DataSeries: An Efficient, Flexible Data Format for Structured Serial Data

Eric Anderson, Martin Arlitt, Charles B. Morrey III, Alistair Veitch
HP Labs, 1501 Page Mill Road, Palo Alto, CA
{eric.anderson4, martin.arlitt, brad.morrey, alistair.veitch}@hp.com

Categories and Subject Descriptors
H.3.2 [**Information Storage**]

General Terms
Design, Performance

Keywords
data format, trace-analysis, compression, performance

ABSTRACT
Structured serial data is used in many scientific fields; such data sets consist of a series of records, and are typically written once, read many times, chronologically ordered, and read sequentially. In this paper we introduce DataSeries, an on-disk format, run-time library and set of tools for storing and analyzing structured serial data. We identify six key properties of a system to store and analyze this type of data, and describe how DataSeries was designed to provide these properties. We quantify the benefits of DataSeries through several experiments. In particular, we demonstrate that DataSeries exceeds the performance of common trace formats by at least a factor of two.

1. INTRODUCTION
Traces, recordings and measurements taken from computer systems, networks and scientific infrastructure are vitally important for a large variety of tasks. In every area of computer system design, traces from existing systems have been used to validate hypotheses, test assumptions and estimate performance. This is true of I/O subsystems [3, 14, 26], processor systems [19], network systems [15] and memory systems [24], among others. Traces and logs are also extremely useful for fault-finding, auditing and debugging purposes [20]. Traces composed of failure data have been used to determine system reliability [7, 23, 21]. Trend analyses of performance information is a core operation of various management tools [11]. Scientific and medical instrumentation can generate very large amounts of data [4], which also needs to be stored, filtered and analyzed.

The data stored in each of these diverse cases is *structured serial data*, which we define as a series of records, each record having a specified structure (i.e., containing the same set of variables or fields). Structured serial data has four defining characteristics: its structure is record-oriented; it is typically written only once, and is read many times; it is usually ordered in some manner, e.g., chronologically; and it is typically read sequentially. We have designed and built DataSeries, an on-disk data format, run-time library, and set of tools that is optimized for storing and analyzing this type of data. We show that DataSeries outperforms common trace formats and databases by at least a factor of two, and in some cases up to an order of magnitude. DataSeries also requires far less disk space (factors vary from 4x to 8x in test data sets).

We desire six key properties of a data format and analysis system for structured serial data:

1. **Storage efficiency**: the data should be stored in as few bytes as possible. There are several driving factors behind this requirement. First, the amount of data stored can be large (we have I/O traces comprising billions of records), and despite rapidly decreasing storage prices, the cost to store data can still be considerable, particularly when the data must be kept for long periods of time. Second, we have learned that one of the primary factors behind analysis efficiency is the speed at which data can be retrieved. Regardless of the storage technology used, more highly compressed data can substantially speed access times.
2. **Access efficiency**: accessing, interpreting and encoding trace data, whether reading or writing, should make efficient use of CPU and memory resources. From experience and experimental analysis, we have learned that the second major factor determining analysis efficiency is the CPU overhead of interpreting data once it has been read off disk.
3. **Flexibility**: adding additional fields should not affect users of the trace data. Removing or modifying data fields should only affect programs that use those fields, and the system should catch incorrect usage. Further, the format should not constrain the type of data being stored, and should allow multiple record types in a single file. Experience tells us that formats that are not flexible lead to severe maintenance issues for the format interpretation code and for analysis systems.
4. **Self-description**: the data set should contain the metadata that describes the data. This is another experience-driven requirement. We have had problems trying to interpret and use trace data from other organizations because of missing metadata, and with maintaining organizational knowledge of our own metadata over time.

5. **Usability**: the data format should have an associated programming interface that is both expressive and easy to use. The user model of the data and its analysis should be easy to describe and the interface to it should easily allow for common operations (e.g., scanning an entire data set, and processing specific fields only).

6. **Integrity**: many structured serial data files are intended as archival traces. To protect against media errors or incorrect software systems, files need to be self-contained and contain internal checksums that enable integrity checking.

There are, inevitably, tradeoffs to be made in satisfying these properties. For instance, XML is an extremely flexible data format, but is not very efficient. In the course of our work (which tends towards the analysis of very large data sets), and the design of DataSeries, we have chosen to prioritize efficiency over many of the other properties.

Although numerous tracing and measurement systems have been developed over the last 20–30 years, we are not aware of any that meet all of these properties. We analyze some of these in our description of related work (Section 2).

We provide four primary contributions in this paper. First, we introduce DataSeries, a data format and associated library, which was specifically designed to meet the six key properties discussed above, and relate some of the experiences that led us to make various design decisions. Second, we discuss how DataSeries can support very large data sets (e.g., hundreds of billions of records) on modest systems. Third, we describe how we have used DataSeries in practice to store a wide variety of data types. Fourth, we demonstrate the performance and storage efficiency of DataSeries in a set of controlled experiments, using empirical data sets. Throughout, we have tried to emphasize the lessons learned and how they might be applied to other fields.

DataSeries software is publicly available under a BSD license from `http://tesla.hpl.hp.com/opensource/`. Given the benefits of DataSeries that we demonstrate, we argue that DataSeries should be considered for use by any application that needs to store large amounts of structured serial data. Indeed the Storage Networking Industry Association (SNIA) I/O traces tools and analysis (IOTTA) technical working group [12] has proposed DataSeries as the basis for a standard I/O trace data format.

The remainder of this paper is organized as follows. Section 2 describes the strengths and weaknesses of existing storage technologies relative to DataSeries. Section 3 describes the design of DataSeries, including on-disk and in-memory formats, and introduces the programming model. Section 4 presents empirical and benchmark results to illustrate and quantify the benefits of DataSeries. Section 5 concludes the paper with a summary of our work.

2. RELATED WORK

We classify the related work into three categories: those that use a customized binary format, those that use a text-based format, and relational database systems. For more complete and quantitative comparisons, please refer to [5].

Custom binary formats are usually serialized or directly written versions of an in-memory data structure. As such, they usually achieve storage and access efficiency, but do not achieve flexibility, self-description, usability, and integrity. However, as we show in Section 4.2, unless the authors are careful they can also fail to achieve access efficiency.

Text formats such as Comma-Separated Value (CSV) can achieve flexibility and self-description. XML achieves flexibility, self-description and usability. However, they fail to achieve storage efficiency and integrity, and can fail access efficiency by multiple orders of magnitude. A highly-tuned CSV implementation only got to within 2–7x the access-efficiency of DataSeries, and 4–7x the storage efficiency [5].

Relational databases achieve flexibility, self-description, usability, and integrity. RDBMS's were designed to handle updates, so do very limited compression drastically hurting their storage efficiency. Our results show >10x improvement on storage efficiency for DataSeries over MySQL. Similarly, the generality of SQL can be detrimental. Even for fairly simple queries running entirely on in-memory data, DataSeries runs 2–7x faster than MySQL. Retrieving the data for a more complicated calculation on the client further slows the relative performance.

3. DESIGN

DataSeries' data model is conceptually very similar to that used by relational databases. Logically, a DataSeries file is composed of an ordered sequence of *records*, where each record is composed from a set of *fields*. Each field has a *field-type* (e.g., integer, string, double, boolean) and a name. A DataSeries record is analogous to a row in a conventional relational database. For efficiency, we group a set of rows that have the same fields and field types into an extent. We call the type of that extent the *extent-type*. Usually an analysis runs over a collection of extents, so one or more extents of the same type is similar to a database table.

3.1 Data format

A single DataSeries file comprises one or more extents (potentially with different extent-types), a header and extent-type extent at the beginning of the file, and an index extent and trailer at the end of the file. The header on a DataSeries file contains the DataSeries file version, and check values that enable the reader to determine the endianness encodings of the data types. DataSeries files are always written out using the native formats of the writing system. This typically minimizes byte-swapping overheads, as the architecture reading the files is almost always the same as the architecture writing the files. The library transparently converts in the rare cases this is not true, and DataSeries files can be explicitly "repacked" if desired.

The extent-type extent contains records with a single string-valued field, each of which contains an XML specification that defines the extent-types of all the other extents in the file. We chose to use XML as we saw no point in creating a new grammar to represent this information, as it is flexible (for instance, it easily allows the addition of options describing fields) and allows embedded comments.

The trailer consists of the offset and size (after compression)

of the index extent. The offset is used to read the index extent, which has two fields, an extent-type and an offset, to allow direct access to extents of a single type. Storing the index and trailer at the end of the file enable efficient writing, since writing applications will often not know the final extent sizes *a priori*, so space for the index cannot be allocated until after the extents have been written.

Each extent consists of a header, followed by the fixed-sized fields and separately the variable-sized data. A reference to the variable-sized data from the fixed-size fields allows for variable-sized fields. This separation allows for direct access to fixed-sized fields since they will be at a known offset, and one indirection to get at the variable-sized data. Both fixed- and variable-sized data may be compressed, using any one of a number of standard compression algorithms [2, 10, 17, 18]. The extent header contains metadata about the data in the extent, such as the compressed sizes of the fixed and variable data, the number of records in the extent, the uncompressed size of the variable-length data, the compression mode, the extent-type of the extent, and checksums of the extent before and after compression to guard against hardware and software errors. Programs usually disable checksum validation during extent reading to improve performance at the cost of reduced integrity.

The extent format is designed for efficient access. Values are packed so that once an extent is read into memory, an analysis can iterate over the rows simply by increasing a single counter, as if for an array containing structures. Individual values are accessed by an offset from that counter and a C++ cast operation.

As we initially built DataSeries prototypes, we realized the need to add options that control various aspects of the field and extent descriptions in order to meet the storage and access efficiency and flexibility properties. While space precludes a full description (see [5]), we describe some of the more significant below:

- **nullable fields**: there are many instances when a field value may or may not be present, and we found it useful to be able to indicate that a given value is null. This option is implemented by generating a hidden boolean column that determines if the value is null.
- **double scale**: often, the full 53-bit precision of a double is not needed. Hence, it is possible to get much better compression by zeroing unimportant lower-order bits by scaling and rounding. This option lets the user specify a specific scale factor for the values stored.
- **relative packing**: specifies that a given field be packed relative to another, using delta encoding. This option is useful for timestamps and index values that may be large, but are small relative to each other. This enables fewer bits to be used to store a given value.
- **unique packing**: ensures that every string (arbitrary-length binary data) field is stored only once. If a data set contains many repeated values, this option can significantly increase the compression ratio, as compression algorithms only partially remove duplicate data.
- **versioning**: since our format is archival, we expect changes in the fields used in a data set over time. To handle these changes we introduced a version number associated with a type. A program compatible with version $n.m$ of the format will operate properly on any file with version $n.m', m' >= m$. Programs can also detect the version number and adapt appropriately.

One addition we tried, and later rejected, was that of extending a double field's precision by allowing the specification of a base value. This was introduced when we realized that storing microsecond precision time values, using the UNIX epoch (00:00:00 UTC on January 1, 1970), was not possible with a single double. We added an option to specify a base value for the double fields, and then stored the actual value relative to that base value. Unfortunately, experience showed that this option was less useful than hoped for. It proved difficult and confusing for those developing analyses to deal with, particularly in the case where multiple files, each with different bases, were being used. Arithmetic on values of this type was complicated and inefficient. Ultimately, we deprecated this option in favor of other time representations, dependent on the domain.

We have now chosen to explicitly represent the units and the epoch for time values. The units are chosen based on the original precision, e.g., nanoseconds for NFS attribute values, or microseconds for pcap [16] trace files. We then use a 64-bit integer to store the time value, and have a special field that can translate in and out of the raw units from input formats such as a double, or a (second, nanosecond) pair. Analyses are written to operate over the "raw" time format, and then they use the field to convert to more normal formats for printing. If the analysis needs to calculate windows, it can convert a normal format to a "raw" format. For example to calculate mean bytes per 30 second window, an analysis would first convert 30 seconds into the raw format, then accumulate values for that interval, output the mean and reset the statistic. The use of units and epoch also allows us to print out time values in a nicer human readable form regardless of the input type. To deal with older files that are missing units and epoch, we added the ability to specify units and epochs based on the extent type, extent version and field names.

If there is no natural unit for a set of traces, for example because they have cycle counter times, then we recommend units of 2^{-32} seconds. This choice allows for the maximum precision possible when providing the same range as a UNIX time value since the UNIX time value uses 32 bits to represent the seconds. We expect at some point to need to move to a 128 bit fixed or floating point representation to deal with the continually increasing range of times for trace data and the increasing precision of the clocks measuring the times. Theoretically, the maximum required bits would be around 200 since that would be sufficient to represent all times for the age of the universe using the Planck time granularity.

3.2 Programming model

Four general functionalities are supported by DataSeries: reading a DataSeries file, analyzing the data in a DataSeries file, writing a DataSeries file, and writing an alternative out-

put format (e.g., CSV). These functionalities meet all of the typical needs for users of structured serial data, and thus facilitate the usability (property 5) of DataSeries.

There are two key concepts in the C++ API provided by DataSeries. The first is that of an *ExtentSeries*, which is an iterator over the rows in at least one extent. Iterators in DataSeries are similar to iterators in relational databases [8], but they typically operate in bulk over an entire extent in a single pass to improve access efficiency. Iterators in DataSeries are also not required to operate over extents of identical type, instead an iterator specifies its compatibility for types. Normally compatibility is exact type compatibility, but it can also be loose, which means that the types are compatible if all the fields can be found. The second key concept is that of a *module*, which accepts a series of extents, processes them and passes them to downstream modules. Similar to River [1], modules can have multiple inputs if they are joining together two different series. The ability to add functionality in a modular fashion enhances the flexibility (property 3) of DataSeries.

Analysis programs generally have a main program that builds a sequence of modules to perform multiple analyses in a single pass over the data. Each module processes an extent at a time, and iterates over each of the rows in the extent. For easier usability, a row analysis module can be used that will call a processRow() function on each row. The `dstypes2cxx` program will automatically generate the boilerplate needed for a module to be used as a row analysis module. For efficiency reasons, the transfer of extents between modules is a transfer of ownership, i.e., the module that returns an extent must not continue to access it. We are considering relaxing this constraint in the future to allow multiple modules to read-share a single extent at the same time so that we can increase the parallelism for analysis without increasing memory requirements. DataSeries also provides modules which exploit parallelism in operations, such as decompressing or compressing and writing extents to disk in parallel, to increase the access efficiency on multi-core systems. This improvement is very important for compression, which is usually slow, but can also be important for simple analyses that run faster than a single core can decompress.

The DataSeries source distribution contains numerous other built-in general modules, for example, some that convert extents to text, and others that perform simple versions of the SQL select/group-by statement. We also have converters from various input formats (pcap, log file, SRT [25]) into a DataSeries representation. Last, there are type-specific analyses for converted NFS, LSF batch jobs [22] and logical/physical disk volume traces. More detailed information on these modules and programming examples is available in [5] and the source distribution.

4. PERFORMANCE RESULTS

In this section we provide examples to illustrate quantitatively the effectiveness of DataSeries. Section 4.1 describes how we have used DataSeries to store and analyze a large data set on a modest system. Section 4.2 demonstrates the flexibility provided by DataSeries for selecting between storage and access efficiency needs. Section 4.3 provides a case study which illustrates both the reusability of DataSeries and its access efficiency compared to existing work. Additional examples and comparisons can be found in [5].

4.1 Scalability

The largest data set we have is a trace of NFS traffic to and from busy enterprise file servers. The primary extent type in this data is the common records type which stores information about each of the 200 billion request and reply messages. We have secondary tables that store information about each packet captured, as well as information on NFS operations like read and write requests and mount requests. The total data set is about 6.3TiB compressed with gzip.

As a demonstration of the real-life performance of DataSeries, consider the following example. Utilizing our NFS traces, we calculated a cube [9] over the common data in NFS operations. Using a machine with two dual-core 2.4 GHz CPUs, we found the analysis ran at 2.11 Mrows/s for user time, 1.77 Mrows/s for total CPU time, and 3.32 Mrows/s for elapsed time. Analysis over only the larger data sets (>100 Mrows) got somewhat better (3.55 Mrows/s) wall clock time because the startup and shutdown overhead are more amortized and prefetching had more time to take effect.

4.2 Examining storage and access efficiency

We have performed extensive experiments on the effect of compression on analysis performance [5]. We summarize the results on storage efficiency below:

- For archival storage, or preparation for network distribution, bzip2 [2] compression with large extents (16–64MB) makes the most sense. Keeping the size at most 64MB allows for some parallelism during compression, and later decompression, but extracts almost all of the potential compression available. If gzip [10] compression is enabled then it will automatically be used when it achieves better compression than bzip2.
- For analyses with sufficient disk bandwidth, lzo [18] with small extents (96–128KB) gets the maximum performance. The optimal extent size is somewhat below the L2 cache size as there needs to be sufficient cache space to hold the entire uncompressed extent, some of the compressed data while it is uncompressed, and any additional program state.
- With more constrained disk bandwidth, gzip with small extents is better than lzo as it trades a 10–30% reduction in decompression rate for a 10–40% improvement in compression; if disk bandwidth is a bottleneck then the improved compression is more valuable. The additional compression from longer extents is minimal, and bzip2, while compressing much faster, decompresses so much slower that only a highly imbalanced system would benefit from bzip2 decompression.
- For online compression of data, lzf [17] with small to moderate-sized extents is best, as it compresses at roughly 100MB/s (about 10x faster than gzip at level 1). If the data has many unique strings, the unique packing option is very fast and improves compression when combined with moderate-sized extents. Otherwise, staying within the L2 cache remains a priority.

Access efficiency is more difficult to measure. Our most extensive analysis of this occurred using the 1998 World Cup traces [13]. These traces came with a sample analysis program that calculated operations/server, minimum and maximum object IDs, and other simple statistics. Since the raw format of the data was a binary structure, we expected that the DataSeries version would have a faster wall clock time (because of parallel decompression), but a slower CPU time. We were surprised to learn that we actually used less CPU time. The reason was a combination of fread inefficiency when used one record at a time, and a slow byte-swap routine used by the sample program. DataSeries avoids the former by doing bulk processing and the latter by switching the ordering to the native order when a file is written or repacked. Further investigation indicated that DataSeries was also executing unnecessary instructions because our field accessors supported nullable fields, but this happened to be unused in this format. We added C++ template-based fields, but this did not reduce the penalty for access to the data because the g++-3.3 compiler used with RHEL4 is fairly poor at optimizing templated code. However, with the g++-4.3 compiler, the penalty is removed and the analysis code uses the same number of instructions on both sides.

The following experience highlights the tradeoff between general and specific code. Using HP-UX block I/O traces, we examined the performance difference for fields that handle nullable and non-nullable data. We compared the same analysis using special-case fields, templated fields, basic fields, general fields, and a general-purpose program. We implemented special-cases of the fields to work around the compiler difficulties, and found that templated fields were 2% slower than the special case fields, the basic fields were another 1% slower, and the general fields were yet another 1% slower. The minor performance differences were because each field was used exactly once, so most of the cost was in the memory access to the field, which was usually not in cache as a different thread performed decompression. Conversely, the fully general program was substantially slower (20%) as it determines the expression at runtime, and is capable of grouping on any field type, while the specific-case programs can only group by 32-bit integer fields.

We expect that if we added prefetching code to the row analysis module we would see a larger difference in the performance. We measured a roughly 10x difference in access time through the basic and the special-case fields when using a program that repeatedly accessed the same field and just accumulated the sum. We believe extra branch instructions are interfering with the CPU's ability to fully pipeline the code with the basic fields while the special-case fields reduce the access down to a single instruction which can be combined with the add instruction. The general fields on in-cache data are about 2.5x slower than the basic fields, showing the overhead of the additional virtual function call.

4.3 A case study

In an effort to experiment with using DataSeries to represent and analyze traces generated by other people, we converted the NFS traces used by Ellard *et al.* [6] into DataSeries. These "Ellard traces" were originally stored as compressed text files, one record per line. The first part of each line is a series of fixed fields, followed by a set of key-value pairs, and finally some debugging information. A scanning program read trace files and outputs summary information.

Our evaluation consisted of two parts. First, we wrote a reversible conversion program to verify that we were properly preserving all of the information. We found that the DataSeries files were on average 0.77x the size of the original files when both were compressed using gzip. The compression improvements came as a result of the unique string packing, delta encoding, and the elimination of the key fields through the use of fixed field names and representing null fields in a single bit before generic compression.

Second, we wrote an analysis program that implemented the first three examples in the README that came with the Ellard tools. These examples were all variants of a "select count(*) group by field" query. Table 1 presents our experimental results (using machines with two 2.4GHz dual-core CPUs, details in [5]). The first two rows of Table 1 show that the compression algorithm has little effect on the performance of the Ellard tool. With gzip, our analysis program ran about 76x faster on those data files (row 5). With lzo compression, which decompresses more quickly than gzip, our analysis program ran about 107x faster (row 6), in exchange for slightly larger (1.14x) data files. This also illustrates how DataSeries can be optimized for a given purpose (e.g., greater compression for archival storage versus faster decompression for more-efficient analysis).

The least speedup with our tool occurred with the bzip2 compression algorithm and large extent size (row 9). Although this configuration was still 20x faster than the Ellard tool, it is noticeably slower than our other configurations. The substantial increase in system time for bzip2 (11.82s with 16MB extents versus 1.14s for lzo with 64KB extents) occurs because glibc calls mmap/munmap for all extent allocations. This results in a substantial amount of page zeroing in the kernel, and hence a large increase in the system time.

4.4 Other results

We have performed a number of other experiments that demonstrate DataSeries' benefits. Due to space considerations, we will describe them only briefly. One study of data compression showed that for a relatively large disk I/O trace, DataSeries provides superior disk space utilization; stored in DataSeries, the trace consumed 1.9GB of disk space, but 14GB in CSV format and 8.5GB as a MySQL database. Using the same trace in a series of analyses showed that DataSeries delivers results 6–7x faster than both formats.

Our results and experience with DataSeries have also illustrated some of the tradeoffs between generality and performance. We have found that by adding various type-specific functionality, we can significantly improve performance. For instance, for the analysis of the World Cup traces, we added new accessors that did not check for nullable fields, and found that we decreased CPU time by 8%. Similarly, we found that we could "group" multiple analyses into a single routine (therefore needing to pass over the data only once), which can decrease the overall time taken by the same factor. This is especially effective when compared to the time taken in a more general solution (e.g., SQL queries) which cannot be combined in this manner. The primary tradeoff

Table 1: Detailed performance comparison for Ellard and DataSeries analysis programs.

row	tool	compression algorithm	extent size	mean CPU time (s)			CPU speedup	mean wall time (s)	wall time speedup
				user	system	total			
1	Ellard	gzip		537.58	7.80	545.38	1.000x	545.71	1.000x
2	Ellard	bzip2		638.48	12.68	651.16	0.836x	571.49	0.955x
3	DataSeries	gzip	64KB	21.45	1.14	22.59	24.147x	5.81	93.945x
4	DataSeries	gzip	128KB	23.30	1.19	24.49	22.268x	6.30	86.604x
5	DataSeries	gzip	512KB	22.91	3.62	26.53	20.557x	7.16	76.186x
6	DataSeries	lzo	64KB	18.71	1.14	19.85	27.472x	5.10	106.897x
7	DataSeries	lzo	128KB	21.15	1.10	22.25	24.514x	5.74	95.022x
8	DataSeries	lzo	512KB	24.07	4.07	28.14	19.382x	7.40	73.762x
9	DataSeries	bzip2	16MB	94.38	11.82	106.20	5.136x	27.66	19.732x

is the added complexity, effort and time involved in writing and using such type-specific accessors, which may not be worthwhile for one-shot analyses.

5. CONCLUSIONS

We have described DataSeries, a data format that enables the efficient and flexible storage and analysis of structured serial data. This type of data is used in numerous applications in all areas of computing and science. We have identified the properties required for a system that processes such data, and shown through a series of experiments and comparisons to other systems that DataSeries satisfies these properties. In particular, DataSeries offers significant performance and storage efficiency benefits.

We have also described the various lessons learned while designing and implementing DataSeries. A general theme from this has been that, when dealing with very large quantities of structured serial data, there are significant performance, efficiency and reliability gains to be made by giving up a relatively small amount of generality and flexibility. Overall, we believe that DataSeries should be considered for use in any application that processes structured serial data.

6. REFERENCES

[1] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, 1999. ACM Press.

[2] bzip2 compression library, http://www.bzip.org/, accessed September 2007.

[3] Z. Chen, Y. Zhang, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *ACM SIGMETRICS*, pages 145–156, June 2005.

[4] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187–200, July 2000.

[5] http://tesla.hpl.hp.com/opensource/DataSeries-tr-snapshot.pdf.

[6] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 203–216, San Francisco, CA, 2003. USENIX.

[7] A. Ganapathi and D. Patterson. Crash data collection: a windows case study. In *Dependable Systems and Networks*, pages 280–285, July 2005.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, 1993.

[9] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Data Mining and Knowledge Discover*, volume 1, pages 29–53, 1997.

[10] gzip compression library, http://www.gzip.org/, accessed September 2007.

[11] E. Hoke, J. Sun, and C. Faloutsos. Intemon: Intelligent system monitoring on large clusters. In *VLDB*, pages 1239–1242, September 2006.

[12] http://iotta.snia.org/, accesed July 2008.

[13] http://ita.ee.lbl.gov/html/contrib/WorldCup.html, accessed July 2008.

[14] M. Ji, A. Veitch, and J. Wilkes. Seneca: remote mirroring done write. In *USENIX Technical Conference*, pages 253–268, June 2003.

[15] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[16] http://www.tcpdump.org/, accessed September 2007.

[17] lzf compression library, http://www.goof.com/pcg/marc/liblzf.html, accessed September 2007.

[18] lzo compression library, http://www.oberhumer.com/opensource/lzo/, accessed September 2007.

[19] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *ACM SIGMETRICS*, pages 216–227, June 2006.

[20] R. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. *SIGPLAN Notices*, 28(12):1–11, 1993.

[21] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST*, pages 17–28, February 2007.

[22] Platform Load Sharing Facility, http://www.platform.com/Products/Platform.LSF.Family/, accessed September 2007.

[23] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you. In *FAST*, pages 1–16, February 2007.

[24] S. Sohoni, R. Min, Z. Xu, and Y. Hu. A study of memory system performance of multimedia applications. In *ACM SIGMETRICS*, pages 206–215, June 2001.

[25] http://tesla.hpl.hp.com/public_software/; SRT and trace data sections; accessed September 2007.

[26] M. Uysal, A. Merchant, and G. Alvarez. Using MEMS-based storage in disk arrays. In *Conference on File and Storage Technologies (FAST)*, pages 89–102, April 2003.