POSTER: MemFS: an In-Memory Runtime File System with Symmetrical Data Distribution

Alexandru Uta, Andreea Sandu, Thilo Kielmann
Dept. of Computer Science, VU University Amsterdam, The Netherlands
a.uta@vu.nl, a.sandu@vu.nl, thilo.kielmann@vu.nl

I. 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 [5]. 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 [5]. 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.

II. MemFS

The MemFS distributed file system 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.

1) Storage: This layer exposes a node's main memory for storing the data in a distributed fashion. We use the Memcached [2] key-value store.

2) Data Distribution: 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.

3) File System Client: 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.

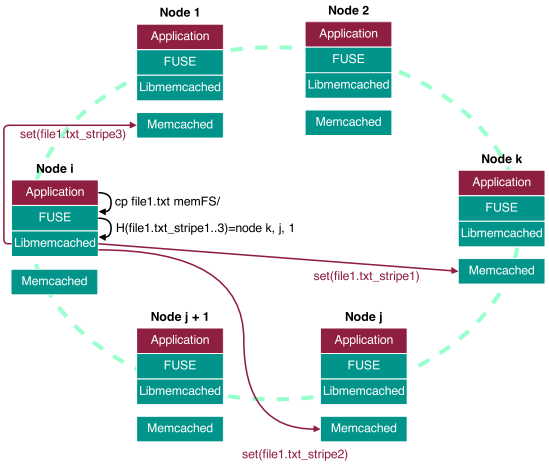
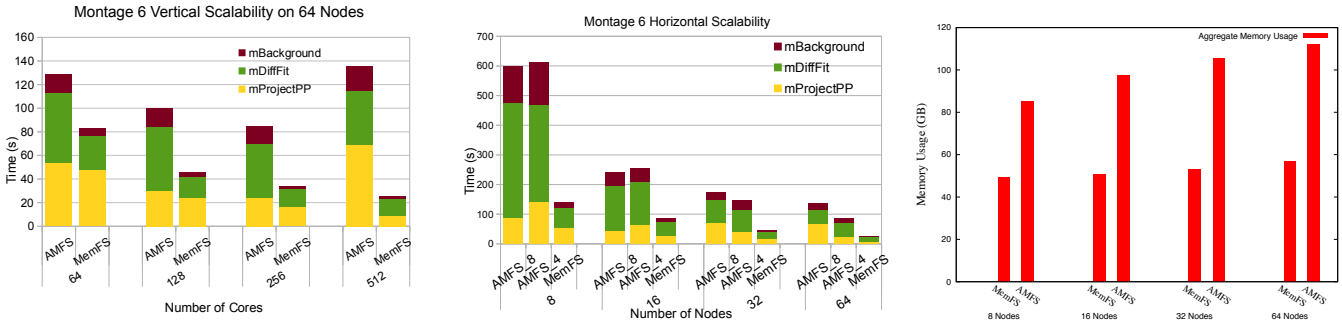


Fig. 1. MemFS System Design



(a) 64 nodes, scaling up to 512 cores

(b) 8–64 nodes, scaling out to 512 cores

(c) Memory usage while scaling out to 512 cores

Fig. 2. Montage Vertical and Horizontal Scalability and Memory Consumption

III. Montage Results

We ran the Montage [3] workflow on our local DAS4 system (`www.cs.vu.nl/das4/`). Its compute nodes are equipped with dual-quad-core Intel E5620 2.4 GHz CPUs and 24GB memory. The nodes are connected using Quad Data Rate (QDR) InfiniBand, using the IP over InfiniBand (IPoIB) interface results in approximately 1GB/s bandwidth.

We compare the performance achieved by MemFS to the locality-oriented AMFS [5] in-memory file system. AMFS improves application performance by issuing only local writes and uses the AMFS Shell scheduler for executing compute tasks on those nodes that actually store needed files to improve read performance. AMFS Shell, however, can only guarantee that one file per job achieves data locality. In case multiple files are read per scheduled job, expensive remote reads become necessary. MemFS, in contrast, uniformly distributes file stripes across storage nodes by means of a distributed hash function to achieve balanced memory consumption, while utilizing the aggregate bandwidth among all nodes. For all experiments, the compute nodes also operate as storage nodes, for both AMFS and MemFS.

AMFS Shell originally had been designed such that it schedules one task per compute node. For using our multicore cluster, we have modified AMFS Shell such that it can schedule multiple jobs at a time on a given node, while preserving data-locality when using AMFS. When using MemFS as the storage backend, the multicore-aware scheduler simply submits multiple jobs at a time, ignoring data-locality.

We assessed each system's scaling behaviour. By scaling *vertically* (*up*), we analyze the system behaviour on a fixed number of nodes, while gradually increasing the number of compute cores used for task processing. Conversely, by scaling *horizontally* (*out*), we analyze the system behaviour while gradually increasing the number of compute nodes.

Figure 2a shows the vertical scalability of the two file systems for the Montage 6 workflow. The results were determined on 64 nodes, using gradually 1, 2, 4, and 8 compute cores each. MemFS shows good scalability up to 512 cores, while AMFS scales only up to 256 compute cores.

Figure 2b depicts the horizontal scalability comparison of the two file systems with Montage 6. We scaled out the systems from 8 to 64 compute nodes, using 8 cores each. Indicated by the vertical scalability results (Figure 2a), we also show AMFS using only 4 cores each which is faster with 32 and 64 nodes. The results show that while both file systems achieve good horizontal scalability, MemFS leverages better performance.

Figure 2c shows the aggregate memory consumption for the two file systems with Montage 6. The measurements were taken at the end of each experiment presented in Figure 2b on 8 to 64 nodes. The graph shows the superior memory management of MemFS for all scales. Increased data usage of AMFS can be explained by its replication-on-read policy for improving data locality.

IV. 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] 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.