Detailed Analysis of I/O traces for large scale applications

N. Nakka, A. Choudhary, W. K. Liao
Electrical Engineering and Computer Science
Northwestern University, Evanston, IL, USA.
{nakka, choudhar,wkliao}@eecs.northwestern.edu

L. Ward, R. Klundt, M. I. Weston
Sandia National Laboratory
Albuquerque, New Mexico, USA.
{lee,rklundt,miwesto}@sandia.gov

***Abstract* - In this paper, we present a tool to extract I/O traces from very large applications running at full scale during their production runs. We analyze these traces to gain information about the application. We analyze the traces of three applications. The analysis showed that the I/O traces reveal much information about the application even without access to the source code. In particular, these I/O traces provide multiple indications towards the algorithmic nature of the application by observing the changes of data amount and I/O request distribution at the checkpoints. Adaptive Mesh Refinement (AMR) is one of the kind of algorithms that can exhibit such I/O behavior. This is the first study of I/O characteristics of unbalanced AMR-supported applications at scale. The key observations that we made in the trace were (1) Variation in aggregate data sizes across checkpoints for AMR and non-AMR applications, (2) Variation in the number of I/O calls by a client depending on the nature of the application, (3) Use of temporary files by applications and possible erroneous calls to I/O functions, (4) Variation in average data transfer size according as whether the application has AMR support or not, (5) Aggregation of I/O for processes executing on a single physical node through MPI-IO calls, and (6) Updates to specific data structures in the checkpoint file.**

*Keywords:*Large scale I/O tracing, I/O trace analysis, adaptive mesh refinement

I. INTRODUCTION

Tracing I/O access patterns on large supercomputing clusters has always been challenging. These challenges are found in developing an efficient and light weight tracing solution that does not add significant overhead in terms of memory and processor time to the traced application. Researchers from LANL have provided a comprehensive survey of the currently available I/O tracing techniques [1]. Keeping these constraints in mind, researchers at Sandia National Laboratories have developed an I/O tracing framework for light-weight tracing of large scale applications on Catamount systems. I/O traces are a valuable source of information for debugging distributed applications and as guidance for I/O benchmark development. This paper demonstrates the first analysis of traces of large scale applications in production. These I/O traces provide interesting insights to the nature of the traced application, without viewing the source code of the application. The goal was not to automate the analysis process but rather to understand how valuable the trace information was in inferring application behavior.

II. RELATED WORK

Using the taxonomy developed in [1] three differing I/O tracing tools LANL-Trace [2], Tracefs [3], and //TRACE [4] have been evaluated. LANL-Trace adds a high overhead to the traced application, prohibiting it from being used at scale for real-world complex applications. Besides this, it does not support anonymization for distribution of the traces. Tracefs has advanced tracing features but has a high installation overhead. Particularly on parallel file systems, which are the subject of the current study. //TRACE has been recommended for replaying file system traces but also does not support anonymization. Huang et. al. [5] developed a high resolution disk I/O trace system built into the Linux operating system but does not collect information about the driving IO API call. Carey et. al. [6] and Cattell et. al. [7] provided I/O tracing mechanisms for large database systems, an application that is very different from even the most casual single program, multiple data application. Ramakrishnan et. al. [8] analyzed I/O traces in commercial computing environments to understand file access behavior. They showed that a relatively small fraction of files are active and studied the dynamic sharing of files but did not analyze distributed applications. Ousterhout et. al. [9] performed a trace-driven analysis of the UNIX BSD 4.2 file system and found some interesting conclusions on usage of files on the file system but, again, were not concerned with distributed applications. Howard et. al. [10] improved the performance of the Andrew file system using observations made on a prototype implementation. Joukov et. al. [11] designed and implemented Replayfs to replay the file system traces at the VFS level. Stardust [12] examines interactions between the client and server – this is it's "end-to-end" claim, and it's strength. We wished to capture application interaction with the VFS. i.e., at the system call boundary and not request flow across the network. The client operating system has the opportunity to reformat the application calls in a file systems specific way. Stardust instrumented NFS and, so, the Stardust traces will show read/write calls no larger than the maximum payload size for the protocol. Our tool is independent of such specifics, recoding the actual call, response,

978-1-4244-4921-7/09/$25.00 ©2009 IEEE

and timing as given by and from the perspective of the application at the "system call" boundary.

These traces are roughly the equivalent of capturing the Windows32 IO API calls described in Magpie [13]. There did not exist such a tool on Catamount. The equivalent, under Linux, is strace(1). strace is too slow (milliseconds of overhead per call) to preclude tracing coupled applications at scale. We do not have the resources to implement the Magpie system on a large supercomputer; the instrumentation points are not there.

III. Description of Trace Utility [14]

It is of great benefit and interest to system developers and administrators to acquire good understanding of usage modes on large scale machines. To address the lack of such a tool researchers at the Sandia National Laboratory have implemented a tracing utility that is meant to be used with the Catamount Lightweight Kernel (LWK) [15]. The tracing utility, along with the LWK and the application run on Red Storm, a Cray XT3+ capability class machine[1]. By itself Catamount does not provide directly user accessible I/O capabilities as it is a custom microkernel. Application I/O is managed by the inclusion of user level library sysio[2]. This library provides a virtual file system implementation which allows the application simultaneous access to various file systems. Each sysio library call provides a hook at entry and exit. The tracing needs to be initialized through a function call at the beginning of the program. Otherwise, the hook does nothing.

When the tracing functionality is activated, traversal of an entry or exit hook triggers an event, which consists of a call into the tracing code, passing a record of qualifying information for the call. Each event results in the encoding of the call, type of hook, and the qualifying information into a buffer in the tracing code. The design includes double buffering and use of asynchronous write calls to dump a buffer when full[3].

The tracing utility traces only I/O calls of the application interacting with the file system. File system interactions of the tracing utility itself are not traced. On detection of internal error conditions the tracing halts, and allows the application to continue without interruption if possible.

Each process in the application job generates one file containing the I/O traces encoded in an efficient, platform independent, binary format. A dictionary describing the binary file format must be generated using a provided external utility on the host platform. The resulting traces can be decoded into readable format by using the dictionary and the provided binary decoder on any machine.

A. Details of the Trace Output Format

Each process in the parallel job generates a set of trace events. The final translated output is the concatenation of all the trace sets, and is in human readable format. Note that the trace can contain the data transfers of the client (process) across multiple checkpoints (commonly also known as restart dumps) of the application. Each set of per-client trace events begins with a header line with the following format (bold indicates keywords):

header*(headerbom,*_**headernode**_*(node_nid.pid)*_**headerlength**_*(<bytes>))*

In this trace event the node on which the trace events are generated is uniquely identified by the pair nodeid.processid.

All other contents of the final output have the following format:

tracetype*(*_**ENTER**_*|*_**EXIT**_*)*_**time**_*:(secs)*_**time**_*:(msecs)*_**str**_*(<event_name>)<infolist>*

Each system call which occurs in the traces will have an Enter/Exit pair present. The time fields above denote seconds and microseconds since the Epoch (00:00:00 UTC, January 1, 1970) as returned by the gettimeofday call.

The *<infolist>* contains a hierarchically arranged collection of qualifying information for the particular I/O system call and varies according to the call.

A simple example is the event for entry to the open syscall. The trace event contains, after the timestamp, the name of the call, a string with a sanitized version of the pathname, and the incoming values for open flags and mode.

tracetype(ENTER)***time***:(1205528630)***time***:(529780)***str***(open)***str***("filename")***flags***(578)***mode***(436)

A more complex example is the stat call, which resolves to a call to fxstat within the sysio library. Here are the Enter and Exit events for that call:

tracetype(ENTER)***time***:(125528630)***time***:(531185)***str***(fxstat)***ver***(1)***fd***(3)

tracetype(EXIT)***time***:(125528630)***time***:(531368)***str***(fxstat)***return***(0)***stat***(st_dev(0)st_ino(10702772)st_mode(33204)st_nlink(1)gid(41776)uid(41776)st_rdev(0)st_size(0)st_atime(1205528558)st_mtime(1205528630)st_ctime(1205528630)st_blksize(2097152)st_blocks(0))

The Enter event for fxstat provides the incoming values of the version and file descriptor. The Exit event provides a return code, and the contents of the stat struct items which have been acquired by the call. The item names are taken from the stat struct definition of the machine where the traces have been generated.

---

1 http://cray.com/products/xt4/index.html

2 http://sourceforge.net/projects/libsysio

3 Asynchronous behavior is dependent on the capability of the underlying file system where the trace data is written.

The call names in the trace events reflect the actual API call within the sysio library where the trace event is being recorded. Generally these follow the POSIX[4] definitions and should be self-explanatory.

Some calls are specific to sysio. For example, all I/O transfers in the released traces are implemented at the lowest level with asynchronous versions of, usually familiar, calls, and iowait/iodone[5]. The ireadv and iwritev trace events, for instance, provide information identifying the file descriptor, the pointer to an I/O vector, and the length requested for the transfer. The request is queued, to be completed asynchronously if possible. The iodone routine is used to poll for completion, but is not traced. The return code in the iowait Exit event reports the number of bytes transferred. Here is an example of the sequence of trace events associated with a write call:

```
tracetype(ENTER)time:(125528630)time:(586386)str(iwritev)fd(3)ziovec(base(0x200EFDC0)len(524288))
tracetype(EXIT)time:(125528630)time:(590489)str(iwritev)ptr(0x200B1110)
tracetype(ENTER)time:(125528630)time:(590490)str(iowait)ptr(0x200B1110)
tracetype(EXIT)time:(125528630)time:(590530)str(iowait)return(524288)
```

Also provided are a set of asynchronous vector I/O calls which perform extent based transfers. Both ireadx and iwritex calls take as input a file descriptor, a list and count of memory specifications (struct iovec *) and a list and count of extent specifications (struct xtvec *). The return value is, as for all data transfer calls, a transaction identifer ioid_t. The extent specifications are (offset, length) pairs and define the locations in the file involved in the data transfer, as follows:

```
struct xtvec {
  off_t xtv_off; // Stride/Extent offset.
  size_t xtv_len;// Stride/Extent length.
};
```

The ireadx/writex calls reconcile the memory specification list and file extent list in order, performing transfers as directed, until one or the other of the lists is exhausted. Here is an example of the trace events produced by iwritex and corresponding iowaits:

```
tracetype(ENTER)time:(1216152765)time:(250413)str(iwritex)fd(1)count(4)ziovec(base(0x537D50)len(10),base(0x538590)len(10),base(0x537D70)len(10),base(0x538BB0)len(17))count(4)yxtvec(off(0)len(15),off(20)len(5),off(30)len(10), off(50)len(10))
tracetype(EXIT)time:(1216152765)time:(250487)str(iwritex)ptr(0x538DF0)
tracetype(ENTER)time:(1216152765)time:(250505)str(iowait)ptr(0x538DF0)
tracetype(EXIT)time:(1216152765)time:(250521)str(iowait)return(40)
```

The above trace indicates that the iwritex call sourced data from four buffers of length 10, 10, 10, and 17, and sent the data to four locations in the file, specified by (offset, length) pairs (0,15),(20,5),(30,10),(50,10). The Exit event from the iowait call reported 40 bytes transferred. This implies that 7 bytes in the last buffer went unused, which is correct since the file extent specification list was exhausted before sourcing all data referred to in the memory specification.

In the following discussion for the sake of brevity and clarity of presentation, we do not include the *tracetype* and *time* stamp, whenever an excerpt from a trace is to be shown for illustrating an observation.

IV. Applications Traced

Traces for 3 very large scientific applications were collected while they were executing representative production tasks. All three use the Message Passing Interface (MPI) to efficiently support inter-client communications.

Alegra. ALEGRA [16] is a coupled physics framework whose roots go back to 1990, when the authors joined Sandia National Laboratories and began development of a shock physics code based on arbitrary Lagrangian-Eulerian finite element algorithms. The application can be executed on multiple clients among which the input problem is distributed. The refinement of the input distribution depends on the number of clients. In this paper we analyze the two traces from Alegra provided with the same input problem , one executed on 2744 clients and the other on 5832 clients.

S3D IO Kernel. S3D is a compressible Navier-Stokes solver coupled with an integrator for detailed chemistry (CHEMKIN-compatible), and is based on high-order finite differencing, high-order explicit time integration, and conventional structured meshing [17]. The IO portion of S3D was extracted and it was this kernel that was traced.

CTH. CTH [18] is a family of codes developed at Sandia National Laboratories for modeling complex multi-dimensional, multi-material problems that are characterized by large deformations and/or strong shocks. A two-step, second-order accurate Eulerian solution algorithm is used to solve the mass, momentum, and energy conservation equations. CTH includes models for material strength, fracture, porous materials, and high explosive detonation and initiation.

Both the ALEGRA and CTH traces were taken with production binaries and parametrized with production input decks. However, in order to keep the trace files to a reasonable size, the trace files themselves have been truncated to include only the

[4] IEEE Std 1003.1 available at http://standards.ieee.org/

[5] man pages for the calls ireadv, ireadx, iwritev, iwritex, iodone, and iowait are available on Cray XT3 machines

first few compute-dump cycles. The remaining cycles, for both, are similar to the first few, excepting file address use, of course.

V. VARIATION IN DATA TRANSFER SIZE ACROSS CHECKPOINTS

For each application the amount of data transfer per second was aggregated and the data transfer size was plotted against time. The plot for the cth application is shown in Figure 1.

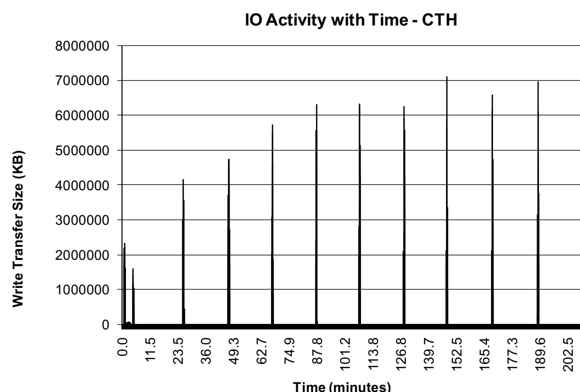

Figure 1. I/O Activity with Time for CTH

In the figure each set of spikes represents the data transfer for a checkpoint. It shows that a checkpoint was taken after about every 20 minutes. However, the total amount of data transfer for each checkpoint varies, increasing gradually in the beginning and reaching a saturation point at about the 5th checkpoint. This would seem to be a side-effect of the AMR nature of the cth application. As the application progresses, for every iteration the mesh is refined further to improve the computation efficiency. As a byproduct, the amount of data dumped for each checkpoint seems to fluctuate until the adaptive algorithm reaches some equilibrium. The cth application was traced for about 200 minutes. Since cth never settled to a completely reproducible pattern, the amount of trace data included is far larger. Once again, AMR seems the prime driver for this behavior.

Alegra performs checkpoint dumps with a period of about 4 minutes. The traces contain 4 checkpoint dumps. The times captured by the traces were sufficient to capture the behavior. For instance, the 12 minutes of Alegra capture 4 time steps. Capturing the remaining 10,000 minutes would show the same behavior. For alegra 2744 shown in Figure 2 the group of data transfers starting at 3 minutes, 5.3 minutes, 7.8 minutes and 10.3 minutes correspond to the 4 checkpoints. For alegra 5832 (shown in Figure 3) the checkpoints start at 4.4 minutes, 7.9 minutes, 11.6 minutes and 15.3 minutes. From a visual observation it is clear that the aggregate data transferred for each checkpoint is the same for every consecutive checkpoint for both versions of alegra. This strong lack of variability provides evidence for its non-adaptive nature.

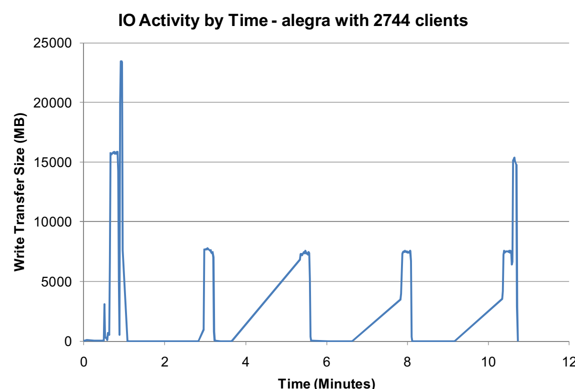

Figure 2: I/O Activity with Time: alegra with 2744 clients

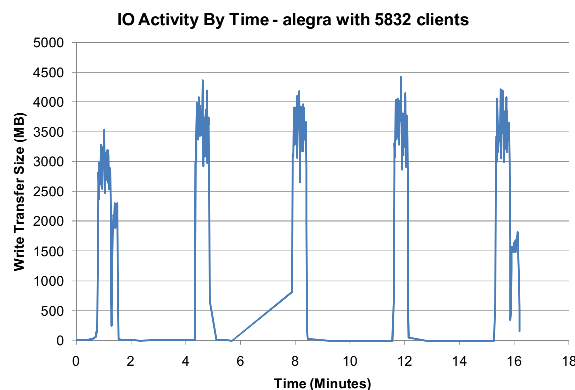

Figure 3: I/O Activity with Time: alegra with 5832 clients

VI. VARIATION IN THE NUMBER OF I/O CALLS

The traces for different applications were analyzed and the number of I/O calls made by each client counted. The clients were distributed into classes depending on the number of I/O calls they made. For alegra (refer to Figure 4 and Figure 5) and s3d it was seen that there were only a very few, static, roles for clients. Data and the related work is statically partitioned among the allocated nodes at the beginning of the application and this distribution is maintained throughout the lifetime of the run. Similarly, then, the amount of data written out by each client remains the same throughout the lifetime of the run. The following figures illustrate this phenomenon. In all three applications one of the clients shows a relatively high number of I/O calls. This is the head node that reads the input decks and communicates with all other nodes to parameterize and initialize the run.

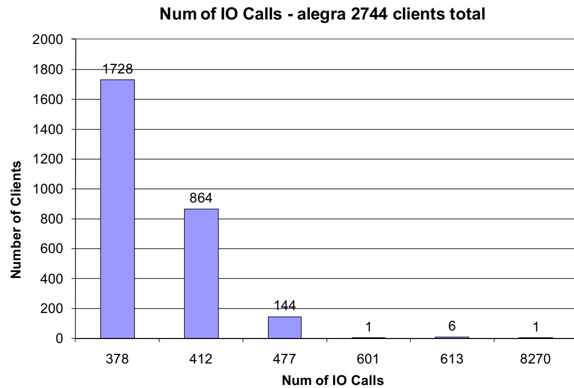

Figure 4: Client distribution by #I/O calls: alegra 2744

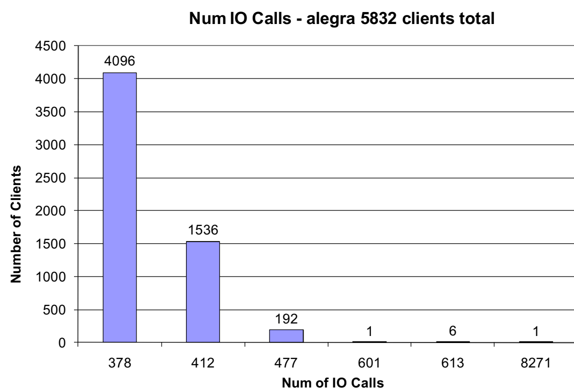

Figure 5: Client distribution by #I/O calls: alegra 5832

On the other hand for the AMR-supported application, cth, there is a wide range of the number of calls made by all clients, as shown in Figure 6. This again appears only to be explained if we consider that this AMR application distributes and redistributes either or both of its data and the amount of work per cycle required among its clients. At each iteration the data distribution is redefined to adaptively balance the computation. Figure 6 demonstrate this wide spectrum in the distribution of clients.

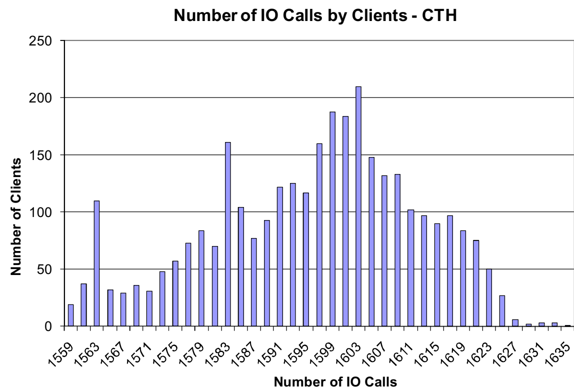

Figure 6: Client distribution by #I/O calls: cth

VII. Use of Temporary Files

The I/O trace for alegra showed calls to open two temporary files. After using the file during the course of the application, the I/O trace showed calls to rmdir to remove these entries. The "rmdir" system command is normally used to remove directory entries, when the directory is empty. This returned an error response with an error code for the rmdir command. The error code returned was ENOTDIR ("Is not a directory") since the names belonged to temporary files and not directories. The "unlink" function was then called with the same filenames as parameters to remove these file entries. This gives us an understanding of the application semantics. Firstly, that the application uses two temporary files. Secondly, the application attempted to remove these names using calls to both rmdir and unlink. One of them would be successful depending on whether the name refers to a directory or a file and the other would return an error.

```
str(open)str("file1")flags(577)mode(436)
str(open)return(3)
.
.
.
str(open)str("file2")flags(577)mode(436)
str(open)return(3)
.
.
.
str(rmdir)str("file1")
str(rmdir)errcode(-20)
str(unlink)str("file1")
str(unlink)return(0)
.
.
.
str(rmdir)str("file2")
str(rmdir)errcode(-20)
str(unlink)str("file2")
str(unlink)return(0)
```

VIII. Aggregation of I/O

The s3d application was traced in two different setups: s3d_fort is doing file per process I/O via posix I/O calls, and s3d_MPI-IO is doing single shared file I/O via MPI IO calls. Figure 7 shows that for s3d_fort all processes were performing almost the same amount of I/O uniformly. However for s3d_MPI-IO, based on sizes of trace output, we noticed that half the processes were doing very little. This is evident in the distribution of clients according to the number of I/O calls made as shown in Figure 8.

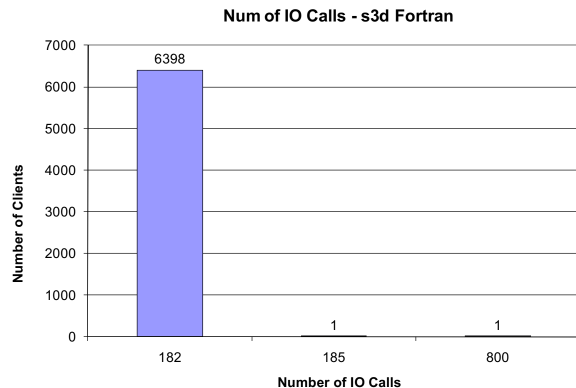

Figure 7: Client distribution by #I/O calls: s3d fortran

This behavior was not readily attributable. In this case, the source code was examined and revealed that the s3d application uses collective IO support within the ROMIO implementation of MPI-IO. ROMIO appears to chose one process from each node to use as an aggregator. This collective behavior is enabled in s3d by the use of a romio hint, 'romio_no_indep_rw', in the call to MPI_Info_set. This causes only aggregators to call open(), but not non-aggregators.

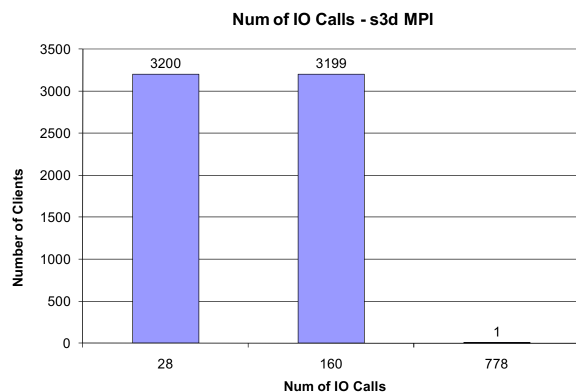

Figure 8: Client distribution by #I/O calls: s3d MPI-IO

IX. DATA TRANSFER BLOCK SIZE, SEQUENTIAL VS. RANDOM ACCESS I/O

The I/O trace shows the block of data transferred by the size of each transfer. The calls to `lseek` and `lseek64` I/O functions that appear before data transfer function calls show whether the application is performing I/O sequentially from the file or accessing random sections of the file.

```
str(ireadv)fd(3)ziovec(base(0x200EFDC0)len
(1048576))
```

The above read function trace from alegra shows that the application is trying to read 1048576 bytes = 1MB of data. Thus the data block size is at least 1 MB.

In the trace for cth, we see a larger access block of 2 MB (2097152 bytes), as shown in the following excerpt.

```
str(ireadv)fd(4)ziovec(base(0x39D07790)len
(2097152))
```

From the offsets to consecutive calls to the `lseek` function in alegra (shown below) it can be concluded that the application does not perform sequential I/O.

```
str(lseek)zoff64(1048576)
str(lseek)zoff64(524288)
str(lseek)zoff64(524288)
str(lseek)zoff64(0)
str(lseek)zoff64(1572864)
str(lseek)zoff64(1048576)
str(lseek)zoff64(524288)
str(lseek)zoff64(524288)
str(lseek)zoff64(0)
```

Another interesting observation in seek offset was in the cth application where the offsets revealed that the application continuously wrote data to the file and read back all the data in a consecutive read. This behavior is unexplained.

X. UPDATES TO SPECIFIC DATA STRUCTURES

The I/O trace was helpful in understanding the workings of the application in updating specific data structures. For example, in analyzing the traces for the cth application, one would except that the application would write the checkpoint file in large chunks to the file system. However, we observed a significant number of writes of size length 4. We inferred that the "length 4 write"s are updates to header information. This is because the client writes a chunk of data at a specific position in the file and then moves the pointer back to a particular location within the written data and again writes the 4 bytes (thus updating some 4 bytes of information within the already written data). In fact, the total number bytes written initially by each client is the same. This makes it clear that this could be some kind of header information. There are several "length 4 write"s in a single checkpoint dump from a client. For all clients, the corresponding offsets in the file for all except the last "length 4 write" seem to be the same. The location of the last length 4 write varies depending on the client, hinting that this could be client specific information in the header.

```
str(lseek64)fd(4)zoff64(0)cmd(0)
str(lseek64)zoff64(0)
str(ireadv)fd(4)ziovec(base(0x39D07790)len
(2097152))
str(ireadv)ptr(0x37D86930)
str(iowait)ptr(0x37D86930)
[1] str(iowait)return(638672)
[2] str(lseek64)fd(4)zoff64(4294667292)
cmd(1)
[3] str(lseek64)zoff64(338668)
[4] str(iwritev)fd(4)ziovec(base(…)len(4))
str(iwritev)ptr(0x37D86930)
str(iowait)ptr(0x37D86930)
str(iowait)return(4)
str(lseek64)fd(4)zoff64(0)cmd(0)
str(lseek64)zoff64(0)
str(ireadv)fd(4)ziovec(base(0x39D07790)len
(2097152))
str(ireadv)ptr(0x37D86930)
```

```
str(iowait)ptr(0x37D86930)
[5] str(iowait)return(638672)
```

In the above excerpt the first call to read (line [1]) returns 638672 bytes. The file pointer is currently at 638672 bytes. The following call to lseek64 ([2]) places the file pointer at a negative offset

4294667292 = – 300004 (in a signed 32 bit word)

relative to the current pointer, placing the final file pointer at (638672 – 300004 =) 338668 bytes as shown in line [3] the excerpt. The application then writes 4 bytes into this location (line [4]), which does not increase the size of the file but overwrite already written data. This is proved by the fact that the next read of the file returns the same 638672 bytes of data (line [5]).

XI. Distribution by data transfer size

We examined the distribution of data transfer size across the application clients. Once again, we note a remarkable distinction between applications with and without support for AMR. Figure 9 and Figure 10 show the distribution of data transfer size for the first two checkpoints of the cth application with AMR executing on 3300 clients. Figure 11 and Figure 12 show the distribution for the alegra application without AMR support. We used the alegra run with 2744 clients for comparison.

The horizontal axis shows the amount of data transferred. Clients are binned into groups based on their data transfer size. The y-axis shows a count of the clients. The distributions show that for an AMR-supported application the amount of data transferred varies widely across the clients. For a non-AMR application, the clients reliably perform similarly sized data transfers.

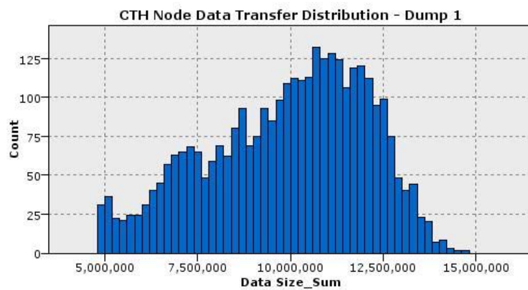

Figure 9: Data transfer size histogram for chkpt#1: cth

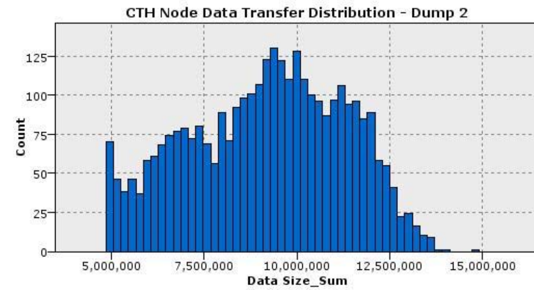

Figure 10: Data transfer size histogram for chkpt#2: cth

XII. Conclusions and Future Work

In this paper, for the first time we have obtained and analyzed I/O traces for very large scale applications executing on a large supercomputer, representative of their production runs. The tracing was performed using an I/O tracing library to be linked into the application. The analysis of the traces showed that application-level I/O traces by themselves can give us many good insights into the nature of the application without requiring access to the source code. Particularly analysis of the traces provide multiple indications of applications behavior, such as the presence of AMR (adaptive mesh refinement). Apart from distinguishing the nature of applications, the traces also revealed aggregation of I/O through MPI I/O, use of temporary files and updates to data sections in checkpoint files. The tracing tool has been developed to work specifically for the Catamount LWK. However, we cannot derive adequate benefit from the tracing mechanism by restricting it to a specific kernel. To address this very issue, we are in the process of porting it to Linux.

Acknowledgements

This work was supported in part by DOE SCIDAC-2: Scientific Data Management Center for Enabling Technologies (CET) grant DE-FC02-07ER25808, DOE FASTOS award number DE-FG02-08ER25848, NSF grants HECURA CCF-0621443, SDCI OCI-0724599, CCF-0833131, CCF-0621443, CNS-0830927 and NSF ST-HEC CCF-0444405.

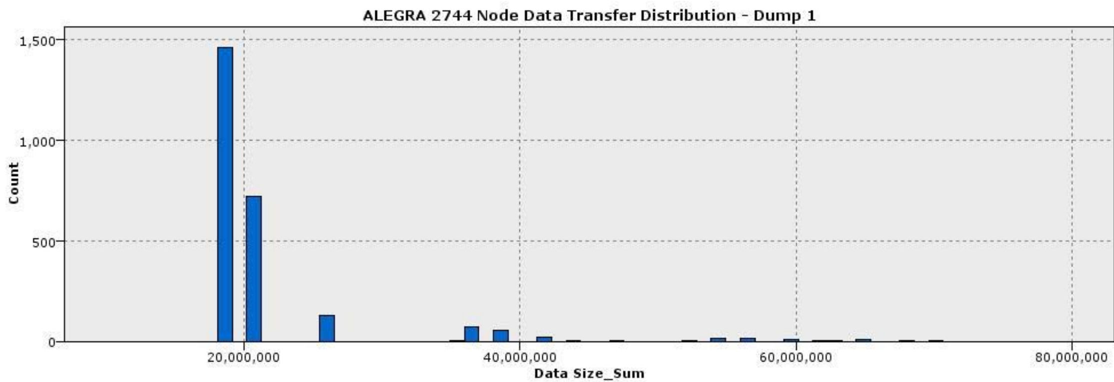

Figure 11: Data transfer size histogram for chkpt#1: alegra 2744

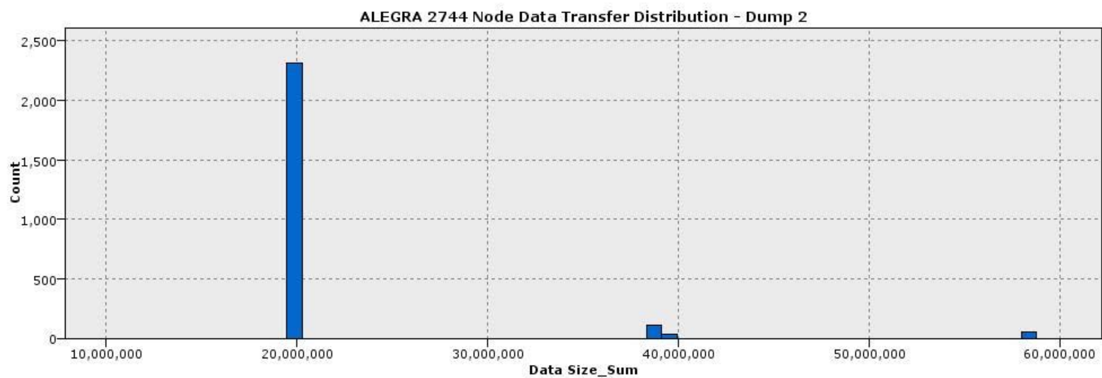

Figure 12: Data transfer size histogram for chkpt#2: alegra 2744

REFERENCES

[1] Konwinski, J. Bent, J. Nunez, and M. Quist. “Towards an I/O tracing framework taxonomy,” In Proceedings of the 2nd international Workshop on Petascale Data Storage, Nov 2007, New York, NY, 56-62.

[2] Los Alamos National Laboratory open-source LANL-Trace.<http://institute.lanl.gov/data/tdata>

[3] A. Aranya, C. P. Wright, E. Zadok. “Tracefs: A File System to Trace Them All,” In Proceedings of the 3rd USENIX Conference on File and Storage Technologies. 2004. pp. 129-145.

[4] M. Mesnier, M. Wachs, R. Sambasivan, J. Lopez, J. Hendricks, G. Ganger, D. O'Hallaron. 2007. //TRACE: Parallel Trace Replay with Approximate Causal Events. In Proceedings 5th USENIX Conference on File and Storage Technologies. pp. 153-167.

[5] Huang, T., Xu, T., and Lu, X. 2001. A high resolution disk I/O trace system. *SIGOPS Oper. Syst. Rev.* 35, 4 (Oct. 2001), 82-87.

[6] Carey, M.J., DeWitt, D.J. & Naughton, J.F. “The OO7 Benchmark”. In SIGMOD Conference on the Management of Data, 1993.

[7] Cattell, R.G.G. & Skeen, J. “Object Operations Benchmark”. ACM Transactions on Database Systems 17,1 (1992) pp 1-31.

[8] Ramakrishnan, K. K., Biswas, P., and Karedla, R. 1992. Analysis of file I/O traces in commercial computing environments. *SIGMETRICS Perform. Eval. Rev.* 20, 1 (Jun. 1992), 78-90.

[9] Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. 1985. A trace-driven analysis of the UNIX 4.2 BSD file system. *SIGOPS Oper. Syst. Rev.* 19, 5 (Dec. 1985), 15-24.

[10] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. 1988. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.* 6, 1 (Feb. 1988), 51-81.

[11] Joukov, N., Wong, T., and Zadok, E. 2005. Accurate and efficient replaying of file system traces. In *Proceedings of the 4th Conference on USENIX Conference on File*

and Storage Technologies - Volume 4 (San Francisco, CA, December 13 - 16, 2005).

[12] Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R. 2006. Stardust: tracking activity in a distributed storage system. *SIGMETRICS Perform. Eval. Rev.* 34, 1 (Jun. 2006), 3-14.

[13] Paul Barham , Austin Donnelly , Rebecca Isaacs , Richard Mortier, Using magpie for request extraction and workload modelling, Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, p.18-18, December 06-08, 2004, San Francisco, CA

[14] Ruth Klundt, Marlow Weston, Lee Ward. “I/O Tracing on Catamount”, Sandia National Laboratories. Albuquerque, New Mexico and Livermore, California.

[15] Kelly, S.M., Brightwell, R.B. “Software Architecture of the Lightweight Kernel, Catamount,” In Proceedings of the 2005 Cray Users' Group Annual Technical Conference, Albuquerque, New Mexico, May 2005.

[16] A.C. Robinson, W.J. Rider et al., “ALEGRA: An Arbitrary Lagrangian-Eulerian Multimaterial, Multiphysics Code,” AIAA-2008-1235, Proceedings of the 46th AIAA Aerospace Sciences Meeting, Reno, NV, January 2008.

[17] Don Monroe. “ENERGY Science with DIGITAL Combustors,” SciDAC Review, http://www.scidacreview.org/0602/html/combustion.html

[18] E. S. Hertel, Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. “CTH: A Software Family for Multi-Dimensional Shock Physics Analysis.” Sandia National Laboratories, Albuquerque, New Mexico, USA