file I/O operations from the application processes are forwarded to them and executed there. Such separation truly reduces complexity on the compute nodes and presents an opportunity for interesting optimizations. This is the design strategy followed by ZOID.

The remainder of this paper is organized as follows. Section 2 describes the I/O infrastructure of IBM Blue Gene/L and the design and implementation of ZOID. Section 3 presents results from experiments that compare the performance of ZOID with the stock infrastructure. Section 4 discusses the challenges of transferring real-time telescope data and the experiences with using ZOID for this task. Section 5 presents the conclusions, ongoing efforts, and future directions.

2. Architecture

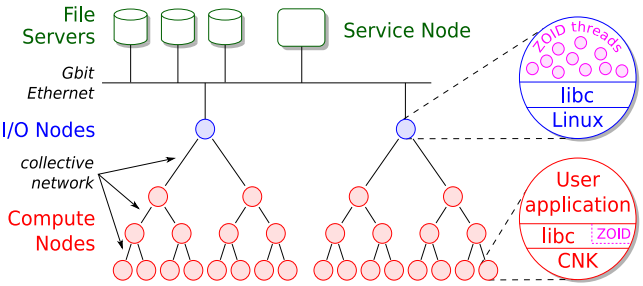

Figure 1. IBM Blue Gene/L I/O architecture (with ZOID-specific components on the right).

Figure 1 presents an overview of the I/O architecture of the IBM Blue Gene/L and also shows how ZOID fits into that architecture. A single IBM Blue Gene/L rack consists of 1,024 compute nodes, each equipped with a dual-core PowerPC 440 CPU, running at a relatively low 700 MHz to reduce heat dissipation. The primary interconnect between the compute nodes is a *3D torus network* (not shown in the figure). For more information on Blue Gene, see the special issue of IBM's Journal of Research and Development [13].

2.1 I/O Infrastructure

The compute nodes (Fig. 1, bottom) run a stripped-down *compute node kernel* (CNK) that lacks the capability to perform file I/O. The nodes are arranged in a binary tree on the *collective network*; I/O nodes are also attached to that network. I/O nodes feature basically the same hardware as compute nodes but run Linux and also have Gigabit-Ethernet links connecting them to the fileservers and the job management system (*service node*). Each I/O node has a fixed subset of compute nodes allocated to it; together these nodes form a *pset*. Depending on the machine configuration, the I/O node to compute node ratio can vary between 1:8 and 1:64. CNK forwards file and socket I/O requests over the collective network to I/O nodes, where they are processed by the Linux kernel.

This model provides for hierarchical I/O layers and reduces the complexity of compute node software. It also provides good scalability—adding a thousand compute nodes should not increase by a thousand the number of simultaneous file system mount requests to the storage servers. I/O nodes, being less numerous than compute nodes, act as "client reducers"—to the fileservers, it appears as if there are fewer (albeit possibly more active) clients to take care of.

A process on I/O nodes called CIOD (*Control and I/O Daemon* [11]) plays a key role in the I/O-forwarding infrastructure. It receives I/O requests forwarded from the compute nodes over the collective network and invokes corresponding Linux system calls. The implementation of CIOD on the Blue Gene/L is basic; it is a single-threaded process that handles I/O requests one by one.

The whole infrastructure is unfortunately inflexible: it is impossible to forward any other calls but the hardcoded subsets of POSIX file I/O and BSD socket APIs. This can result in optimization opportunities being lost, as shown in Figure 2. Even though the application uses high-performance parallel I/O via MPI-IO, most of that effort goes in vain, because the calls need to be translated early on into the POSIX API, as that is the only interface that the CNK

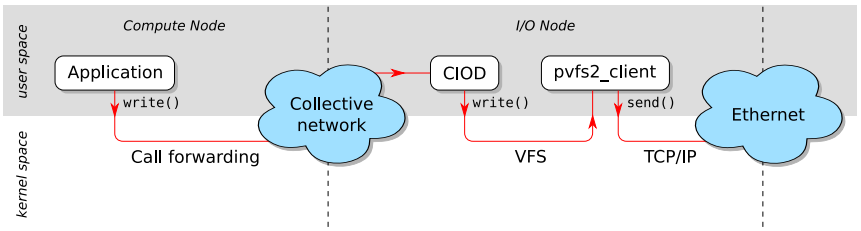

Figure 2. I/O path with CIOD and PVFS.

supports. Subsequently, a significant number of user-kernel context switches take place before the data finally reaches the Ethernet network—especially with PVFS [6], as its client core is implemented in user space. In Section 5 we show some of the shortcuts that can be taken if parallel I/O API can be used between the compute and I/O nodes.

2.2 ZeptoOS I/O Daemon

Having been developed for the Blue Gene/L, ZOID follows the design principles of IBM I/O infrastructure, with a separation of compute and I/O nodes, and so forth. At the lowest level, ZOID is a highly optimized function call-forwarding infrastructure between application processes running on stripped-down compute nodes and daemons on I/O nodes.

It would have been relatively straightforward to replace IBM's CIOD, were it not for the fact that the protocols CIOD uses to communicate with the service node and with the compute nodes are proprietary and undocumented (on the collective network, even the interface to send and receive packets is largely undocumented). We considered completely replacing the protocol on the collective network with our own, but we found that approach to be unrealistic. The problem is that the stock protocol does not allow for a clean separation of application I/O from low-level system control. While we wanted to replace the I/O part, we did not want to deal with system control messages, some of which seemed complex.

In the end, we opted for a hybrid approach. We use CIOD to perform the initial handshaking and loading of the the application code onto the compute nodes. Once started, a ZOID-enabled application sends a special packet to the I/O node that notifies the ZOID daemon running there. The daemon then suspends CIOD, thus gaining full control over the collective network, allowing us to use our own protocol from then on. We do not use the protocol engine in the CNK; instead, we implement our own, in user space, inside a replacement version of the `libc` library that is linked with the application. CNK has an access control mechanism that prevents the user space from accessing the required channel of the collective network—we had to disable it (we distribute the necessary patch with ZeptoOS [16]). When the application is about to terminate, we resume CIOD, so that it can gracefully shut the nodes down. This approach lets us avoid having to implement most of the system control protocols within ZOID. We still had to implement some of its elements, such as the handling of crashing compute node processes or the forwarding of the standard output and error streams from the application processes to the service node. Although the whole process might seem complicated, ZOID is fully integrated into the ZeptoOS software stack, and enabling it is as easy as toggling a single option on the ZeptoOS configuration screen. All the complexity is hidden from the end-user within the I/O node ramdisk startup scripts and a replacement compute node `libc`. From an end-user's perspective, the differences are limited to the final linking stage of the application (replacement `libc` must be used), and possibly also to job submission (a ZOID-enabled kernel profile needs to be selected, if it is not the default).

The custom protocol used by ZOID offers a greater flexibility than does the standard one. In particular, it allows for extensibility via a *plug-in* mechanism. By default, ZOID provides the UNIX plug-in, which forwards standard file I/O and socket APIs (we support a more complete API subset than CIOD). The forwarding is handled transparently by our replacement `libc`. Multiple plug-ins can be active, and they can provide a functionality highly tuned to the particular application. A plug-in consists of three parts: an automatically generated linker library with stub functions to be invoked on the compute nodes; a corresponding automatically generated shared object on I/O nodes that invokes the forwarded function calls; and a hand-coded implementation part running on I/O nodes, which provides the actual implementation code for the forwarded function calls. The linker library needs to be linked with the compute node application, while the other two objects can be dynamically loaded at startup by the ZOID daemon. Applications can invoke the forwarded functions just like any local ones; the execution of a compute process will block until a result is received from the I/O node.

The automatic plug-in code generator is a script, written in Perl, that takes a C header file as input and extracts function prototypes out of it. The prototypes need to adhere to a particular convention, with additional hints to the generator (such as whether an argument is an input, output, or both) in the form of C-style comments. Figure 3 illustrates the annotations for the POSIX `read()` call. An advantage of this approach over an IDL-based one is that we have only one header file to maintain. The function-forwarding infrastructure supports passing of objects (by value and by pointer), C-style character strings, and one- and two-dimensional arrays. There is a user-adjustable limit on the maximum size of function input or output; it is needed because a single I/O node handles operations from multiple compute nodes and thus could easily get overwhelmed by overly large requests. The ZOID daemon exports a small API to the hand-coded implementation functions, allowing them to, for example, find out which compute node process invoked them.

```
ssize_t unix_read(
  int fd /* in:obj */,
  void *buf /* out:arr:size=+1:zerocopy */,
  size_t *count /* inout:ptr */);
```

Figure 3. Declaration of the POSIX `read()` function (`size=+1` in the array argument indicates that the size of that array is provided in the next argument).

Performance was one of ZOID's key design criteria. User-space implementation on the compute nodes reduces the latency by avoiding context switches. It also simplifies support for *zero-copy* operations, which ZOID supports for both input and output. Extensive and highly flexible support for zero-copy operations is also available on the I/O node side; the buffer for the zero-copy data can be provided either by the ZOID daemon or by a custom allocation function from the plug-in. To further reduce the latency, our col-

lective network protocol uses the eager sending mode irrespective of the message size. This can potentially reduce fairness, as compute nodes closer to the I/O node in the collective network's highly irregular tree topology get a preferential treatment, resulting in a near-starvation of the "far" nodes if communication is intensive. Therefore, a rendezvous mode is also available as an option. Requests from compute nodes to transfer "large" messages are then queued in a FIFO order, and only one such message at a time can be transmitted.

Another important characteristic of the ZOID daemon is that it is multithreaded. This allows it to concurrently handle operations from multiple application processes, unlike the CIOD on the Blue Gene/L, which processes them one at a time. Concurrent processing has the potential of reducing the latency and improving the overall performance by exposing the parallel nature of I/O operations to the underlying file and socket I/O subsystems, allowing them to better use the available network bandwidth. Multithreading also significantly increases flexibility, enabling the implementation functions to block for arbitrarily long time while the remaining application processes communicate unobstructed using other ZOID threads. This allows us to, for example, block on a read from an empty socket without causing a deadlock (IBM's CIOD avoids this particular issue by converting the compute nodes' blocking calls to nonblocking ones on the I/O nodes).

Multithreading comes with its own set of challenges, particularly on the Blue Gene/L, where, because of a lack of cache coherence, only one of the two CPU cores is exposed to the Linux kernel (see Section 5 for our current work in this area). Only one thread can thus run at any particular time, so careful tuning was required to ensure that the scarce CPU resources were not being wasted on needless switching between the threads. We can receive a message from a compute node, invoke the application function, and send a reply, all within a single thread. For that matter, we send and receive messages on the collective network within a single loop, to further improve resource utilization and reduce the latency. On compute nodes, where, at least in virtual node mode [11, 13], we have two CPU cores at our disposal, we can even send a message from one core while simultaneously receiving a reply on the other.

3. ZOID I/O Performance

In this section, we explore the performance of ZOID, comparing it to the stock IBM CIOD. The experiments presented here have been conducted on the Argonne Blue Gene, which is a single-rack (1,024 nodes) BG/L machine with an I/O node (ION) to compute node (CN) ratio of 1:32. IBM driver version V1R3M2 has been used, with ZeptoOS kernel and ramdisk running on I/O nodes. The experiments discussed below are generally limited to a single pset and have been conducted with the number of compute nodes ranging between 1 and 32, and one I/O node. The results remain valid for larger configurations because individual I/O nodes are largely independent, as are the parts of the collective network forming psets. Any contention due to scaling can occur only outside the Blue Gene racks (such as within the fileservers) and is outside the scope of this research. The compute nodes run in coprocessor mode; that is, only one core per node runs user code. We measure throughput as a function of an application buffer size used. With CIOD, large application buffers are normally fragmented into chunks of 256 KB, which would put ZOID at an unfair advantage in certain cases; luckily, it is possible to adjust the CIOD buffer size threshold, and we increased it enough to avoid any fragmentation. Every experiment has been repeated at least three times. We show the best result achieved (this is especially important for experiments that used shared resources, in particular the PVFS filesystem, as in some cases those resources were shared with other jobs running on the machine at the same time, resulting in performance fluctuations).

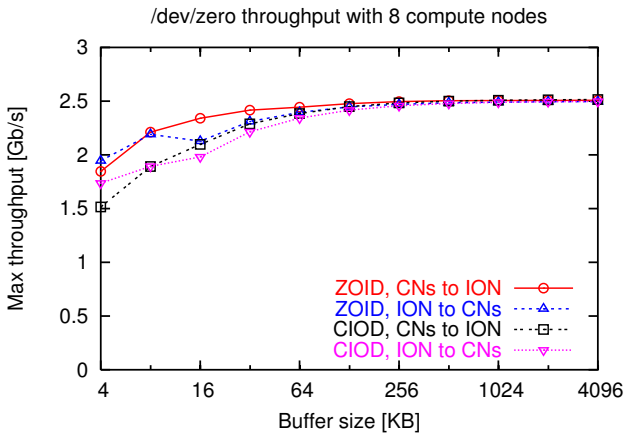

Figure 4. Base collective network performance.

Figure 4 compares the base throughput achieved on the collective network. We show the results for 8 compute nodes here, as they are fairly representative of the whole range tested. Conceptually, to measure the base performance, one transfers large quantities of data across the network, without any processing of that data. While with ZOID we could in principle create a suitable plug-in for that purpose, this is not really possible with CIOD. We have thus settled for reading from and writing to `/dev/zero` on I/O nodes—data gets passed on the collective network just as if we were accessing an ordinary file, and very little processing needs to take place on the I/O node side.

The collective network has a theoretical peak raw bandwidth of 2.8 Gb/s [13].[1] Once the protocol overheads are subtracted, the maximum user data throughput we manage to achieve is around 2.5 Gb/s (see Figure 4). Both ZOID and CIOD achieve that maximum, in either direction. As can be observed, however, with ZOID the performance is clearly superior for smaller buffer sizes (128 KB and less). This result is important to us for at least two reasons. First, we attribute the performance increase to the use of the eager protocol (CIOD uses rendezvous for almost all messages) and the fully user-space implementation of ZOID on the compute nodes. The result thus shows that the advantages are not just theoretical. Second, this result shows that the overhead of the more complex, multithreaded implementation of the ZOID daemon is quite acceptable.

The above experiment could be considered slightly artificial, because on a BG/L the 1 Gb/s Ethernet network makes it impossible to send the data out of an I/O node at anywhere near the speed of the collective network. With ZOID, however, it is possible to use that excess bandwidth by performing data processing on I/O nodes, as described in Section 4.

Figure 5 shows results from a more practical experiment: data transfer to and from an NFS filesystem (we used a dedicated server, not shared with other users). Note that, unlike the previous figure, the y axis is scaled in MB/s, not Gb/s. The outcome is actually mixed, and we include those results here primarily as a warning. In the case of both read and write, ZOID slightly outperforms CIOD for small buffer sizes if only one compute node process is used; this result is consistent with Figure 4. If we increase the number of compute node processes, however, the situation changes: with reading, ZOID significantly outperforms CIOD; with writing, the opposite occurs. More important, however, is that irrespective of which in-

[1] Throughout this paper, we use Gb/s or Mb/s in the context of network interface speeds; 1 Mb equals 1,000,000 bits. In other context, we use MB/s or KB/s; 1 KB equals 1,024 bytes.

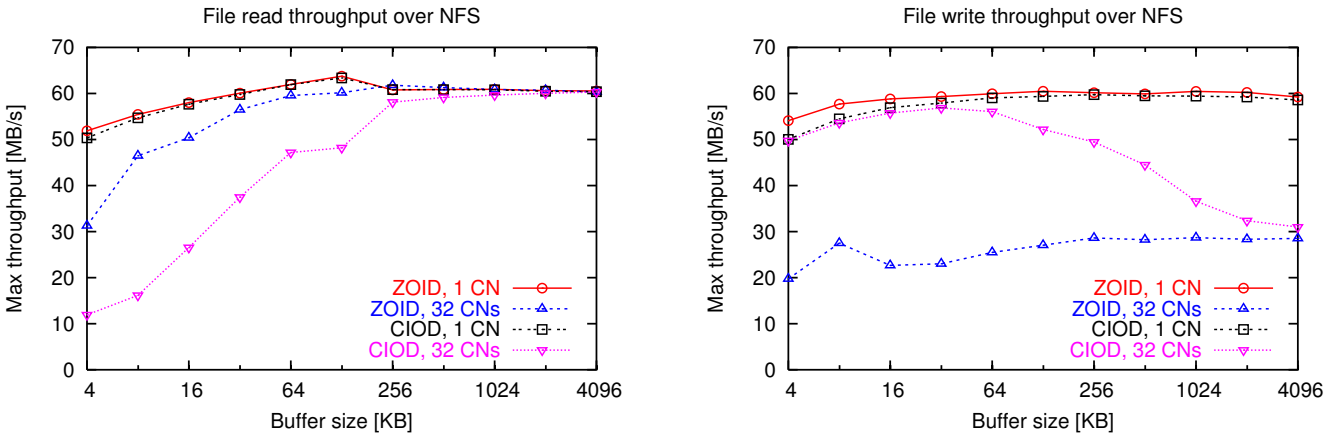

Figure 5. NFS file I/O performance: read (*left*) and write (*right*).

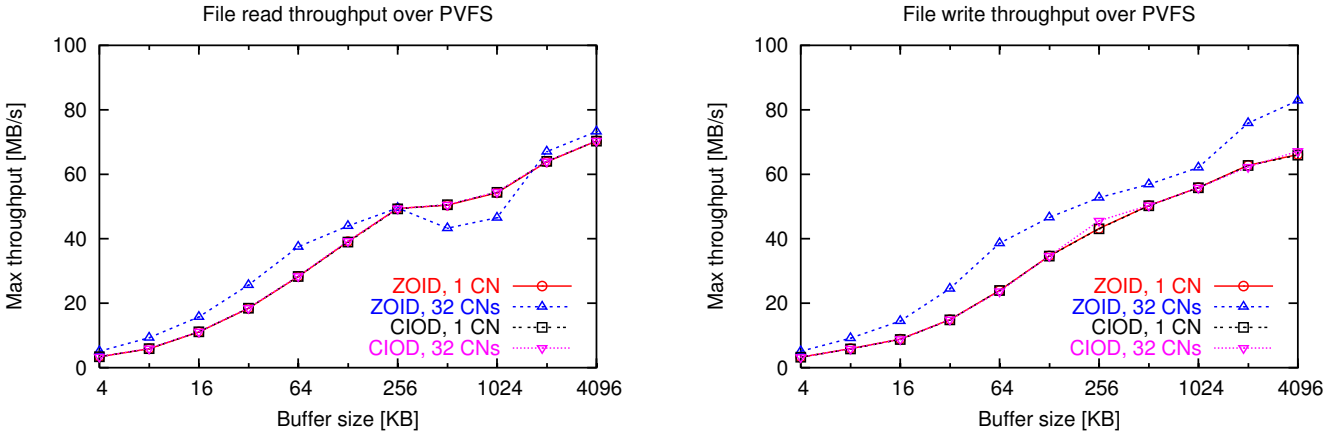

Figure 6. PVFS file I/O performance: read (*left*) and write (*right*).

frastructure is used, the performance with multiple compute node processes is significantly lower than with just one—hardly an encouraging result. The reason is that NFS is heavily optimized for sequential access: its caching and prefetching algorithms are of huge help with one client process. Anyone who has ever had a UNIX home directory on a busy NFS filesystem can readily attest that the performance tends to fall to pieces with multiple clients accessing the server simultaneously. The results presented here show that the problem extends to the case of a single client, provided that it issues multiple requests in parallel.

Multithreading is thus not a panacea—if one hopes to achieve good file I/O performance from parallel applications, a filesystem suited for the task, such as PVFS [6], needs to be used. Figure 6 presents the corresponding results, obtained with a PVFS 2.6.3 filesystem spanning across 14 servers, shared with other users of Blue Gene. With one compute node, the performance of CIOD and ZOID is basically the same, and essentially equals the performance of CIOD with 32 compute node processes, since CIOD serializes file I/O. ZOID with 32 compute node processes acts differently. Let us focus on writing (Fig. 6, right) first. One can see a visible improvement across the x axis, reaching as much as 64% for buffer size 64 KB, or over 16 MB/s for buffer size 4 MB. PVFS is thus quite capable of exploiting the I/O parallelism ZOID presents it with. Unfortunately, the picture is not all rosy. When one looks at the performance of reading (Fig. 6, left), the behavior for small buffer sizes (4–64 KB) is similar to that for writing. However, the performance then degrades, in the range of 512–1024 KB becoming worse than CIOD or ZOID with a single compute node, only to recover for the largest buffer sizes. We have been in touch with the PVFS developers regarding this phenomenon and hope that the problem, which is apparently inside the PVFS client code, will soon be fixed.

Figure 7 compares CIOD and ZOID TCP-socket bandwidth as a function of message size. The throughput numbers do not include protocol overhead. The measurements were made on the LOFAR Blue Gene/L; the parameters of CIOD and ZOID, as well as those of the Linux TCP/IP stack, have been tuned to achieve maximum performance. IBM tripled the CIOD performance during the past two years (see [14], Fig. 12). ZOID outperforms CIOD for small and medium-sized messages, but both are on a par for large messages and approach the theoretical maximum of 1.0 Gb/s when writing.

4. Communicating Real-Time Telescope Data with ZOID

ZOID is used extensively in the LOFAR radio telescope that is being built. LOFAR is the first of a new generation of telescopes and combines the signals of thousands of simple, omnidirectional antennas rather than expensive dishes [5]. Moreover, much of the

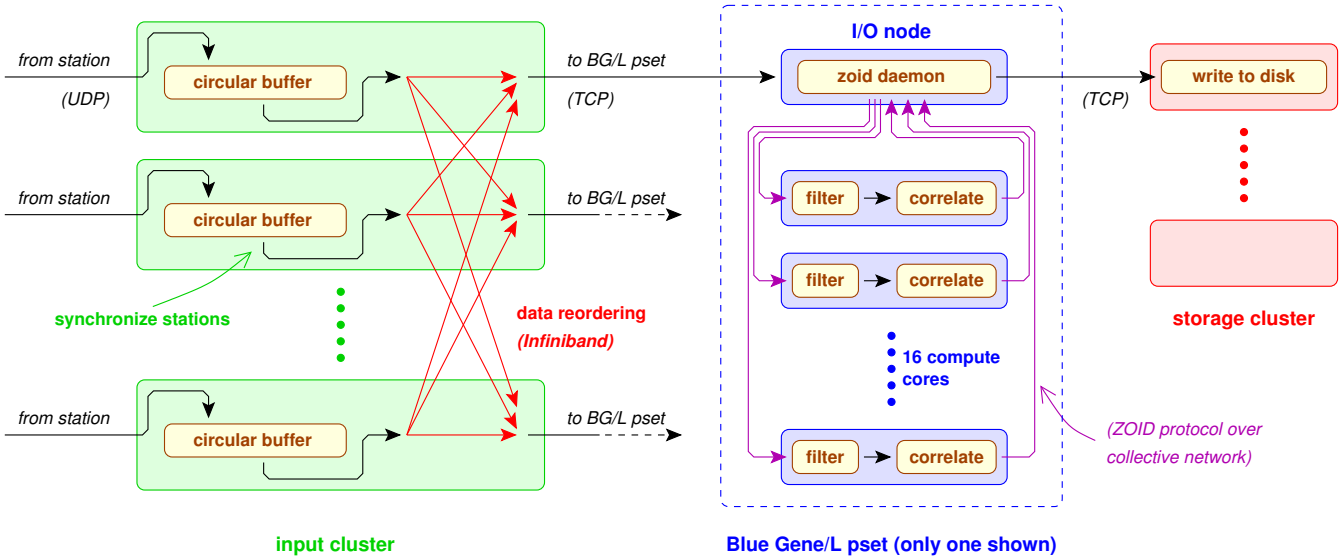

Figure 8. LOFAR central processing data flow.

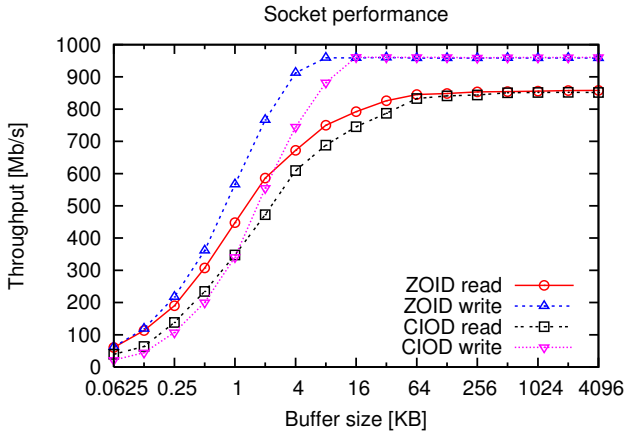

Figure 7. TCP socket performance.

signal processing is done in software, where traditionally custom-built hardware has been used. Eventually, LOFAR will be one of the world's largest telescopes. Its center is located in the Netherlands; several European outer stations will be added.

LOFAR's structure is hierarchical: 192 colocated receivers form a station; their data are locally combined. Each station produces samples from different subbands (frequency ranges), which are complex numbers representing the amplitude and phase of a wave at a particular moment. The data from all stations are centrally collected via a wide-area network, filtered, and correlated in real time on a six-rack Blue Gene/L supercomputer [14, 15]. Currently, 16 stations have been partially built, with a bandwidth limited to 500 Mb/s (48 subbands) each. More stations will be built in the years to come, and the bandwidth of many of these stations will increase to 13 Gb/s per station. Even though the computational requirements are challenging, the processing is I/O bound; therefore our BG/L system is configured with the maximum number of one I/O node to 16 compute cores (= 8 compute nodes). In total, 768 Gigabit-Ethernet (GbE) interfaces are available to receive all samples and write the correlated data.

Figure 8 illustrates how the data flow at the central processing facility, of which the BG/L system is part. A cluster of off-the-shelf PCs, called the *input cluster*, receives the data and buffers the samples for up to some tens of seconds, to synchronize the input streams and to handle short hiccups in the remainder of the processing pipeline, without losing data. Another function of the input cluster is to reorder the data over a fast (Infiniband) switching network. Since the correlator processes each subband independent of the other subbands but needs the subband data from all stations together, the data must be reordered. Each input stream contains many subbands from one station; each output stream contains one subband from many stations. The reordering step is implemented by using `MPI_Alltoallv()`. Implicitly, the input cluster converts the station data stream from UDP to TCP. The stations send UDP packets,[2] but the basic CIOD socket interface on the BG/L supports only TCP.

The reordered data flow from an input cluster node via the I/O node to the compute nodes. We carefully connected the GbE switches to all systems to avoid congestion within or between the switches. Once the data arrive on a compute core, they are filtered through a PolyPhase Filter, which splits each subband into narrow frequency channels, and are correlated. Since each compute core needs multiple seconds to process one second of real-time data, the data are round-robin distributed over the 16 compute cores in each pset: every second with sampled data goes to the next compute core. After processing, the correlated data are sent to another cluster, the *storage cluster*, and stored on disk. Here the real-time pipeline ends. After an observation, the data are calibrated and imaged.

4.1 Using ZOID in the Application

Three different application versions use ZOID. The first is a recompilation of the standard code that uses ZOID TCP sockets instead of CIOD sockets. Each compute core has one TCP connection to one of the input cluster nodes and one TCP connection to one of the storage cluster nodes. Each ZOID daemon (on an I/O node) thus transparently handles 16×2 TCP connections. The ap-

[2] TCP would require large buffers in the FPGAs at the stations and add protocol overhead, while occasionally dropped UDP packets hardly harm the data quality.

plication aligns all data to 16 bytes and sends the data in multiples of 16 bytes; this is the natural word size of the collective network. Both CIOD and ZOID require aligned data for optimal performance.

The second version runs some application-specific code on the I/O node, implemented as a ZOID plug-in, and loaded as shared object by the ZOID daemon. The application runs two threads, "scatter" and "gather," in the address space of the daemon. The scatter thread asynchronously prefetches data from an input cluster node over a (single) TCP connection into application-supplied buffers. These data are scattered over the compute cores in the pset, in round-robin order. The gather thread receives correlated data from the compute cores and optionally adds (integrates) the data from multiple compute cores together, significantly reducing the amount of data that is (asynchronously) sent over the GbE interface to the storage cluster. The first version performs the addition of data on an external computer system; but in the second version, which has the ability to run application-specific code on the I/O node, these additions are done on the I/O node. The amount of data over the collective network does not decrease, but this network is much faster (2.8 Gb/s) than the GbE interface.

The application extends the ZOID protocol with a few functions that are invoked by the compute cores. Such a function invocation is forwarded to the I/O node for execution, using ZOID's function-shipping mechanism. There are two important application-specific functions: one ships telescope data from the I/O node to one of the compute cores; the other returns correlated data from the compute core back to the I/O node. The data are automatically transferred in the output (resp. input) arguments of the function invocation. Both functions use ZOID's zero-copy protocol to achieve high bandwidth.

Running application code in the ZOID daemon process is good for performance, but a disadvantage is that a crashing application thread crashes the daemon as well. Fortunately, the process can be attached to with the gdb debugger, which has proved itself as a useful debugging tool.

Table 1. Number of subbands per pset for CIOD and ZOID.

# subbands/pset	*Mb/s in*	*Mb/s out*	*CIOD*	*ZOID*
1	160	9.5	✓	✓
2	320	19.0	✓	✓
3	480	28.5	✓	✓
4	640	38.0	✗	✓
5	800	47.5	✗	✗

Table 1 shows that ZOID outperforms CIOD. Using 16 stations, each sending 48 subbands, ZOID handles up to 4 subbands per pset in real time, while CIOD handles at most 3. For this test, the gather thread in the ZOID version does not reduce the output data rate by adding correlated data, but both the scatter and gather threads communicate asynchronously.

The third version breaks radically with the previous design and is described below.

4.2 Exploiting the Flexibility of ZOID: A New Approach

Additional stations and a prospective bandwidth growth from 500 Mb/s to 13 Gb/s per station will require significant investments in new input cluster hardware. However, the flexibility of ZeptoOS and ZOID offered a great opportunity: to omit the entire input cluster and move its functionality to the BG/L. This redesign of the central processing facility may lead to an estimated cost saving of €700,000 (US$1,000,000), excluding power and maintenance costs.

We recently implemented and experimented with a software version that bypasses the input cluster. Figure 9 shows the data

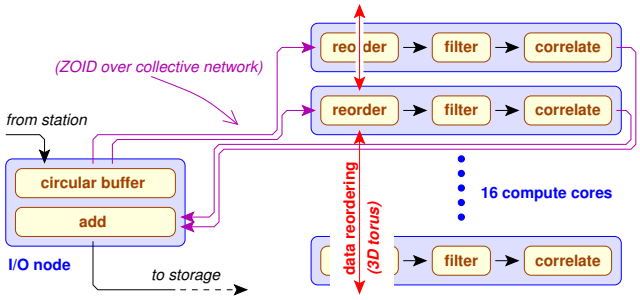

Figure 9. LOFAR data flow within a single pset.

flow within a single pset. The circular buffers are moved from the input cluster nodes to the I/O nodes. An I/O node receives UDP packets from one of the stations. It buffers the samples for up to five seconds in the circular buffer. The limited buffering capacity is due to the small amount of memory on the I/O nodes (512 MB); but thanks to the excellent real-time behavior of the BG/L, five seconds is sufficient to operate without data loss. Each I/O node receives 48 subbands from one station. From the circular buffer, the data are forwarded to the compute cores, using the ZOID protocol.

The compute cores reorder the data over the internal BG/L 3D torus instead of over the external Infiniband switch; they filter, correlate, and send the data to the storage cluster as usual. Each compute core may or may not have to handle station input, depending on whether the pset is connected to a station. Also, a compute core may or may not have to filter and correlate data, depending on whether its computing capacity is needed. Compute cores that handle input or process data (or do both) collectively perform an `MPI_Alltoallv()` with compute cores from other psets. The torus is sufficiently fast, although we had to schedule the work over the compute cores in a complex way to avoid isoplanar cores overloading some torus links. Note that input data are first transferred from an I/O node to a compute node in the same pset (over the collective network) and subsequently (over the torus) to the compute node that really processes the data. If the I/O nodes had been connected to the torus rather than to the collective network or if the collective network had had enough aggregate bandwidth throughout the entire BG/L, data could have been sent directly from the I/O node to the compute core that processes the data, omitting the extra hop.

Table 2. Current LOFAR data rates per I/O node, in Mb/s.

	Payload	*Payload + protocol overhead*
Ethernet in	481	485
collective, ION → CN	492	533
collective, CN → ION	38.0	41.2
Ethernet out	1.27	1.37

Table 2 shows data rates for the nightly observations that we perform with the 16 partial stations currently installed, observing 48 subbands and integrating correlated data over 30 seconds. We use 16 psets to receive data from the stations (one pset per station); 12 of these psets also filter and correlate data (four subbands per pset). The numbers in the table are not upper limits of what ZOID can achieve; these are numbers that ZOID handles with the current station setup. We expect higher data rates in the future. The prospective 13 Gb per station will be divided over 24 psets per station, increasing the input bandwidth to 540 Mb/s per I/O node. Additional future stations will not increase the input data rate per pset, since additional psets will be used. However, the correlated (output) data rate per GbE interface grows quadratically with the number of

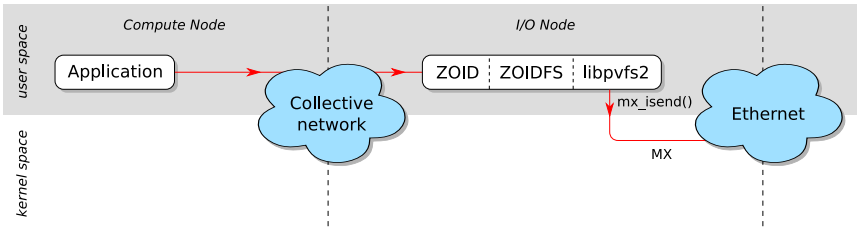

Figure 10. I/O path with ZOID, ZOIDFS, and PVFS (compare with Figure 2).

stations and is inversely proportional to the integration time of the correlated data. Depending on the eventual number of LOFAR stations and the requirements of future observation modes, we expect output data rates between 80 and 200 Mb/s per I/O node.

Since the CPU of the I/O node shares the time between the ZOID threads and the application threads, communication bandwidth is reduced if the application consumes many CPU cycles, especially since the nodes run at a low clock speed. ZOID is highly optimized; the performance of the application itself could be improved at the cost of increased complexity, but we have not needed to do so yet. In Section 5, we elaborate on future work that will improve the performance.

Removing the input cluster has many implications, both positive (e.g., reduced costs) and negative (e.g., program complexity). The advantages are likely to prevail. Omitting the input cluster would not have been possible without ZOID and ZeptoOS, because the standard operating environment (CIOD) provides neither the flexibility to run application software on I/O nodes nor the ability to receive UDP.

5. Conclusions and Future Directions

This paper introduced ZOID—a high-performance I/O-forwarding infrastructure for massively parallel systems. We discussed its design in Section 2, focusing on its extensibility through the support for custom plug-ins, as well as flexibility and high performance owing to its multithreaded daemon architecture and optimized network protocols and data paths. Performance measurements presented in Section 3 confirmed ZOID's potential but also showed the challenges. Some components ZOID depends on, such as the filesystems, could be so strongly optimized for sequential access that accessing them in parallel might significantly slow them—NFS being a prime example. Some of the problems we currently observe are simply manifestations of bugs in the components and will in time be fixed.

Performance is a moving target, especially when making comparisons with components outside of our control. When we started this effort, we could easily outperform stock IBM Blue Gene/L CIOD in a variety of situations. With time, this has become increasingly difficult, as CIOD has seen steady performance improvements. By now, most performance differences that we observe are due to inherent differences in design, not low-level optimizations.

As we mentioned earlier, only one CPU core is used by the Linux kernel on I/O nodes. That core is fast enough to drive the collective network at full capacity, or the Ethernet network, but trying to do anything else at the same time is risky. Yet ZOID encourages putting extra functionality on I/O nodes. The problem is exacerbated by the fact that the interface to the collective network is based on polling. A CPU-bound polling thread puts pressure on the Linux process scheduler and makes the scheduling of other threads unpredictable. While using real-time priorities would address this issue, it would raise its own problems. We are currently working on enabling the second CPU core and using it exclusively for the communication over the collective network. This would free up the much needed CPU resources and make more advanced optimizations viable.

Section 4 showed how ZOID is used in a real application: the central correlator of the LOFAR radio telescope, which processes real-time, streaming data on a Blue Gene/L system. Initially, ZOID was used just as a high-bandwidth replacement for CIOD. However, ZOID's ability to run application code on the I/O nodes led to a redesign of the entire LOFAR data processing system that significantly reduces the costs, by omitting a dedicated input cluster and by moving its functionality to the Blue Gene/L. This clearly shows that flexible and extensible ZOID provides novel solutions to problems of real systems.

More ZOID plug-ins than the two discussed in this paper are under development. ZOID was in fact conceived for use not with the UNIX or LOFAR plug-ins but with ZOIDFS—an interface far better suited for high-performance parallel file I/O than is the POSIX API. ZOIDFS was designed in collaboration with other U.S. DOE laboratories to be easy to integrate with MPI-IO and parallel filesystems: its stateless nature allows it to scale better and makes various optimizations simpler. Thanks to the user-space nature of ZOID, a number of shortcuts can be taken, especially if the client part of the filesystem code can be implemented in user space, as is the case with PVFS. Figure 10 shows the infrastructure we are building for the upcoming Argonne IBM Blue Gene/P machine. Compared to the standard IBM infrastructure (see Figure 2), it has significantly fewer context switches between the user and kernel space, increasing throughput and reducing latency. ZOIDFS will have filesystem-specific implementations. We have been working on a version for PVFS, and we plan to create a generic implementation on top of the POSIX API. Adding modules tuned for other parallel filesystems should be straightforward. With ZOIDFS, we expect to be able to investigate I/O scalability in ways not previously possible. Once an initial implementation is finished, we will test it with a suite of I/O-intensive applications; we intend to publish the results in a subsequent paper.

Another plug-in, *ftb_agent*, is under development within the CIFTS [7] project. That plug-in enables compute node processes to be part of a fault tolerance backplane being developed, providing an interface for throwing and catching fault tolerance events. Fault tolerance is currently not built into ZOID. Static assignment of psets to I/O nodes complicates a failover, should an I/O node fail for any reason. However, the collective network does have links that join the individual psets, so traffic could in principle be rerouted to an alternative I/O node if necessary. Adding fault tolerance to I/O forwarding would require changes to many Blue Gene software components, including the monitoring system, which currently forces an immediate termination of the complete job if it loses connection to any of the associated I/O daemons.

We expect intelligent caching and "collective behavior" to be extremely important with file I/O in order to reduce the load on the fileservers. For many applications, using MPI-IO coupled with the aforementioned ZOIDFS may be the solution of choice. However, we would like to investigate performing similar optimizations with POSIX file I/O, so that more applications can take advantage of

them. Many file I/O calls, such as `open()` or `read()`, could easily be implemented as collective operations. An intelligent ZOID plug-in, in cooperation with the application, could forward only the first of the many individual requests to the filesystem and broadcast the result to every process interested in it. This would reduce the total number of filesystem requests to the number of I/O nodes. If I/O nodes could work cooperatively, they could do even better: large read requests could be split between multiple I/O nodes, a global cache layer could be created, and so forth.

IBM promises to make the Blue Gene/P a more transparent platform than BG/L, with open interfaces between compute nodes, I/O nodes, and the service node. While exposing the inner workings of the existing infrastructure is desirable, it won't make extending that infrastructure significantly simpler. We are working on a full integration of ZOID with the job management system, so that CIOD will never even need to be started. Unlike on BG/L, where ZOID cooperates with the CNK on the compute nodes, on BG/P we plan to have ZOID working with the ZeptoOS Linux kernel on the compute nodes, which is another component we are developing. ZOID on BG/P will take care of system control tasks such as job launching and will perform other "convenience" tasks, such as exporting filesystems from the I/O nodes to compute nodes using FUSE or even forwarding IP traffic, enabling one to, for example, log on a compute node with ssh to debug a problem.

As mentioned, Cray uses a different approach to file I/O on the XT, with filesystem clients on each compute node. However, the architecture does feature I/O nodes. They are currently not used for I/O forwarding, but it appears that they could be used for this purpose if so desired. Hence, we plan to experiment with I/O-forwarding setup on that architecture. ZOID requires user-space access to network devices, which Cray provides through the low-level Portals. ZOID also requires some means of redirecting file I/O operations. Cray currently uses SYSIO for that purpose. ZOID could do the same, or it could use a replacement `libc` as on BG/L, or, with Linux running on compute nodes, a custom FUSE filesystem client could be used.

Acknowledgments: We acknowledge the students who contributed to ZOID: Ivan Beschastnikh, Peter Boonstoppel, Hajime Fujita, Jason Kotenko, and Alex Nagelberg. We also thank Chris Broekema, Ger van Diepen, Martin Gels, Marcel Loose, Ellen van Meijeren, Ruud Overeem, Kjeld van der Schaaf, and Walther Zwart for their contributions to the LOFAR software.

References

[1] ASC Purple. http://www.llnl.gov/asc/computing_resources/purple/.

[2] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Proceedings of the 8th IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sept. 2006.

[3] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. Operating system issues for petascale systems. *ACM SIGOPS Operating Systems Review*, 40(2):29–33, Apr. 2006.

[4] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Computing*, 2008. Accepted.

[5] H. R. Butcher. LOFAR: First of a new generation of radio telescopes. *Proceedings of the SPIE*, 5489:537–544, Oct. 2004.

[6] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, Oct. 2000.

[7] CIFTS: Coordinated fault tolerance for high performance computing. http://www.mcs.anl.gov/research/cifts/.

[8] Cray XT3. http://www.cray.com/products/xt3/.

[9] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 47th Cray User Group Conference*, Albuquerque, NM, May 2005.

[10] Lustre. http://www.lustre.org/.

[11] J. E. Moreira et al. Designing a highly-scalable operating system: The Blue Gene/L story. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, Tampa, FL, Nov. 2006.

[12] J. A. Rathkopf et al. KULL: LLNL's ASCI inertial confinement fusions simulation code. In *Physor 2000, ANS Topical Meeting on Advances in Reactor Physics and Mathematics and Computation into the Next Millennium*, May 2000.

[13] J. J. Ritsko, I. Ames, S. I. Raider, and J. H. Robinson, editors. *Blue Gene*, volume 49 of *IBM Journal of Research and Development*. IBM Corporation, March/May 2005.

[14] J. W. Romein, P. C. Broekema, E. van Meijeren, K. van der Schaaf, and W. H. Zwart. Astronomical real-time streaming signal processing on a Blue Gene/L supercomputer. In *Proceedings of the 18th ACM Symposium on Parallel Algorithms and Architectures*, pages 59–66, Cambridge, MA, July 2006.

[15] K. van der Schaaf, C. Broekema, G. van Diepen, and E. van Meijeren. The LOFAR central processing facility architecture. *Experimental Astronomy*, 17:43–58, 2004.

[16] The ZeptoOS project. http://www.zeptoos.org/.