Bringsel: A Tool for Measuring Storage System Reliability, Uniformity, Performance and Scalability

John Kaitschusk, *Cray, Inc.*, and **Matthew O'Keefe**, *Alvarri, Inc.*

ABSTRACT: *Bringsel is a primary I/O testing program that enables the use of either POSIX or MPIIO calls to perform benchmarking and evaluation to measure the reliability, uniformity, performance and scalability of file systems and storage technologies. It enables the creation of a large number of directories and files using both a threading model (POSIX) and the MPI library for multiple nodes to coordinate testing activity. Bringsel has run on a variety of large scale computing platforms, including Cray XTs, SGI Origin systems, Sun enterprise-scale SMP systems and Linux clusters.*

KEYWORDS: Storage Benchmarking, Reliability, Uniformity, Performance, Scalability, Storage Evaluation

Introduction

In storage systems evaluation there has been a proliferation of tools and applications to help access performance of various file systems, storage equipment and generalized configurations over the years. Examples of such benchmarking software include IOzone[1], IOR[2], FileBench[3], xdd[4], and PRIOmark[5]. In the fall of 1998, one of this papers' authors, Kaitschuck, was evaluating storage technologies as a researcher for a northern California technical computing installation. At that time, a small project was started to develop a benchmarking tool which would allow for both reliability and performance testing in a controlled environment. It would evaluate various vendor technologies under consideration. The characteristics of storage at this California facility included a mix of file sizes and access types within very large directory structures. The tool that was developed, Bringsel, is an I/O benchmark that accounts for all associated operational constraints and generalized I/O workloads typical of a large scale technical computing environment.

Since then Bringsel has undergone several enhancements, including general modifications and extensions. It has been used to test file systems and storage technologies on a variety of operating systems including Irix, AIX, Solaris, Unicos, Unicos/MK, Linux and BSD. It has run on a variety of hardware architectures, including PVP based systems, scalar SMPs, MPPs and large clusters. It has been used at different sites to help understand performance issues, and as a general-purpose diagnostic to isolate faults on storage systems. Because of its exposure to so many operating systems and platforms Bringsel has evolved into a useful general purpose I/O benchmark for technical computing.

[1] IOzone File System Benchmark: www.iozone.org

[2] R. Hedges, B. Loewe, T. McLarty and C. Morrone. Parallel File System Testing for the Lunatic Fringe: the care and feeding of restless I/O Power Users. Proceedings of the 22nd IEEE/13th NASA Goddard Conference (MSST 2005).

[3] Filebench File System Benchmark: www.solarisinternals.com/si/tools/filebench/index.php

[4] `xdd` File System Benchmark: www.ioperformance.com

[5] M. Krietemeyer, D. Versick and D. Tavangarian. The PRIOmark Parallel I/O¬Benchmark: www.ipacs-benchmark.org/index.php?s=download&unterseite=priomark

2. Testing Focus

The objective behind the use of any I/O benchmark is important to consider before it should be used. The Bringsel benchmark assesses the metrics that go to the *quality* of I/O subsystems. These include:

- **Service**: Provide the required API into the storage subsystem for the given application set, POSIX, MPI-IO, and so on. Other service-related items should be are to be considered, such as documentation, and manageability, but Bringsel can not test these.
- **Reliability**: *n* bits of data and (and metadata) written into a storage subsystem should return *n* bits on retrieval. Storage content should not be changed by external load, access frequency (unless intended), or over time. Sites often define reliability only in terms of MTTI or MTBF, as these metrics quantify system uptime. Reliability however must measure not only uptime, but data access and data integrity (uncorrupted data).
- **Uniformity**: Given a general level of performance under load *x* during period *t*, the subsystem should provide an equivalent level of performance within a given delta, at time $t + 1$. Equivalent performance should vary by no more then 20% at $t + 1$, in consideration of the underlying technology/architecture in relationship to the work load. Uniformity can consider factors based on time and/or capacity.
- **Performance**: Provide high levels of performance to meet or exceed latency and bandwidth requirements for applications and access to storage. Many sites consider some performance aspect in relationship to system size, commonly raw I/O bandwidth.
- **Scalability**: Provide specifications for reliability, uniformity and performance at the size required for the system.

While these are categories and not specifics, they apply to most storage subsystems. Requirements may differ for individual sites and applications, but principal quality concerns remain. Requirements vary according to the application(s) run, a site's hardware and software configuration, usage for production versus research and development, and the user load. The Bringsel feature set tries to test as many of these variables as possible, while remaining portable and easy to implement yet with advanced features and optimization capabilities.

3. Bringsel Features

Bringsel is a C-based program that runs in user space, unlike similar programs such as trace[6], or explode[7]. While user space does present some challenges, notably with timing (see buttress[8]), multi-core CPUs have ameliorated these problems to a great extent. Bringsel provides a wide range of features, including:

- symmetric directory tree creation, including both serial and parallel `mkdirs` and `stats` to confirm success in operations.
- a modified hash tree directory scan and integrity check option.
- sequential and random directory tree walks, including symbolic link, file and directory counts.
- multiple API support, including POSIX, file streams, MMAP and MPI-IO.
- multi-node support, via MPI, for MPP and cluster systems, with selectable node delays.
- POSIX thread support, with selectable inter-thread delays, where applicable for operating environment support.
- multiple operation type and block size selection on a per-thread basis.
- a selectable file `checksum` option, via Haval.
- file attribute selection and checking; permissions and ownership.
- iteration and looping support.
- file option support; append, truncate, fsyncs, etc.
- support for a configuration file parser and command line options.
- performance measurements for open/close latency, latency loops, data bandwidth and IOPs.
- performance measurements for directory traversal.
- selectable raw ASCII output, either directly to `tty` or a logging file.

---

[6] M. Mesnier, M Wachs, R. Sambasivan, J. Lopez, J. Hendricks, G. Ganger and D. O'Hallaron //TRACE: Parallel trace replay with approximate causal events. USENIX/FAST07, San Jose, CA. February 2007.

[7] J. Yang, P. Twohey, B. Pfaff, C. Sar and D. Engler. EXPLODE: A Lightweight, General Approach to Finding Serious Errors in Storage Systems USENIX/FAST07, San Jose, CA. February 2007.

[8] E. Anderson, M. Kallahalla, M. Uysal and R. Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. USENIX/FAST04, San Francisco, CA. March 2004.

To explore details of some of these features, we shall examine a subset related to both their implementation and use within Bringsel.

3.1 Options and Configuration Files

Bringsel can take either single command line options or multiple command line options in a configuration file as input. Placing multiple command line options in a configuration file allows multiple iterative passes with standardized test configurations which specify activities such as sequential or random access, directory structures, file sizes and metadata activity.

The following example shows a very simple two-line configuration file: `sample.cnf`

```
#
# Comments start with '#'
#
-T 4 D /snarf/foo:1,2,2 M L c b 32 S 100M alpha
-T 4 a sx D /snarf/foo:1,2,2 L
```

Example 1: Sample Bringsel Configuration File

Execution of Bringsel with the configuration file as the primary input takes place via a simple invocation, in this case: `$ bringsel C sample.cnf`

Use of a configuration file causes the file to be read and any lines starting with '#' to be ignored as general comment lines. The line which starts with `T 4 D /snarf/foo:1,2,2` [....] will cause a series of operations to occur, starting with directory tree creation.

3.2 Symmetric Directory Tree Creation

A portion of the configuration line '`T 4 D /snarf/foo:1,2,2` [….] causes a small symmetric directory tree to be created using four POSIX threads.

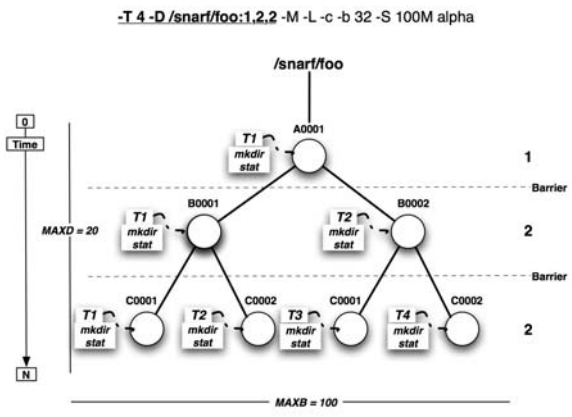

Figure 1 Directory Tree Structure

The directory structure shown in Figure 1 starts with a single top level node under the mount point `/snarf/foo`. Additional lower-level nodes are created in parallel, on each respective level. Barriers exist between levels to allow for top level directories to be completed before the next level is started. The basic operational pass is a `mkdir` followed by a `stat` call to allow for an immediate degree of integrity checking. It is also possible to create the directory structure with a serial-only pass via a `d` directive instead of a `D`. Maximum directory depth for Bringsel is currently 20 levels, while the maximum single level width is 100 nodes. Once the directory structure is created, the next step is to perform a latency loop measurement pass within the structure.

3.3 Latency Loop Measurements

Bringsel can perform two latency measurements. The first measurement is the *selectable latency loop*. This measurement is enabled by passing in an `M` through either a direct command line option or configuration file options. Within the targeted directory structure each active thread creates a number of temporary files that undergo a series of system calls. These include `stat`, `mkdir`, `chmod`, and `utime`.

The loop is iterated across to run a series of these calls against the generated temporary names.

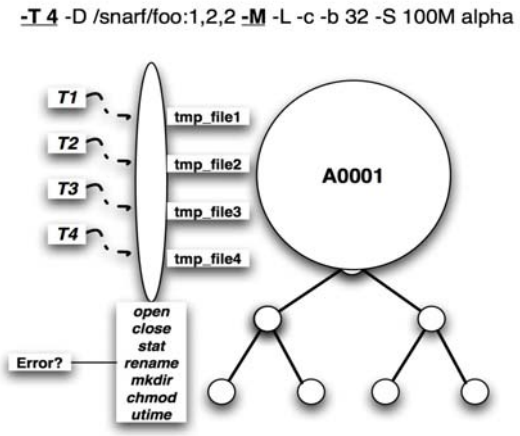

Figure 2: Loop Return Time Measurements

As each system call is made within the loop, return values are checked to flag error conditions. Once each loop is completed, the temporary files and directories are removed from the directory structure. The time measurements for each pass of these latency loops are collected into the respective thread data structures. Each directory within the structure is opened by each thread and the sequence is re-executed in a top / breadth-first order.

Bringsel will then create the permanent files within the directory structure.

3.4 File Creation

Bringsel, as has been mentioned, supports multiple API's for file operations. The default operation uses standard POSIX `libc` calls for file support. In the case of our example configuration file (Section 3.1), the file operations will use the default API to access the file system. As each thread will create an individual file, this is a non-parallel access case.

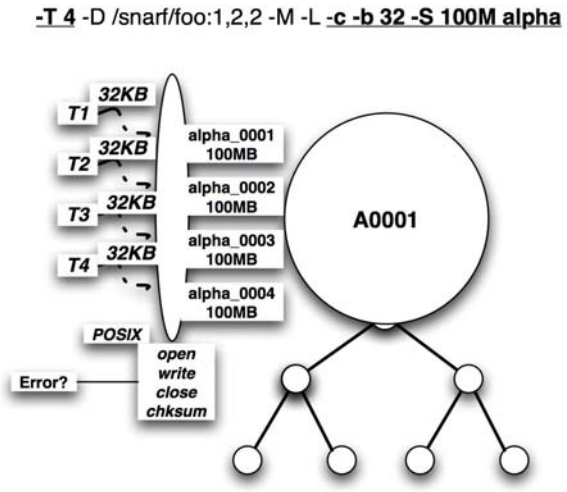

Figure 3: Separate Files Created per Thread

As shown here, each thread creates a separate file named `alpha_`*nnnn*, where *nnnn* is a thread-level designator. Each thread writes 32 KB blocks to create files that are 100 MBs in size. Once the file has been written out to the storage subsystem it is closed. Then an optional checksum is performed on each file to verify the contents of the storage subsystem. The checksum option uses the Haval[9] algorithm. Here all files are, by default, zero-filled, but other fill patterns can be selected by the user. Timing information and general performance data is stored in the associated thread level data structure, once full operations are completed.

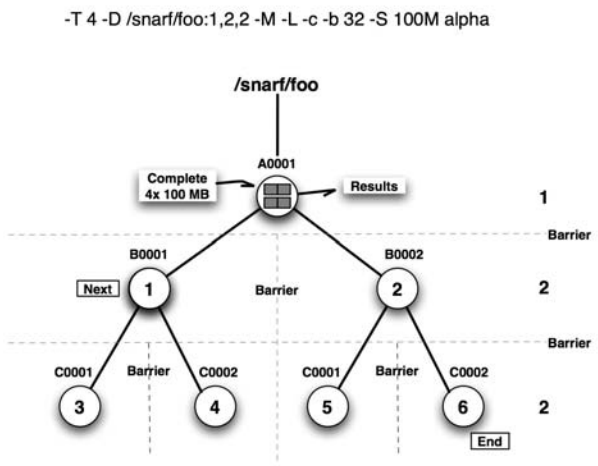

Figure 4: Directory Traversal

File creation in this case progresses much as it did for the latency loop measurement – top / breadth-first. As the results are produced across the structure of larger directory structures, the performance data allows for a better understanding of uniformity as related to decreasing spatial capacity.

At the end of this process we end up with 7 directories. Each contains 4 x 100 MB files for a total of 28 files, consuming approximately 2.8 Gigabytes of space. Obviously this is only a small example of the size and scope of the directory structure and file contents that can be created using Bringsel. It would also be possible to use Bringsel to set some of the file attributes during creation via an `m` directive. A simple example would be `-m +:USER:GROUP:0755` as an addition to what has been shown as the first configuration file entry.

In general-purpose file systems, the ability to stat file system contents, directories and files is a well known issue[10]Since Bringsel is intended to test a wide range of storage subsystems technology, directory walks are essential to the overall functionality of the program. The second active line of the configuration file demonstrates the ability to perform sequential directory walks across the structure we have created. These operations, file creations and directory walks can be combined in more complex operational cases to explore the storage subsystem under examination.

3.5 Directory Walks

The second active line from the configuration file establishes a sequential tree walk by all four threads. The

[9] Y. Zheng, J. Pieprzyk and J. Seberry. HAVAL – A OneWay Hashing Algorithm with Variable Length of Output. Advances in Cryptology – Auscrypt'92 Lecture Notes in Computer Science, Vol. 718 SpringerVerlag, 1993. pp. 83–104.

[10] D. Roselli, J. Lorch and T. Anderson. A Comparison of File System Workloads. Proceedings of the 2000 USENIX Annual Technical Conference San Diego, California, USA June 1823, 2000.

threads start at the very top of the generated directory structure and stat the contents of each directory. As each thread walks down the directory tree it keeps a count of directories, files and any symbolic links encountered in each respective directory.

Bringsel can also perform random walks: `a rx`. Here the starting node is randomly selected from the indicated directory structure and varied with each thread.

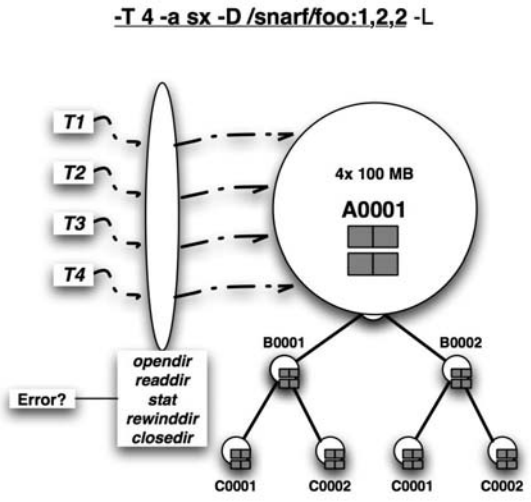

Figure 5: Random Directory Walk

The total time to stat the directory structure is recorded and saved in the thread level data structure. Given an existing directory structure and the associated file contents, Bringsel can perform a series of intermixed operations, for example random writes and sequential walks, at any time. Intermixed operations can span different block issue sizes, from under 1 KB up to 128 MB. A simple example would be a variation of the first line from our configuration file:

```
T 4 a rw,sx D /snarf/foo:1,2,2 L c b 32 S
100M alpha
```

This dispatches two threads to perform random writes on their designated files within the created directory structure, while the remaining two threads perform a sequential walk of the existing tree. The subdivision of thread-level activity in all of these examples has been within a single system. In the case of an MPP or cluster system, Bringsel tracks the *Global Thread Number* (GTN) of each thread as it relates to the node it is running on. Work is then allocated across this global collection of threads for dispatch from the various nodes. Despite these features, there is still a challenge to maintain state across any structure created, along with the associated contents. In the case of really large structures with different file contents and sizes, we need to be certain of storage subsystem integrity when performing any testing.

3.6 Hash Trees and Directories

In order to verify the integrity of large directory structures and their file contents, we must confirm the generalized structure over time. This provides a reliable way to verify storage subsystem functions such as backup, replication and snapshots. Using hash trees, or modified Merkle trees[11,12], Bringsel can scan and check the created data structures. This option, combined with the Haval checksum to verify file contents, allows for full scale integrity checking.

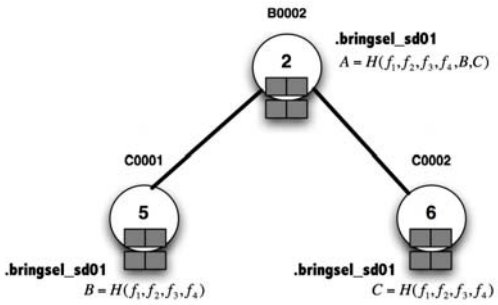

Figure 6: Hash Tree Formulation

Figure 6 shows a portion of the directory structure created earlier. Tree scanning to create the required state information on the structure starts at the bottom and works in a breadth-first manner. Individual directories are scanned for all included files and subdirectories. The basic formulation of this activity takes the form:

```
V = H(f1 → fn,D1 → Dn)
```

Where the function `H()` is the SHA256 checksum[13]. The values for `f` files in the directory, take on the value of the fully qualified name, excluding the user-specified mount point, UID/GID, and file permissions. Any subdirectories have their associated `V` value from their storage file `.bringsel sd01` included in this transformation. The value `V` is then stored in a file `.bringsel sd01` within the current directory. Directory checking following a snapshot, restore or migration can then compare com-

[11] R. Merkle Secrecy Authentication and Public Key Systems. UMI Research Press, 1982. Also appears as a Stanford Ph.D. thesis in 1979

[12] M. Jakobsson, T. Leighton, S. Micali and M. Szydlo. Fractal Merkle Tree Representation and Traversal. RSA Cryptographers Track, 2003.

[13] Secure Hash Standard (SHS). Technical Report FIPS PUB 1802, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 208998900, August 2002.

puted hash values to stored values in each respective file at a directory level. Directory checking, like the scanning operation, takes place in a bottom-up, breadth-first fashion. The time required to complete a directory scan varies with the number of files and the size of the directory tree. The use of SHA256, while more computationally intensive then other checksum methods, provides a minimum probability of nonviable collisions within the structure.

3.7 Log Output

ASCII log output from Bringsel can be sent to a log file or directly to the screen. Bringsel's primary output is composed of two field types associated with each active thread for a given command sequence.

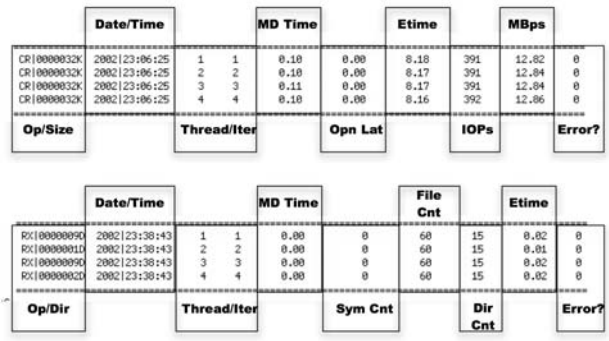

Figure 7: ASCII Log Output

The first field type, shown above, is the active operation type. This includes file creations, sequential writes, sequential reads, random writes and random reads. Here all operations are file creations *CR*. Further columns include:

- block size
- date and time of thread execution
- thread and iteration number
- metadata loop performance measurement
- general open latency measurement
- total elapse time
- IOPs
- MBps
- a general error checksum.

The second field type for directory walks (in this case, random walks *RX*), include:

- date and time of thread execution
- thread and iteration number
- metadata loop performance measurement
- symbolic link count
- file count
- directory count
- elapse time
- general error checksum.

Part of the general output (not shown in Figure 7) is a generalized header which includes version number and selected options for the given run. In addition, log output can include a *Hash IDentifier* (HID) for the individual directories and files. Since the volume of output can be substantial with large directory structures and file counts, we have created a series of short `awk` scripts that aid in data post-processing to an external graphing package. This allows for the production of 2D and 3D graphs using a commercial based package like Excel or DeltaGraph. While such commercial packages are adequate for general use, larger scale structures and files will require a more substantial data handling and visualization capability.

4.0 Sample Results

Bringsel has most recently been used in the evaluation of NAS based storage within Cray.[14], see [14]. Some examples of processed results include a uniformity run across various block sizes with 20 nodes:

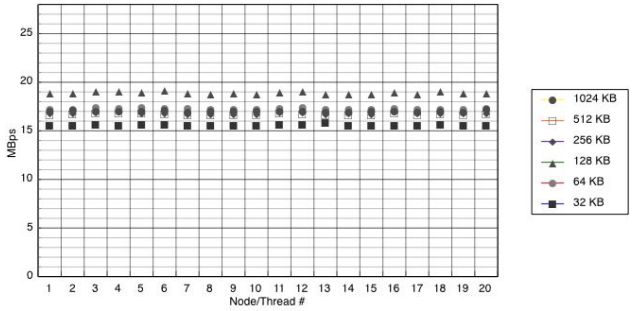

Figure 8: Uniformity Run Across 20 Nodes

[14] J. Kaitschuck, J. Reaney, C. Hertel and M. O'Keefe Performance, Reliability, and Operational Issues for High Performance NAS Storage on Cray Platforms. *CUG Proceedings 2007*. Seattle WA

An aggregate bandwidth run using 24 nodes and two FC RAID array controllers on the back end storage:

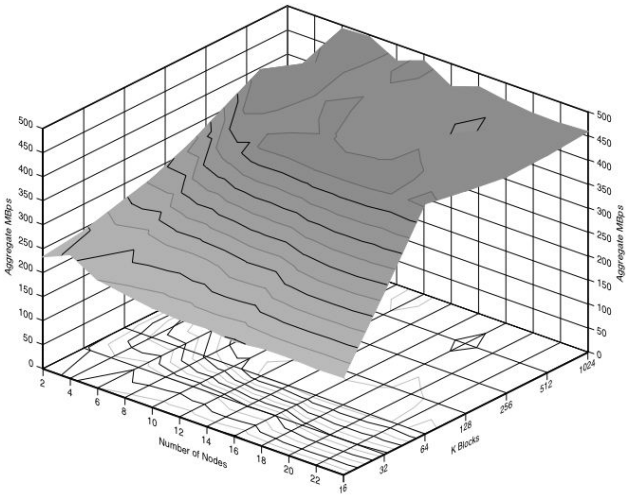

Figure 9: Aggregate BW Run - 24 nodes, 2 FC RAID Array Controllers

5.0 Future Work

Bringsel has some unique features. Possible future additions to this feature set include:

- UPC support for larger shared memory systems and going beyond the current MPI-only multi-node system support.
- Directory tree limits. Currently Bringsel uses full syntax directory support, as shown earlier. Directory tree limits would allow selection of individual subdirectories within large trees for given operations.
- Adding and pruning directories in compact form. This would provide the capability to add and remove directories subsets of entire directory trees.
- Modules to support tracing input/output. Numerous technologies have been developed to provide application-level file traces. Bringsel could be extended to support some of these tracing tools/libraries.
- Better visualization methods via an external application. In order to handle very large directory structures, it would be useful to provide a way to support interactive 2D and 3D exploration of the large data sets that Bringsel can generate.
- External automated test driver. This is to help accelerate testing, via configuration file generation based on output logs from initial Bringsel runs.

6.0 Conclusion

Bringsel provides a diverse and flexible tool for benchmarking file systems and storage subsystems. It allows data gathering for the evaluation of service-specific APIs, reliability, uniformity, performance and scalability. It can create a range from a single file to large directory structures populated by files of differing sizes. It provides the most common APIs and access types for file access and metadata manipulation. These various access types can be intermixed using a POSIX threading model, and multi-node support to provide a useful, mixed I/O benchmark, which also provides advanced integrity checking methods.

About the Authors

John Kaitschuck is currently a Senior Systems Engineer with Cray Federal. He has previously served in a variety of technical and consulting positions in industry and government as both analyst and developer. He has worked with a wide range of HPC issues around systems and system software. He can be reached at jkaitsch@cray.com.

Matthew O'Keefe is a founder and Vice-President of Engineering at Alvarri Inc., a start-up focusing on storage management software. Previously, Matthew founded Sistina Software, sold to Red hat in late 2003; he spent 10 years as a tenured Professor at the University of Minnesota, where he is currently a Research Associate Professor. He can be reached at okeefe@alvarri.com.