In-Memory Runtime File Systems for Many-Task Computing

Alexandru Uta, Andreea Sandu, Ion Morozan, and Thilo Kielmann

Dept. of Computer Science,Vrije Universiteit, Amsterdam, The Netherlands

1 Introduction

Many scientific computations can be expressed as Many-Task Computing (MTC) applications. In such scenarios, application processes communicate by means of intermediate files, in particular input, temporary data generated during job execution (stored in a runtime file system), and output. In data-intensive scenarios, the temporary data is generally much larger than input and output. In a 6x6 degree Montage mosaic [3], for example, the input, output and intermediate data sizes are 3.2GB, 10.9GB and 45.5GB, respectively [6]. Thus, speeding up I/O access to temporary data is key to achieving good overall performance.

General-purpose, distributed or parallel file systems such as NFS, GPFS, or PVFS provide less than desirable performance for temporary data for two reasons. First, they are typically backed by physical disks or SSDs, limiting the achievable bandwidth and latency of the file system. Second, they provide POSIX semantics which are both too costly and unnecessarily strict for temporary data of MTC applications that are written once and read several times. Tailoring a runtime file system to this pattern can lead to significant performance improvements.

Memory-based runtime file systems promise better performance. For MTC applications, such file systems are co-designed with task schedulers, aiming at data locality [6]. Here, tasks are placed onto nodes that contain the required input files, while write operations go to the node's own memory. Analyzing the communication patterns of workflows like Montage [3], however, shows that, initially, files are created by a single task. In subsequent steps, tasks combine several files, and final results are based on global data aggregation. Aiming at data locality hence leads to two significant drawbacks: (1.) Local-only write operations can lead to significant storage imbalance across nodes, while local-only read operations cause file replication onto all nodes that need them, which in worst case might exceed the memory capacity of nodes performing global data reductions. (2.) Because tasks typically read more than a single input file, locality-aware task placement is difficult to achieve in the first place.

To overcome these drawbacks, we designed a distributed, in-memory runtime file system called MemFS that replaces data locality by uniformly spreading file stripes across all storage nodes. Due to its striping mechanism, MemFS leverages full network bisection bandwidth, maximizing I/O performance while avoiding storage imbalance problems.

2 MemFS

The MemFS distributed file system [5] consists of three key components: a *storage* layer, a *data distribution* component, and a *file system client*. Typically, all three components run on all application nodes. In general, however, it would also be possible to use a (partially) disjoint set of storage servers, for example when the application itself has large memory requirements.

The storage layer exposes a node's main memory for storing the data in a distributed fashion. We use the Memcached [2] key-value store. MemFS equally distributes the files across the available Memcached servers, based on file striping. For mapping file stripes to servers, we use a hashing function provided by Libmemcached [1], a Memcached client library. We use the file names and stripe numbers as hash keys for selecting the storage servers. We expose our storage system using a FUSE [4] layer, exposing a regular file system interface to the MTC applications. At startup, the FUSE clients are configured with a list of storage servers. Through the Libmemcached API, the FUSE file system communicates with the Memcached storage servers.

Figure 1 shows the overall system design of MemFS, using the example of a write operation, issuing Memcached *set* commands; for read operations, *get* commands would be used instead.

MemFS had originally been designed for tightly-coupled compute clusters where premium networks like Infiniband provide intersection bandwidth of several tens of Gb per second. Experimentation has shown that MemFS works very well with these networks. But also with slower interconnects, like Gb Ethernet, MemFS shows its superiority, compared to locality-based approaches.

Currently, a limiting factor for MemFS performance is the user-space implementation based on FUSE, that causes significant CPU load for processing file-system operations on the client side. An alternative MemFS implementation is providing a kernel-based file system that reduces the amount of context switches between user space and kernel space to the absolute minimum. Our

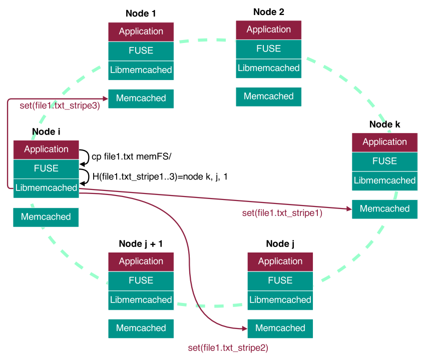

Fig. 1. MemFS System Design

kernel-based version of MemFS shows significant reduction in CPU loads on the client side.

The drawback of a kernel-space file system is that it requires superuser privileges, which can be a limiting factor for deployment on cluster machines. When using MemFS in virtualized cloud environments, however, the kernel-space implementation can be used easily and efficiently.

3 Conclusions

MemFS is a fully-symmetrical, in-memory distributed runtime file system. Its design is based on uniformly distributing file stripes across the storage nodes belonging to an application by means of a distributed hash function, purposefully sacrificing data locality for balancing both network traffic and memory consumption. This way, reading and writing files can benefit from full network bisection bandwidth, while data distribution is balanced across the storage servers.

Acknowledgments

This work is partially funded by the Dutch public-private research community COMMIT/.

References

1. B Aker. Libmemcached. http://libmemcached.org/libMemcached.html, 2014.
2. Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.
3. Joseph C Jacob, Daniel S Katz, G Bruce Berriman, John C Good, Anastasia Laity, Ewa Deelman, Carl Kesselman, Gurmeet Singh, Mei-Hui Su, Thomas Prince, et al. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*, 4(2):73–87, 2009.
4. Miklos Szeredi et al. FUSE: Filesystem in userspace. *http://fuse.sourceforge.net/*, 2014.
5. Alexandru Uta, Andreea Sandu, and Thilo Kielmann. MemFS: an In-Memory Runtime File System with Symmetrical Data Distribution. In *IEEE Cluster 2014*, Madrid, Spain, September 2014. (poster paper).
6. Zhao Zhang, Daniel S Katz, Timothy G Armstrong, Justin M Wozniak, and Ian Foster. Parallelizing the execution of sequential scripts. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013.