The Case for a Versatile Storage System

Samer Al-Kiswany, Abdullah Gharaibeh, Matei Ripeanu
Electrical and Computer Engineering Department, The University of British Columbia
{samera, abdullah, matei}@ece.ubc.ca

ABSTRACT

Storage systems in emerging large-scale (a.k.a. peta-scale) computing systems often introduce a performance or scalability bottleneck. To deal with these limitations we propose a new operational approach: versatile storage, an application-optimized and highly configurable storage system that harnesses node-local resources, is configured and deployed at application deployment time, and has a lifetime dependent on the application lifetime.

Our prototype evaluation, using synthetic and application-level benchmarks, on a small cluster as well as on a 96K processor machine, provides evidence that the versatile storage approach can bring valuable benefits to large scale deployments in terms of storage system performance and scalability.

Categories and Subject Descriptors

D.4.3 (**Operating Systems)**: File Systems Management - Distributed file systems. D.4.7 (**Operating Systems)**: Organization and Design - Distributed systems. C.4 (**Performance of Systems**) Design studies.

General Terms

Performance, Design, Experimentation.

Keywords

Versatile storage system, Storage system specialization, Dynamic deployment, High performance storage

1. INTRODUCTION

Today's large-scale computing systems (e.g., supercomputers, cloud computing infrastructures) aggregate thousands of computing nodes and offer ample computational power. These systems support applications that generate an intense storage workload. For instance, large-scale data-intensive scientific applications [1, 2] often use thousands of compute nodes to analyze terabytes of stored data. For such applications, the storage system *throughput* and *scalability* play a key role in the overall application performance. The risk is that the I/O system is the bottleneck for an expensive set of compute resources.

Moreover, the applications that use these resources are highly heterogeneous over multiple axes related to storage system usage patterns and required semantics. These axes include: *file granularity, read vs. write workload composition* (applications may produce read- or write-only workload), *data lifetimes, data durability requirements* (e.g., some files can be recomputed), *consistency requirements, data sharing levels* (e.g., sometimes thousands of nodes concurrently access the same data), and *security requirements* (e.g., in terms of authentication, integrity and confidentiality). Further, and equally important, data is rarely shared between applications.

A *one-size-fits-all* storage system that serves such diverse requirements while meeting the access throughput requirements of data-intensive applications is particularly complex and costly. Moreover, this approach often introduces a scalability bottleneck.

An alternative approach is *specialization*: that is, exploiting workload characteristics to optimize the data store for application-specific usage patterns. Google File System (GFS) [3] and PVFS [4] are only two examples of this approach: GFS optimizes for large datasets and append access patterns, while PVFS optimizes for sequential reads/writes to large datasets.

Apart from *specialization,* a second opportunity is present in the environments we target: *underutilized resources.* In batch-oriented computing systems, applications are often allocated a set of dedicated nodes for the duration of the application run. While a central storage system may struggle to provide reasonable storage performance, the node-local resources (storage space, IO channels, sometimes memory) and the interconnections are often underutilized yet 'close' to the running application.

Versatile storage is our solution to exploit these two opportunities. Versatile storage aggregates node-local resources to build a dedicated storage system that is optimized for the application usage patterns and has a lifetime coupled to the application's lifetime. MosaStore, our versatile storage system prototype, can aggregate the underutilized resources available on cluster/commodity nodes allocated to the application. Unlike specialized storage systems (e.g. GFS, PVFS) MosaStore provides a broader set of optimizations that target specific workloads and can be switched on/off and configured at deployment time.

The contribution of this paper is three fold: First, it proposes 'versatile storage': a storage-system-per-application deployment approach in which a highly configurable storage system is configured at deployment time as part of the application initialization script and has a lifetime tied with the lifetime of the application it supports. Multiple storage system instances may be deployed at the same time to serve concurrently running applications. Second, it presents a modular, configurable, and extensible storage system architecture that supports a set of application specific optimizations. Finally, it presents preliminary experience deploying and evaluating MosaStore, our versatile storage system prototype.

The rest of this paper presents the characteristics of the target workloads that motivate our approach (Section 2), derives the system requirements (Section 3), presents the MosaStore design (Section 4) and preliminary experience deploying it (Section 5), presents related work (Section 6), and concludes (Section 7).

2. WORKLOAD CHARACTERISTICS

To highlight the diversity of the workloads we target, and the opportunity for application-specific optimizations, this section presents three motivating scenarios for versatile storage, then discusses in detail the characteristics of the workloads we aim to support.

First, in a *data parallel processing* scenario, compute nodes perform the same task on mutually disjoint subsets of a larger dataset. For instance, BLAST [1], a popular bioinformatics application, searches a set of novel protein sequences in a large dataset of known sequences. Often the search set is partitioned over thousands of compute nodes where each node matches a subset of the proteins against the complete dataset, a scenario that leads to an intense read load for the storage nodes holding the dataset. This scenario is particularly demanding in large-scale deployments in which a compute node's local storage is not large enough to hold the complete dataset, preventing caching it locally and leading to a situation where all nodes access the dataset stored on the central storage for each protein sequence [5]. A versatile storage system that aggregates the nodes' local storage to cache the input dataset and is optimized for sequential read operations can dramatically reduce the stress on the central storage and increase system overall performance. While refactoring the application for a different partitioning mechanism is possible, we are looking for file-system level support that does not require application changes.

A second example is *checkpointing*. Long-running applications periodically write large snapshots to the storage system to capture their current state. In the event of a failure, applications recover by rolling-back their execution state to a previously saved checkpoint. The checkpoint operation and the associated data have unique characteristics [6]. First, checkpointing is a write-intensive operation. Second, checkpoint data is written once and often read only in case of failure. Finally, consecutive checkpoint images present the application state at consecutive time steps and hence may have a high level of similarity. A versatile storage system that can absorb the bursty checkpointing writes, reduce the data transfers and storage space usage by detecting similarities between checkpoint images, and asynchronously write the checkpoints to the central storage can increase the system overall performance.

Third, *workflow based processing*, used by a large number of scientific applications, is generally composed of three main phases: *stage-in* input-data from central storage to the compute nodes local storage, multiple *computation stages* that communicate through intermediate files, and *stage-out* the final results to the central storage. These three phases impose an intense workload on the storage system: The stage-in and stage-out phases are composed of sequential read-only, and sequential write-only, respectively, operations by all the compute nodes. The computation phase imposes an intense disjoint read/write access pattern. A shared versatile storage system that is deployed with the workflow control engine and aggregates the storage resources of allocated compute nodes can offer the following advantages: reduce the load on the central storage system in the stage-in phase by transferring significantly fewer copies of the input data to the aggregate storage space; reduce the overhead generated by the intermediate files' operations in the computation phase by keeping the intermediate files in the aggregate storage space, instead of the central storage; and, for applications composed of many small tasks (e.g., BLAST), increase the system performance by overlapping the transfer of completed tasks' results and the computation of the remaining tasks in the stage-out phase.

Workload Characteristics: We classify applications' storage workload along the following axes:

- *Data life-time*. While computation results generally need to be durably stored, some files (e.g., workflow's intermediate files and checkpoint images) are only temporary.
- *Read/Write composition*. Large-scale scientific applications often generate mostly read loads (e.g., bioinformatics sequence matching), or mostly write workloads (e.g., checkpointing), or different phases of the application generate different types of workloads (e.g., workflows).
- *Data compressibility*. Some scientific applications generate large outputs that are highly compressible by detecting similarities across files (e.g., checkpointing, visualization data, and Monte-Carlo simulations).
- *Locality*. Some applications exhibit high access locality, that is, the working sets of multiple application instances running on different nodes significantly overlap, while in other cases application instances running on different nodes have disjoint working sets.
- *Consistency requirements*. A large group of scientific applications do not require consistency guarantees. In the space defined so far, read-only workloads, such as the aforementioned bioinformatics application, are a clear and extreme example. Similarly, workloads with good locality leading to completely disjoint read/write set (e.g., workflows and checkpointing) are another example. Elsewhere in this space, the consistency requirements cannot be inferred from workload properties and depend entirely on the application semantics.

These characteristics can be exploited through different optimization techniques to enhance storage system performance with regard to different metrics. For instance, buffering can dramatically enhance throughput for write only workloads. Likewise, similarity detection mechanisms can save considerable storage space for highly compressible workloads.

However, it is important to note that different optimizations may negatively impact each other's performance. For instance, consistency mechanisms often have a negative impact on write throughput. Consequently, these optimizations do not generally coexist on the same data pipeline.

3. VERSATILE STORAGE SYSTEM DESIGN REQUIREMENTS

From the target computing systems and workload characteristics detailed above, we derive the following requirements for a versatile storage system design:

- *Easy to deploy*. A versatile storage system should be easy to deploy as part of the application startup script. Further, it should be able to access the system central storage for automatic data pre-fetching or storing persistent data or results.
- *Easy to integrate with applications*. The storage system should implement POSIX API to facilitate access to the aggregated storage space, without requiring changes to the application.
- *Easy to configure*. The storage system should be easy to configure and tune for an application workload and deployment environment. This includes ability to control local resource usage, in addition to controlling application-level storage system semantics, such as file lifetime, consistency and data reliability requirements.
- *Performance and scalability*. The storage system should efficiently use the node-local storage and networking resources to provide high performance access to the stored data, and it should be able to scale to support thousands of compute nodes.
- *Tunable security*. Applications should be able to tune the security level in terms of access control, data integrity, data confidentiality, and accountability. Further, the security mechanism should be compatible with the security infrastructure deployed on exiting production systems.
- *Range of consistency guarantees.* The storage system should provide a set of consistency models ranging from no-

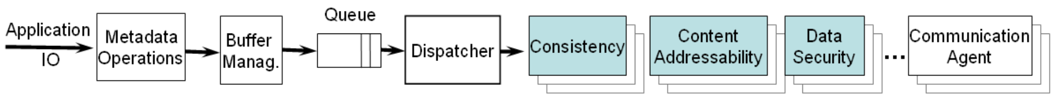

Figure 1. SAI configurable IO pipeline. A pipeline configuration example.

consistency (e.g., suitable for read only workloads), to session consistency, to application-specific consistency models.

- *Efficient storage of partially similar data.* The storage system should enable optimizations for workloads producing partially similar outputs, for example, by supporting versioning and content-based addressability.

4. MOSASTORE ARCHITECTURE

This section presents an overview of MosaStore and discusses the design of its main features.

4.1 Overview

MosaStore integrates three components: a metadata manager, a number of donor nodes that contribute storage space to the system, and the client-side System Access Interface (SAI). Datasets are fragmented into smaller *chunks* that are striped across donor nodes for fast storage and retrieval.

- *The metadata manager* maintains the entire system metadata (e.g., donor node status, file chunk distribution and dataset attributes). Similar to a number of other storage systems we choose a metadata service decoupled from stored data.
- *The donor nodes* contribute storage space (memory or disk based) to the system. Donor nodes interact with the manager to publish their status using soft-state registration, serve clients' chunk store/retrieve requests, and perform garbage collection.
- *The system access interface (SAI)* implements the mechanisms to access the storage space and client side optimizations including caching and content addressability functions. From an application perspective, the SAI provides two methods to access the storage system: a mountable kernel module that supports the POSIX file system API and a storage system library that facilitates direct integration with the application.

Data storage and retrieval operations are initiated by the client (the SAI) via the manager. To retrieve a file, the SAI first contacts the metadata manager to obtain the chunk-map (i.e., the location of all chunks corresponding to the file). Then, the actual transfer of chunks occurs directly between the storage nodes and the SAI. The write operation follows a similar protocol. After the completion of the write operation, the new data chunks are replicated asynchronously in the system.

The following subsections present the design of MosaStore IO pipeline (section 4.2), as well as that of the stages we have experimented with to date: read/write optimizations (section 4.3), content addressability (section 4.4), and configurable security (section 4.5).

4.2 The Configurable IO Pipeline

To support a broad set of optimizations, the data pipeline at the SAI is configurable (Figure 1). The pipeline includes a number of fixed, yet configurable, stages and an extensible set of optimization modules that can be enabled at configuration time.

The set of fixed modules (white modules in Figure 1) include the following modules: The metadata operations module that implements the mechanism to create/query metadata entries. The buffer management module that manages the read/write buffers. The IO request queue that effectively decouples the application from the request processing (i.e., the application is released after putting the IO request in the queue allowing the rest of the pipeline operations to perform asynchronously). The dispatcher module that creates multiple threads to handle the rest of the pipeline thus enabling parallel processing to harness the host's multicore capabilities and parallel striping to multiple donor nodes. Finally, the communication agent manages the network connection to the donor nodes at the end of the pipeline.

The rest of the pipeline is composed of a set of modules (e.g., content addressability, consistency, data security, pre-fetching, and data compression) whose existence, order in the pipeline, and specific parameters are entirely configurable (grey modules in Figure 1). Further, this set of optimization modules is extensible: MosaStore provides a generic API that new modules must implement. Moreover, the SAI implements a flow manager that orchestrates the processing of these pipeline modules based on the application provided configuration information.

To support the pipeline both the manager and the donor nodes implement an extensible request-processing engine to process the pipeline modules' requests. The extensibility of the request-processing engine is necessary to facilitate developing pipeline modules that require module-specific interaction with the manager and/or donor nodes to support system-wide operations (e.g., they may require access to metadata). The request-processing engine loads the developer-defined modules that implement request processing callbacks, dispatches requests to the appropriate callback, and provides a stable interface to expose manager or donor nodes' internal state that may be needed for request processing.

To configure the system at deployment time, the user specifies through a configuration file which modules (e.g., optimization for sequential access, content addressability) should be enabled and their specific configuration. This requires that the user has some information about the application's generated workload and is able to translate this information into specific configuration directives. While these tasks are definitely not trivial, they are beyond the scope of this paper.

4.3 Optimizations for High Throughput

The read/write optimization stage implements optimizations for sequential read/write access, a frequent pattern in our target workloads. For the read, MosaStore enables concurrent read ahead operations: the SAI fetches multiple data chunks at the same time to efficiently harness the node network connection. For writes, the SAI writes to a stripe-width of donors to achieve maximum throughput, and uses buffering to decouple the application write operations from the actual data transfer to achieve higher application perceived throughput.

4.4 Support for Content Based Addressability

Scientific applications often generate massive amounts of data, sometimes with high data similarity. This property can be used to reduce storage and data transfer requirements. Further, the management of these files is simplified by supporting versioning to maintain the tight relationship between related files. The challenge, however, is to offer similarity detection at runtime without operating system or application support. Addressing data by its content supports this feature in a natural way. Thus, MosaStore provides:

- *Content-based chunk naming.* MosaStore identifies data chunks by the hash of their content. The current version supports equal-sized chunks rather than variable size chunks whose boundaries are determined based on content as well (e.g., similar to LBFS [7]). The tradeoff is between the computational overheads to detect chunk boundaries based on content and potential savings in terms of storage and lower generated network traffic.

- *Support for copy-on-write and versioning*. MosaStore supports versioning and copy-on-write so that chunks that are identified as similar can be shared between different file versions. When a new version of a file is produced, only the new chunks need to be propagated to donor nodes. The new file version metadata will integrate the information of the new chunks and the chunks already stored.

4.5 Configurable Security

We aim to provide a number of configurable security levels [8] allowing the administrator to enable/disable the following security services: authentication, data confidentiality and integrity during transfers and while stored on the donor nodes, and accountability (the ability to identify malicious clients or donor nodes). The complexity of the design is increased by the fact that we aim to operate MosaStore in both *completely trusted* environments (e.g., a cluster where the applications have dedicated nodes) and *partially trusted environments* (e.g., a desktop grid where only the metadata manager is hosted on a trusted node).

Security is integrated in the pipeline as multiple stages: for instance, data channel security (i.e., transport integrity and confidentiality) is part of the underlying communication layer where it can be enabled or disabled, while stored data security (e.g., integrity and confidentiality) is supported as a developer-provided stage , that is a gray stage in Figure 1.

5. SYSTEM EVALUATION

The current prototype implements the pipeline described with all its fixed components and two optimization modules: content addressability and data security. We evaluated this prototype using a range of benchmarks that focus on the performance of each supported optimization independently. Due to space limitations we only present here a large-scale experiment that evaluates MosaStore's scalability and performance and demonstrates our ability to deploy and configure the versatile file system at application deployment time – we use a synthetic workload as well as a real workflow-processing application deployed on up to 96K processors.

The MosaStore prototype has been deployed on the BlueGene/P supercomputer at ANL – a peta-scale machine with around 160K processors, served by a GPFS storage system with 24 IO servers (each with 20Gbps network connectivity), and with a sustained IO rate of around 8GBps. The compute nodes are diskless and mount a RAM disk for the OS. A complete description of the application and the BlueGene/P platform is presented in [5]. One characteristic worth mentioning is that the application input data does not fit entirely in the memory of a single compute node.

In the following experiments MosaStore is deployed as part of the application startup script to aggregate the preciously little RAM-based storage space available at each compute node such that the input data can be staged-in and accessed locally by a pool of compute nodes. The deployment is configured for optimized sequential read/write access. As we argue in Section 2, a versatile storage system brings benefits to all three stages of a workflow-based application, mainly by offering a fast, intermediate data store co-located with the application that reduces the demand on the central storage.

Synthetic evaluation. Figure 2 presents the system throughput while running two synthetic applications each running on up to 96K processors. To simulate a real data-intensive application that generates intermediate files after computing for some time, each synthetic application writes 1MB of data to the storage system after waiting for 4sec or 32sec. The two applications differ only in the wait time and, consequently, generate different loads on the storage system. The applications are executed with three storage configurations: GPFS – the applications write directly to the GPFS deployed with the machine, GPFS+MosaStore (MS) – the applications write to MosaStore aggregated memory-based storage, and MosaStore asynchronously flushes the data to GPFS, and RAM – the applications write to the local, RAM-based disk. This configuration cannot be used in a real-world deployment (due to limited space and lack of support for data-aware scheduling), however it provides an upper bound for the achievable write throughput.

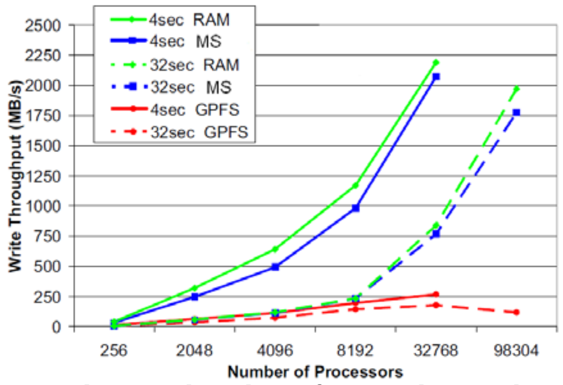

Figure 2. The write throughput of two synthetic applications (4sec and 32sec) running on up to 96K processors writing to: GPFS, GPFS+MosaStore(MS), and local RAM. (source [5])

From this experiment (Figure 2) we can derive two observations: First, in terms of *performance*, the application writing through MosaStore achieves throughput close to the maximum achievable write throughput (that achieved by directly writing to the RAM of independent nodes) in spite of using a centralized metadata service. Additionally, we note that MosaStore enables one order of magnitude write throughput increase compared to the deployed GPFS. Second, in terms of *scalability*, while GPFS performance peaks at less than 250MBps, MosaStore write throughput scales well with the number of nodes even under this intense workload. Two reasons explain this performance difference. First, while GPFS uses fixed number of dedicated storage nodes and IO paths, MosaStore is able to exploit more resources as the system incorporates more nodes. In particular, MosaStore is able to transparently exploit the RAM based storage at the compute nodes and the high bandwidth interconnect compared to GPFS with fewer network connections and disk based storage nodes. Second, unlike GPFS, MosaStore's IO path is optimized for sequential write operations with no consistency checks, avoiding unnecessary overheads. This preliminary result highlights the potential benefits the versatile storage approach can bring to large scale computing systems.

Application-level evaluation. MosaStore was used [5] to support DOCK6 [2], a compute bound bioinformatics application that screens drug compounds against metabolic protein targets. The protein dataset used by DOCK6 is larger than a single compute node's local storage. DOCK6 application is composed of three main phases: first, it reads input, computes the docking, and writes temporary results; second, it summarizes, sorts, and selects the results; and, third, it archives the results. We deployed a MosaStore instance per group of nodes, each group contains up to 256 nodes. This decision was made mainly to control the load on MosaStore's central manager, and to distribute the IO load on more storage nodes. The later point is desirable since the current prototype does not implement smart replication and data placement. The experiment runs DOCK6 on 8K processors. While MosaStore enables moderate improvements for the first (1.06x faster) and third (1.51x faster) application phases, it enables an 11.76x performance improvement for the second phase. This improvement is mainly explained by MosaStore's ability to store the temporary files produced by the first application phase locally and avoid shipping them to the central storage system. Overall, MosaStore enabled 1.52x performance improvement for the complete application. Further, running the first phase of DOCK6 workflow with MosaStore support on 96K processors achieved 1.12X application-level speedup, an expected moderate enhancement for a compute-bound stage.

6. RELATED WORK

Workload-optimized storage systems. Building storage systems geared for a particular class of I/O operations or for a specific access pattern is not uncommon. For example, GFS [3] optimizes for large datasets and append access; BAD-FS [9] optimizes for batch job submission patterns over wide area network connections; parallel file systems (PVFS [4], GPFS [10]) also target large datasets and provide high I/O throughput for parallel applications.

Versatile storage differs in its design and deployment goals. MosaStore aims to incorporate a broad set of optimization techniques, enable high configurability at deployment time, and support multiple applications through customized, per application deployment.

Contributory storage. A number of storage systems [11, 12] aggregate space contributions from collaborating nodes to provide a shared data store. Their basic premise is the availability of a large amount of idle disk space on nodes that are online for the vast majority of the time. The specific technical solutions vary widely as a result of different targeted deployment environments (local vs. wide-area networks) and workloads (e.g., read/write vs. read-only).

Aspect-oriented systems. Finally, our work is in the spirit of aspect-oriented system building, as different modules in MosaStore' data pipeline address different system needs. A recent related effort in this direction, the FLUXO [13] project, aims to separate Internet services logical functionality from the architectural decisions made to support performance, scalability and reliability. FLUXO approach is to profile the target application load and restructure the service at compile time, to include commonly used optimizations such as caching or replication.

7. SUMMARY AND FUTURE WORK

This paper proposes versatile storage: an operational approach for efficient resource usage in emerging large-scale computing systems. A versatile storage system implements a set of optimization techniques and is highly configurable at deployment time, such that application-specific workload can be exploited through dedicated, per-application deployments. Further, such an approach can be built at low cost by exploiting the underutilized node-local resources and interconnect bandwidth. We developed MosaStore, an initial versatile storage system prototype. Our preliminary evaluation indicates that MosaStore can bring valuable benefits to a diverse set of workloads.

Our future efforts will be focused on two main directions: First, we plan to complete the implementation and the evaluation of our system. We are extending the prototype with a set of modules that provide commonly used optimizations and plan to evaluate the versatile storage system prototype with a diverse set of applications. The experience thus gathered will help back up our extensibility and configurability claims, and equally important, will help better understand the degree to which the building blocks and the structure of the data-pipeline depend fundamentally on the optimization criteria at hand (e.g., performance vs. reliability vs. power). The second direction is to explore solutions to simplify and possibly automate the task of determining optimal versatile storage systems configurations. Past work on auto-tuning and autonomous systems offer good starting points in this direction.

8. ACKNOWLEDGMENTS

We thank Allan Espinosa of University of Chicago for running the DOCK6 experiment. We also thank the anonymous reviewers and our shepherd, Kimberly Keeton, for their insightful comments.

9. REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. Myers, et al., *Basic Local Alighnment Tool.* Molecular Biology, 1990. **215**: p. 403–410.

[2] *Overview of DOCK.* [cited 2009; http://dock.compbio.ucsf.edu/Overview_of_DOCK/index.htm

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung. *The Google File System*. in *19th ACM Symposium on Operating Systems Principles*. 2003. Lake George, NY.

[4] P. H. Carns, W. B. Ligon-III, R. B. Ross, and R. Thakur. *PVFS: A Parallel File System for Linux Clusters*. in *4th Annual Linux Showcase and Conference*. 2000. Atlanta, GA.

[5] Z. Zhang, A. Espinosa, K. Iskra, I. Raicu, et al. *Design and Evaluation of a Collective I/O Model for Loosely-coupled Petascale Programming*. in *Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS)*. 2008.

[6] S. Al-Kiswany, M. Ripeanu, S. Vazhkudai, and A. Gharaibeh. *stdchk: A Checkpoint Storage System for Desktop Grid Computing*. in *International Conference on Distributed Computing Systems (ICDCS '08)*. 2008. Beijing, China.

[7] A. Muthitacharoen, B. Chen, and D. Mazieres. *A Low-bandwidth Network File System. SOSP*. 2001. Banff, Canada.

[8] A. Gharaibeh, S. Al-Kiswany, and M. Ripeanu. *Configurable Security for Scavenged Storage Systems*. in *Workshop on Storage Security and Survivability (StorageSS)*. 2008.

[9] J. Bent, D. Thain, A. C.Arpaci-Dusseau, R. H. Arpaci-Dusseau, et al. *Explicit Control in a Batch-Aware Distributed File System*. *NSDI*.2004. San Francisco, California.

[10] F. Schmuck and R. Haskin. *GPFS: A Shared-Disk File System for Large Computing Clusters*. *FAST*. 2002.

[11] S. S. Vazhkudai, X. Ma, V. W. Freeh, J. W. Strickland, et al., *Constructing collaborative desktop storage caches for large scientific datasets.* ACM Transaction on Storage (TOS), 2006. **2**(3): p. 221 - 254.

[12] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. *Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs*. *SIGMETRICS*. 2000.

[13] E. Kıcıman, B. Livshits, and M. Musuvathi. *FLUXO: A Simple Service Compiler*. in *Workshop on Hot Topics in Operating Systems (HotOS)*. 2009.