# Towards High Performance Key-Value Stores through GPU-Accelerated Coding

Dongfang Zhao$^\star$, Kan Qiao$^{\dagger\star}$, Qi Guo$^\ddagger$, Iman Sadooghi$^\star$, and Ioan Raicu$^{\star\S}$

$^\star$Illinois Institute of Technology $^\dagger$Google $^\ddagger$Microsoft $^\S$Argonne National Laboratory

***Abstract*—This paper presents a GPU-accelerated distributed key-value store system. Conventional key-value stores are criticized by the costly replication due to significant space overhead and degraded I/O performance. We propose to break full-size replicas into smaller pieces and leverage GPUs to encode them with a few additional parities in parallel. The advantage of this approach is two-fold. First, the space overhead is greatly reduced because parity codes achieve the same availability as full-size replicas, but take less space. Second, the end-to-end I/O performance is significantly improved because the data parallelism is fully exploited by the massive number of GPU cores. We provide a detailed analysis of the proposed approach for distributed key-value stores, and implement a system prototype named Gest. Gest has been deployed and evaluated on a variety of testbeds from a commodity Linux cluster to a leadership-class supercomputer. Experimental results show that Gest achieves high data availability, high space efficiency, and high I/O performance, collectively at the same time.**

## I. Introduction

Cloud platforms (for example, Microsoft Azure, Amazon EC2) provide a simple yet versatile hashtable-like API to its underlying distributed storage such as `set(key,value)` and `value` $\leftarrow$ `get(key)`. The hashtable API relaxes the conventional POSIX interface, simplifies otherwise complicated file operations, enables a unified I/O interface to a large variety of applications, thus gains increasing popularity. Many storage subsystems underneath are implemented as distributed key-value stores, which, despite slight implementation differences, usually assign the file name (internally as a unique ID) as the key and the file content (or, blob) as the value.

The state-of-the-art approach for distributed key-value stores to achieve high availability is replication: several remote replicas are created and updated when the primary copy is touched. This approach is easy to implement and provides a quick recovery of failures. Nevertheless, redundant file replicas pose the following limitations: (1) significant space overhead, (2) additional disk I/O, and (3) more network bandwidth consumption. For instance, if every value keeps two replicas, the space overhead is roughly 200% along with tripled disk I/O and network traffic. Thus the combined overhead staggers the overall performance and cost of key-value stores particularly for today's data-intensive applications.

While recent studies [9, 21, 24, 29, 33, 38] focus on algorithms, protocols, and models for better manipulating replicas, this work is orthogonal to them as we propose to completely replace full-size replication by more space-efficient parity coding along with GPU acceleration. Specifically, instead of sending out full-size replicas, we split the value into smaller chunks, transform them with additional parities with GPUs, and disperse the encoded chunks onto remote nodes. The recovery procedure is the reverse of the above: collecting the encoded chunks, merging them, and decoding the merged (encoded) data back into the original file concurrently by GPUs. The advantages of this approach is two-fold as follows.

First, the additional space required by the redundant data is greatly reduced. Recall that the conventional replication approach makes a full duplication of the primary copy. Thus, if each primary copy has two replicas (many production systems have this default setup), the space efficiency is as low as 33%. On the other hand, if the data is split up into $m$ chunks that are then encoded with two additional parities, the system can also tolerate two failures as two additional full-size replicas. The space efficiency with parities, however, is higher: $\frac{m}{m+2}$, as long as $m$ is set to a nontrivial value larger than 1. We assume that the total number of nodes is at least $(m+2)$ to ensure that each chunk has its own node to reside.

Second, the data parallelism is heavily exploited by the massive number of GPU cores and multiple network links. The conventional full-size replica is transferred from the node of the primary copy to the remote node in a serial fashion: a single process or thread sends the entire file over the network. This approach incurs low utilization of both the computing resources and network bandwidth. Modern CPUs have multiple cores, but in this case only one of them works on sending the replica. Similarly, a lot of network traffic occurs on the link between the primary copy and the remote node holding the replica, but other network links are barely involved in the replication process. In contrast, our proposal—chopping the file into smaller chunks followed by coding with additional parities—leverages the many-cores available in GPUs and utilizes multiple network paths between the primary copy and several (i.e., $m+2$) remote nodes.

To make matters more concrete, Fig. 1 shows an example of a distributed key-value store deployed on a 6-node cluster [1] (`Node 1` to `Node 6`). In this example, value `V1` is split into four chunks (`Chunk 1` to `Chunk 4`), and encoded with two parities (`Parity 1` and `Parity 2`) by six GPU cores. The encoded chunks are then transferred to six remote nodes (`Node 1` to `Node 6`). The two parities allow for up to two

[1]Consider `Node 0` as the master or login node (for example, preprocessing, scheduling, metadata management) only in this example.

failed nodes out of the total six nodes. The space utilization rate is thus $\frac{4}{2+4} = 67\%$ (as opposed to $33\%$ in the full-size replication); and the transfer time is roughly 4X faster because each chunk is roughly a quarter of the original value `V1` in size.

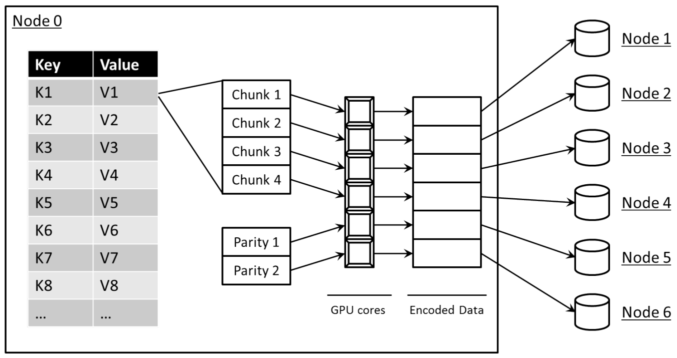


Fig. 1. A distributed key-value store, which can tolerate up to two failures, is deployed on six nodes with GPU acceleration.

We do not show the recovery procedure, as it is as straightforward as the reversed process of Fig. 1; yet one might argue that the recovery stage could be costly comparing with the full-size replication: decoding the chunks and merging them back into the original copy could possibly take more time than simple data replication. We believe (and will justify in evaluations, such as Fig. 13) that the recovery phase is not a performance bottleneck because: (1) the decoding process is accelerated by GPUs, and (2) the merging process is conducted concurrently on multiple network paths. In fact, literature [5] shows that the GPU decoding rate is in the order of O(1 GB/s) that is an order of magnitude faster than the disk I/O throughput (normally in the order of O(100 MB/s)). Thus, if the GPU computation is pipelined with I/O, then the system performance is not throttled by the decoding procedure.

This work provides detailed analysis of the proposed mechanism for achieving high performance of distributed key-value stores while maintaining high availability as the conventional data replication. In addition, we design and implement a system prototype of a distributed key-value store named Gest: a GPU-accelerated encoded store. To the best of our knowledge, Gest is the first distributed key-value store system with GPU-accelerated encoding. Gest has been deployed and evaluated on a variety of platforms from a commodity Linux cluster to a leadership-class supercomputer. Experiment results demonstrate that GPU-boosted coding is a promising approach towards data availability, space efficiency, and I/O performance collectively at the same time.

In summary, this paper makes the following contributions:

- *Proposal of an unconventional mechanism to achieve high data availability, low space overhead, and improved I/O performance in distributed key-value stores*
- *Detailed analysis of the proposed mechanism with a large space of parameters*
- *Design and Implementation a key-value store system prototype, Gest, with GPU-accelerated coding*
- *Extensive evaluation of Gest on a variety of testbeds from clusters to supercomputers*

The remainder of this paper is structured as follows. Section II presents some background of distributed key-value stores, parity coding, and GPU computing. We describe the system design of Gest in Section III. Section IV analyzes the performance of the proposed mechanism, and Section V reports the experimental results of Gest. We finally conclude this paper in Section VII.

## II. Background

### A. Distributed Key-Value Store

A distributed key-value storage usually has a simpler interface than a POSIX filesystem. Most file operations are implemented by `set(key, value)` and `value` $\leftarrow$ `get(key)`, rather than conventional POSIX APIs such as `fh` $\leftarrow$ `fopen(fname)`, `fwrite(ptr, sz, cnt, fh)`, `fread(ptr, sz, cnt, fh)`, `fclose(fh)`, and so forth. It greatly simplifies the code complexity and allows developers focus more on the business logic rather than the I/O syntax. Thus major web service vendors have started employing key-value stores in production system. For example, Dynamo [8] is a key-value storage system that many Amazon's web services (AWS) rely on.

Recently, distributed key-value stores have received plenty of research interests to improve the performance of large-scale applications. For instance, Debnath et al. present two key-value systems (SkimpyStash [7], FlashStore [6]) on memory-class storage and report their performance improvement for the XBOX LIVE Primetime online multi-player game.

All of aforementioned systems use data replication to achieve reliability. In spite of its simplicity, data replication maintains several fullsize replicas resulting in low space efficiency and consequently high I/O cost. This work is focused on improving space efficiency and I/O performance while retaining high data reliability in distributed key-value stores.

### B. Erasure Coding

Erasure coding has been studied by the computer communication community since the 1990's [30, 37], as well as in storage and filesystems [17, 23, 35, 48]. Plank et al. [35] make a thorough review of erasure libraries. The idea is straightforward: a file is split into $k$ chunks and encoded into $n > k$ chunks, where any $k$ chunks out of these $n$ chunks can reconstruct the original file. We denote $m = n - k$ as the number of redundant chunks (parities). Each chunk is supposed to reside on a distinct disk. Weatherspoon and Kubiatowicz [47] show that for total $N$ machines and $M$ unavailable machines, the availability of a chunk (or replica) $A$ can be calculated as

$$A = \sum_{i=0}^{n-k} \frac{\binom{M}{i}\binom{N-M}{n-i}}{\binom{N}{n}}.$$

Fig. 2 illustrates what the encoding process looks like. At first glance, the scheme looks similar to file replication,

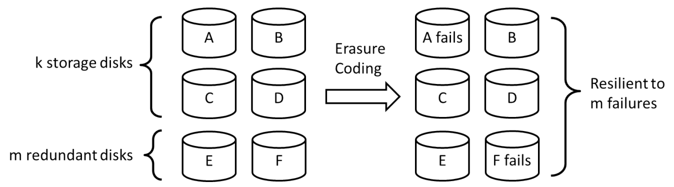
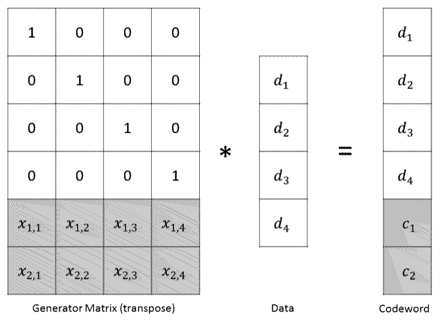


Fig. 2. Encoding $k$ chunks into $n = k + m$ chunks so that the system is resilient to $m$ failures

as it allocates additional disks as backups. Nevertheless, the underlying rationale of erasure coding is completely different from file replication for its complex matrix computation. As a case in point, one popular erasure code is Reed-Solomon coding [36], which uses a generator matrix built from a Vandermonde matrix to multiply the $k$ data to get the encoded $k + m$ codewords, as shown in Fig. 3.



$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} \\ x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} \end{bmatrix} * \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ c_1 \\ c_2 \end{bmatrix}$$

Fig. 3. Encoding 4 files into 6 codewords with Reed-Solomon coding

Compared to data replication, erasure coding has three important features.

First, erasure coding offers higher space efficiency, defined as $\frac{k}{n}$. This is because redundant parities are smaller than the file itself. Tanenbaum and Steen [44] report that erasure coding outperforms data replication by 40% - 200% in terms of space efficiency.

Second, erasure coding consumes less network bandwidth, because we need not send the entire files but only fractions of them (i.e. parities). This feature is critical in limited network resources, e.g. geographically dispersed Internet-connected Cloud computing systems built with commodity hardware.

The last and often underrated advantage is security. Rather than copying the intact and non-encrypted file from one node to another, erasure coding chops the file into chunks, then encodes and disperses them to remote nodes. This process is hard to reverse if the encoding matrix is wisely chosen. Moreover, erasure coding-based data redundancy guarantees data security with $(k-1)$ compromised nodes because the minimal number of chunks to restore the original file is $k$. In contrast, a simple file replication cannot tolerate any compromised nodes; if one node (with the replica) is compromised, the entire file is immediately compromised. Therefore, for applications with sensitive data, erasure coding is the preferred mechanism over replication.

The drawback of erasure coding stems from its computation overhead. It places an extensive burden on the computing chips, so it is impractical for production storage systems. This is one of the reasons why prevailing distributed storage systems (e.g. Hadoop distributed file system [40], Google file system [15]) prefer data replication to erasure codes.

## C. GPU Computing

The graphics processing unit (GPU) was originally designed to rapidly process images for the display. The nature of image manipulations on displays differs from tasks typically performed by the CPU. Most image operations are conducted with single instruction and multiple data (SIMD), where a general-purpose application on a CPU takes multiple instructions and multiple data (MIMD). To meet the requirement of computer graphics, GPU is designed to have many more cores on a single chip than CPU, all of which carry out the same instructions at the same time.

The attempt to leverage GPU's massive number of computing units can be tracked back to the 1970's in [11]. GPUs, however, did not become popular for processing general applications due to its poor programmability until GPU-specific programming languages and frameworks were introduced such as OpenCL [42] and CUDA [32]. These tools greatly eased the development of general applications running on GPUs, thus opened the door to improving applications' performance with GPU acceleration, which are usually named general-purpose computing on graphics processing units (GPGPU). GPGPU gains tremendous research interest because of the huge potential to improve the performance by exploiting the parallelism of GPU's many-core architecture as well as GPU's relatively low power consumption, as shown in [4, 28].

Table I shows a comparison between two mainstream GPU and CPU devices, which will also be used in the testbeds for evaluation later in this paper. Although the GPU frequency is only about 30% of CPU, the amount of cores outnumbers CPU by $\frac{384}{8} = 48X$. So the overall computing capacity of GPU is still more than one order of magnitude higher than CPU. This GPU's power consumption should also be noted; only $\frac{0.91}{15.63} = 5.8\%$ of CPU.

TABLE I
COMPARISONS OF TWO MAINSTREAM GPU AND CPU

| Device | Nvidia GeForce GT 640 | AMD FX-8120 |
|---|---|---|
| Number of Cores | 384 | 8 |
| Frequency (MHz) | 900 | 3100 |
| Power (W / core) | 0.91 | 15.63 |

## III. SYSTEM DESIGN

An overview of Gest architecture is shown in Fig. 4. Two services are installed on each Gest node: metadata management and data transfer. Each instance of these two services on a particular node communicates to other peers over the network when requested metadata or files cannot be found on the local node.

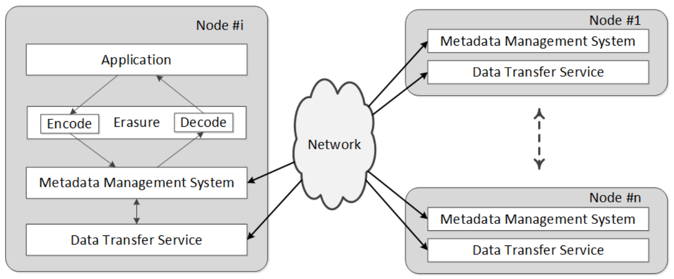


Fig. 4. Architectural overview of Gest deployed on an $n$-nodes distributed system. End users run applications on the $i^{th}$ node where files are encoded and decoded by coding algorithms.

To make matters more concrete, Fig. 5 illustrates the scenario when writing and reading a file for $k = 4$ and $m = 2$. On the left hand side when the original file (i.e. `orig.file`) is written, the file is chopped into $k = 4$ chunks and encoded into $n = k+m = 6$ chunks. These 6 chunks are then dispersed into 6 different nodes after which their metadata are sent to the metadata hashtable, which is also physically distributed across these 6 nodes. A file read request (on the right hand side) is essentially the reversed procedure of a file write: retrieves the metadata, transfers the chunks, and decodes the file.

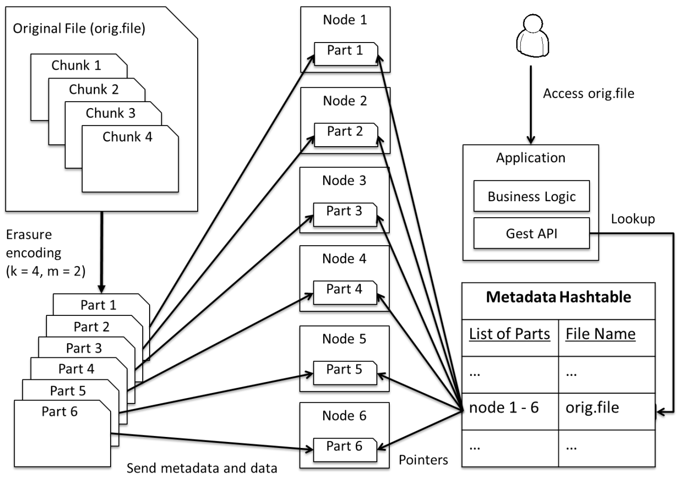


Fig. 5. An example of file writing and reading on Gest

## A. Metadata Management

The traditional way of handling metadata for distributed systems is to manipulate them on one or a few nodes. The rational is that metadata contains only high level information (small in size), so a centralized repository usually meets the requirement. Most production distributed storage systems employ centralized metadata management, for instance the Google file system [15] keeps all its metadata on the master node. This design is easy to implement and maintain, yet exposes a performance bottleneck for the workloads generating a large amount of small files: the metadata rate from a great number of small files can easily saturate the limited number of metadata servers.

In contrast, we implement Gest's metadata management system in a completely distributed fashion. Specifically, all meatdata are dispersed into a distributed hashtable (ZHT [27]). While there are multiple choices of distributed hashtable implementations such as Memcached [14] and Dynamo [8], ZHT has some features that are crucial to the success of serving as a metadata manager: ZHT is implemented in C/C++, takes the lowest routing time, and supports both persistent hashing and dynamic membership. ZHT is installed as daemon services on Gest nodes.

This paragraph gives a high level overview of ZHT to make this paper self-contained; more details can be found in [27]. ZHT has a similar ring-shaped look as the traditional DHT [41]. The node IDs in ZHT can be randomly distributed across the network. The correlation between different nodes is computed with some logistic information like IP address. The hash function maps a string to an ID that can be retrieved by a `lookup(k)` operation at a later point. Besides common key-value store operations like `insert(k,v)`, `lookup(k)`, and `remove(k)`, ZHT also supports a unique operation, `append(k,v)`, which we have found quite useful in implementing lock-free concurrent write operations. The replicas in ZHT should not be confused with the file replicas in Gest; ZHT replicas are used for metadata fault tolerance, and have no direct impact to Gest's reliability.

In Gest, clients have a coherent view of all the files (i.e. metadata) no matter if the file is stored in the local node or a remote node. That is, a client interacts with Gest to inquiry any file on any node. This implies that applications are highly portable across Gest nodes and can run without modifications or recompiling. The metadata and data on the same node, however, are completely decoupled: a file's location has nothing to do with its metadata's location.

Besides the conventional metadata information for regular files, there is a special flag to indicate if this file is being written. Specifically, any client who requests to write a file needs to acquire this flag before opening the file, and will not reset it until the file is closed. The atomic compare-swap operation supported by ZHT guarantees file's consistency for concurrent writes.

## B. Data Transfer

As a distributed system, Gest migrates data back and forth across the network. Ideally, data transfer should be efficient and reliable. User Datagram Protocol (UDP) is efficient in transferring data, but is an unreliable protocol. Transmission Control Protocol (TCP), on the other hand, is reliable but has a relatively low efficiency.

We have developed our own data transfer service – Gest Data Transfer service (GDT) on top of UDP-based Data Transfer (UDT) [16], which is a reliable UDP-based application level data transport protocol for distributed data-intensive applications. UDT adds its own reliability and congestion control on top of UDP, thus delivers higher transfer rate than TCP. Similarly to ZHT, GDT is installed as a daemon service on each Gest node.

### C. Coding Algorithms

Besides daemon services running at the back end, Gest plugs in encoding and decoding modules on the fly. Currently, we support two built-in libraries Jerasure [34] and Gibraltar [5] as the default CPU and GPU libraries, respectively. Gest is implemented to be flexible enough to support more libraries.

Jerasure is a C/C++ library that supports a wide range of erasure codes: Reed-Solomon coding, Minimal Density RAID-6 coding, Cauchy Reed-Solomon coding, and most generator matrix coding. One of the most popular codes is the Reed-Solomon encoding method, which has been used for the RAID-6 disk array model. This coding can either use Vandermonde or Cauchy matrices to create generator matrices.

Gibraltar is a Reed-Solomon coding library for storage applications. It has been demonstrated to be highly efficient when tested in a prototype RAID system. This library is known to be more flexible than other RAID standards; it is scalable with parity's size of an array. Gibraltar has been created in C using Nvidia's CUDA framework.

### D. Workflows

When an application writes a file, Gest splits the file into $k$ chunks. Depending on which coding library the user chooses to use, these $k$ chunks are encoded into $n = k + m$ chunks, which are sent to $n$ different nodes by GDT.

At this point the data migration is complete, and we will need to update the metadata information. To do so, ZHT on each of these $n$ nodes is pinged to update the file entries. This procedure of metadata update on the local node is conducted by an in-memory hashmap whose contents are asynchronously persisted to the local disk.

Reading a file is just the reversed procedure of writing. Gest retrieves the metadata from ZHT and uses GDT to transfer (any) $k$ chunks of data to the node where the user makes the request. These $k$ chunks are then decoded by the user-specified library and restored into the original file.

### E. Pipeline

Because the encoded data are buffered, GDT can disperse $n$ encoded chunks onto $n$ different nodes while the file chunks are still being encoded. This pipeline with the two levels of encoding and sending allows for combining the two costs instead of summing them, as described in Fig. 6.

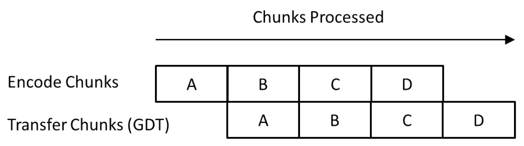


Fig. 6. Pipelining of encoding and transferring for a write operation in Gest

### F. User Interface

Gest provides a completely customizable set of parameters for the applications to tune how Gest behaves. In particular, users can specify which coding library to use, the number of chunks to split the file (i.e., $k$), the number of parity chunks (i.e., $m = n - k$), the buffer size (default is 1 MB), and the like.

## IV. Analysis

This section presents the analysis of the proposed mechanism for achieving fault tolerance in distributed key-value stores. In particular, we show how the parameters quantitatively affect the system in terms of space utilization and I/O performance. Evaluation on the real system will be presented later in Section V.

### A. Assumptions

We assume the multiple paths between the primary copy and the remote nodes have no interference. Therefore, if a full replica is split into 4 chunks and sent to 4 different nodes concurrently, then the transfer time is roughly reduced to 25% comparing with the full-size replication (assuming the data are already loaded into memory). As mentioned before, the full-size replica is transferred in a serialized manner.

For the encoding and decoding processes, we do not distinguish between the rates between both. This is because literature [5] shows that the difference between the two processes is marginal. It should be noted that this assumption only holds for this analysis section; we will report the performance difference between encoding and decoding procedures in Section V.

We also assume the parity code is in the same size of the chunk. In practice, this is the case of most coding algorithms. It also greatly simplifies the analysis in this section.

### B. Parameters

We consider the following parameters when analyzing the abstraction model of Gest. The size of the primary copy is denoted by $s$. The primary copy is split into $k$ chunks. The number of tolerable failures should be the same as the number of parities in Gest, which is denoted by $m$. For example, if we request that Gest is resistant to two failed nodes, then $m$ should be set to 2. The coding throughput (both encoding and decoding) is $c$. The network bandwidth is indicated by $b$. All these parameters are summarized in Table II.

TABLE II
PARAMETERS OF GEST ENVIRONMENT

| Variable | Unit | Meaning |
|---|---|---|
| $n$ | Number | Number of nodes |
| $s$ | Byte | Size of the primary copy |
| $k$ | Number | Number of chunks |
| $m$ | Number | Number of parities |
| $c$ | Byte / second | Coding rate |
| $b$ | Byte / second | Network bandwidth |

### C. Space Utilization

As discussed before, the space utilization (or, storage efficiency) of conventional data replication is

$$E_{rep} = \frac{s}{s \cdot m} = \frac{1}{m}$$

The space utilization for Gest is, however, higher:

$$E_{gest} = \frac{s}{s \cdot \frac{k+m}{k}} = \frac{k}{k+m}$$

Note that, in practice $m$ is a small number (for example, 2) and $k$ is usually set to $n - m$. By doing this, all the nodes are involved in the data redundancy procedure. Therefore the storage efficiency of Gest can be also expressed in an alternative way (assuming all nodes participate the coding process):

$$E_{gest} = \frac{n-m}{n} = 1 - \frac{m}{n}$$

A variant of Gest is to keep one primary copy and only apply the coding on replicas. This is for those applications having many read requests. So an intact copy will avoid the frequent decoding procedures. Indeed, such a full-size copy is to trade some space for the improved file-read performance. In this case, the storage efficiency is

$$E_{gest} = \frac{s}{s + s \cdot \frac{k+m-1}{k}} = \frac{k}{2k+m-1}$$

Similarly, if we assume $k = n - m$, then

$$E_{gest} = \frac{n-m}{2 \cdot (n-m) + m - 1} = 1 - \frac{n-1}{2n-m-1}$$

### D. End-to-End I/O Performance

Both conventional replication and Gest need to load the primary copy from disk to memory, so we do not differentiate them. The difference lies in two parts. First, Gest introduces computational overhead when encoding the data. Second, Gest reduces the network transfer time since smaller chunks are migrated in parallel.

Specifically, the time to transfer the full-size replicas is

$$Time_{rep} = \frac{m \cdot s}{b}$$

Gest, on the other hand, needs to take both the GPU encoding time and the transfer time into account for each encoded chunk:

$$Time_{gest} = \frac{s}{k \cdot c} + \frac{s}{k \cdot b}$$

where the first term represents the GPU coding and the second one is for network transfer.

Therefore, the speedup is

$$Speedup = \frac{Time_{rep}}{Time_{gest}} = \frac{\frac{m \cdot s}{b}}{\frac{s}{k \cdot c} + \frac{s}{k \cdot b}} = \frac{m \cdot k \cdot c}{b + c}$$

If we look at $b$ and $c$ in practice when GPU is leveraged, we usually have $b << c$. For example we will show in later evaluation part (Fig. 8 and Fig. 9) that for small $m$'s the GPU coding throughput is higher than 1 GB/s (as opposed to O(100 MB/s) for mainstream hard disks). Consequently, the speedup can be expressed as

$$Speedup \approx m \cdot k$$

This is also the case when the coding and transfer is pipelined; the $Time_{gest}$ term is essentially degraded to $\frac{s}{k \cdot b}$.

## V. Evaluation

### A. Test Beds

Gest has been deployed on three testbeds: a Sun-Oracle cluster (HEC), a high-performance GPU cluster (Sirius), and an IBM Blue Gene/P supercomputer (Intrepid [19]). Each HEC node has 8 GB RAM, dual AMD Opteron quad-core processors (8-cores at 2 GHz), and an OS with Linux kernel version is 2.6.28.10. Each Sirius node has an 8-core 3.1 GHz AMD FX-8120 CPU along with a 384-core 900 MHz Nvidia GeForce GT 640 GPU, and 16 GB RAM. Sirius' operating system is OpenSUSE, compiled by Linux kernel 3.4.11 with CUDA version 5.0 installed. Intrepid has 40K-nodes each of which has quad core 850 MHz PowerPC 450 processors and runs a light-weight Linux with 2 GB RAM.

All experiments are repeated at least five times, or until results become stable (i.e. within 5% margin of error). The reported numbers are the average of all runs. Caching effect is carefully precluded by reading a file larger than the on-board memory before the measurement.

### B. Experiment Design

We compare the conventional file replication to erasure coding algorithms of different parameter combinations at different scales. The list of candidate mechanisms is summarized in Table III, along with the number of chunks for both the original file and the redundant data.

TABLE III
List of data redundancy mechanisms considered in Gest

| Mechanism Name | Chunks of Original File | Chunks of Redundant data |
|---|---|---|
| Replica | 1 | 2 |
| Erasure1 | 3 | 5 |
| Erasure2 | 5 | 3 |
| Erasure3 | 11 | 5 |
| Erasure4 | 13 | 3 |
| Erasure5 | 27 | 5 |
| Erasure6 | 29 | 3 |

For file replication, the "chunks of the original data" is the file itself, and the "chunks of redundant data" are plain copies of the original file. We choose 2 replicas for file replication as the baseline, since this is the default setup of most existing systems. Therefore we see `<Replica, 1, 2>` in Table III.

Similarly, Gest with different erasure-coding parameters are listed as `Erasure[1..6]`, along with different file granularity and additional parities. We design experiments at different scales to show how Gest scales. Specifically, every pair of erasure mechanisms represents a different scale: `Erasure[1,2]` for 8-nodes, `Erasure[3,4]` for 16-nodes, and `Erasure[5,6]` for 32-nodes. For example, tuple `<Erasure6, 29, 3>` says that we split the original file into 29 chunks, encode them with 3 additional parities, and send out the total 32 chunks into 32 nodes.

The numbers of redundant parities (i.e. "chunks of redundant data") for `Erasure[1..6]` are not randomly picked, but in accordance with the following two rules. First, we are only interested in those erasure mechanisms that are more reliable than the replication baseline, because our goal is to build a more space-efficient and faster key-value store without compromised reliability. Therefore in all erasure cases, there are at least 3 redundant parities, which are more than the replica case (i.e. 2). Second, we want to show how different levels of reliability affect the space efficiency and I/O performance. So there is one additional configuration for each scale: the redundant parities are increased from 3 to 5.

## C. Data Reliability and Space Efficiency

Fig. 7 shows the tolerable failures (i.e. the number for failed nodes that the system can tolerate) and space efficiency for each of the 7 mechanisms listed in Table III. The tolerable failures (histograms) of `Erasure[1..6]` are all more than `Replica`, so is the space efficiency. Thus, in addition to comparable data reliability, erasure codes outperform data replication in terms of both reliability and efficiency. Before we investigate more about performance in §V-E, the following conclusions are drawn from Fig. 7.

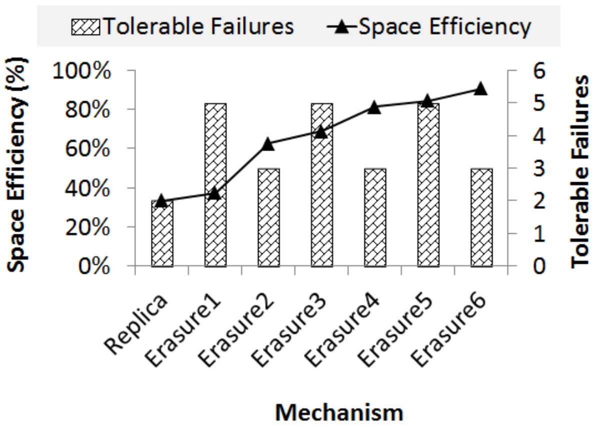


Fig. 7. Data reliability and space efficiency

First, a larger scale enables higher space efficiency. This is somewhat counter-intuitive, as it is a well-accepted practice to collect data to a small subset of nodes (e.g. collective I/O buffers small and dispersed I/O to reduce the number of small I/Os). Fig. 7, however, demonstrates that when redundant parity stays the same, space efficiency is monotonic increasing on more nodes. The reason is that with more nodes the redundant parity is in finer granularity and smaller in size. Therefore less space is taken by the redundant data.

Second, for a specific erasure code at a particular scale, reliability and efficiency are negatively correlated. For example, if we increase tolerable failures from 3 to 5, the space efficiency goes from 65% down to 35% (i.e. `Erasure2`→`Erasure1`). This is understandable, as increasing parities take more space.

## D. Coding Rate

This section evaluates the encoding and decoding rates of computing devices in our test beds. The encode and decode throughput of Jerasure (for CPU) and Gibraltar (for GPU) are plotted in Fig. 8 with $m$ increasing from 2 to 128 while keeping fixed storage efficiency = 33%. The buffer size is set to 1 MB. In all cases, with larger $m$ values, the throughput decreases exponentially. This is because when the number of parity chunks increases, encoding and decoding take more time which reduces the throughput.

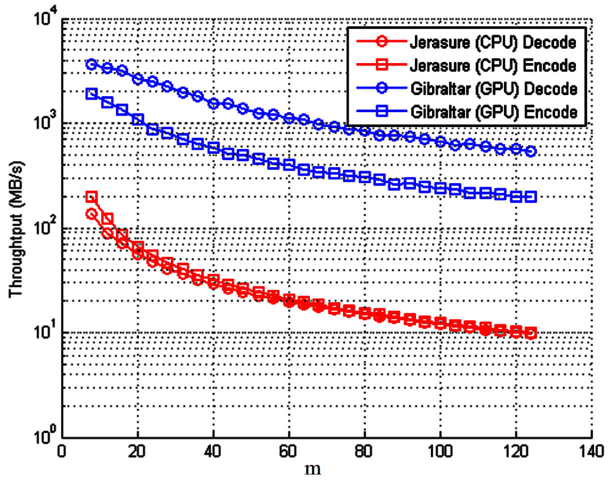


Fig. 8. Throughput with buffer size = 1 MB, storage efficiency = 33%

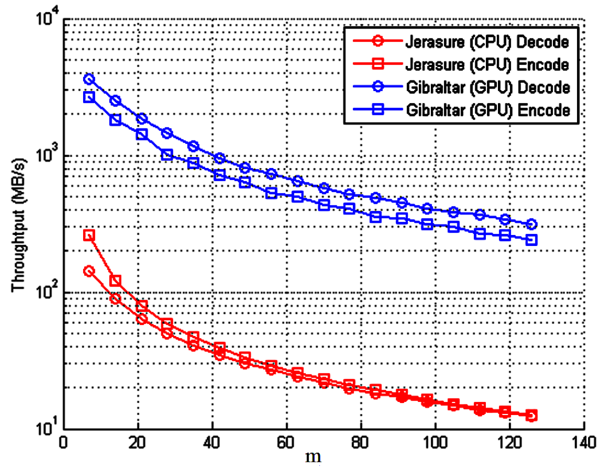


Fig. 9. Throughput with buffer size = 1 MB, storage efficiency = 75%

We then change the storage efficiency to storage efficiency = 75% and measure the throughput with different $m$ values in Fig. 9. Similar observations and trends are found just like the case for storage efficiency = 33%.

From both experiments above we observe a significant gap between Gibraltar and Jerasure. There is more than 10X speedup with Gibraltar, which suggests that GPU-based erasure coding would likely break through the CPU bottleneck.

## E. I/O Performance

We compare the read and write throughput of all mechanisms listed in Table III on the HEC cluster at 8-nodes, 16-nodes, and 32-nodes scales. The files to be read and written

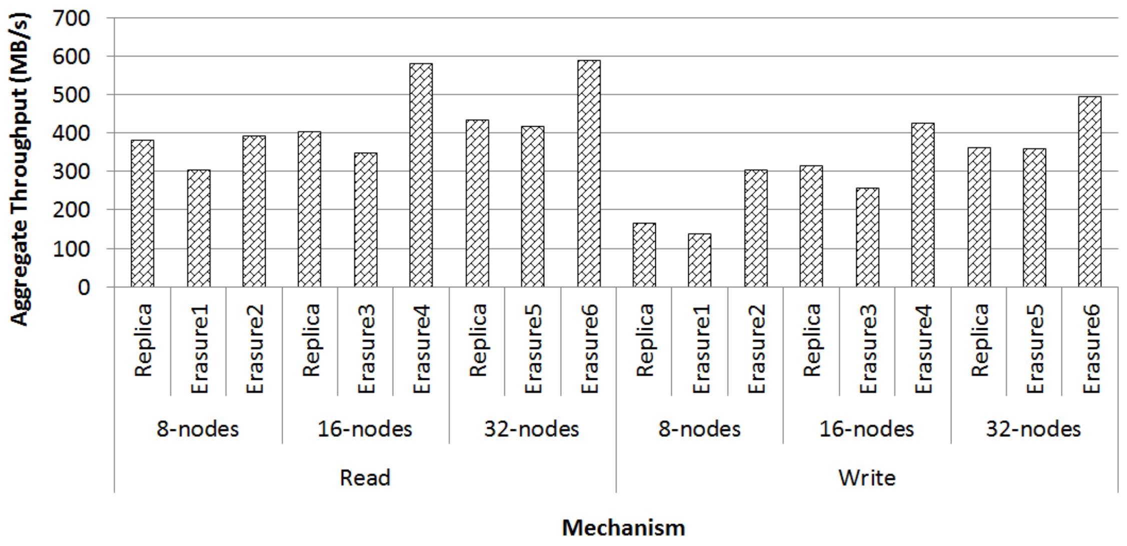


Fig. 10. Performance on the HEC cluster

are 1GB per node, with block size 1MB. Fig. 10 shows the results.

One important observation from Fig. 10 is the promising performance by erasure coding even on CPUs. In many cases (e.g. file read on 16-nodes with Erasure4), Gest delivers higher throughput than the replication counterpart. This is because replication uses more network bandwidth: two extra full-sized replicas introduce roughly a double amount of data to be transferred than erasure coding. We will discuss how GPUs further improve Gest performance in Fig. 13.

Fig. 10 also shows that, besides the number of nodes, the number of redundant parities greatly impacts the I/O performance. Simply increasing the number of nodes does not necessarily imply a higher throughput. For instance, `Erasure2` on 8 nodes delivers higher I/O throughput than `Erasure3` on 16 nodes.

It is worth mentioning that the promising erasure-coding results from HEC should be carefully generalized; we need to highlight that the CPUs of HEC are relatively fast – 8 cores at 2 GHz. So we wonder: what happens if the CPUs are less powerful, e.g. fewer cores at a lower frequency?

To answer this question, we deploy Gest on Intrepid, where each node only has 4 cores at 850MHz. For a fair comparison, we slightly change the setup of last experiment: the replication mechanism makes the same number of replicas as the additional parities in erasure coding. That is, the reliability is exactly the same for all replication- and erasure-based mechanisms. Moreover, we want to explore the entire parameter space. Due to limited paper space, we only enumerate all the possible parameter combinations with constraint of 8 total nodes, except for trivial cases of a single original or redundant chunk. That is, we report the performance in the following format `(file chunks: redundant parities)`: (2:6), (3:5), (4:4), (5:3), and (6:2); we are not interested in (1:7) and (7:1), as the former is identical to 7 replicas and the latter to 1 replica.

As shown in Fig. 11, for all the possible parameters of erasure coding, Gest is slower than file replication. As a side note, the throughput is orders of magnitude higher than other testbeds because Intrepid does not have local disk and we run the experiments on RAM disks. This experiment confirms our previous conjecture on the importance of computing capacity to the success of Gest. After all, the result intuitively makes sense; a compute-intensive algorithm needs a powerful CPU. This, in fact, leads to one purpose of this paper: what if we utilize even faster chips, e.g. GPUs?

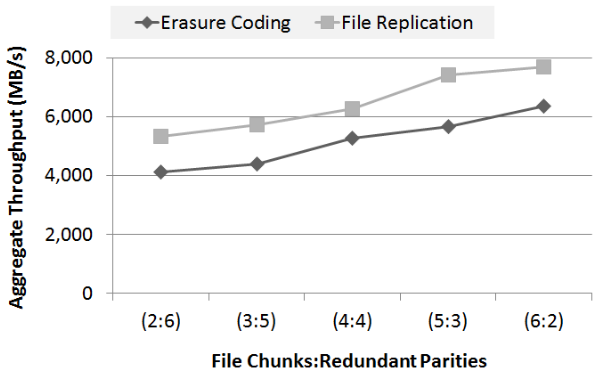


Fig. 11. Performance on Intrepid

Before discussing the performance of GPU-powered Gest at scales, we investigate GPU and CPU coding speed on a single Sirius node. As shown in Fig. 12, GPU typically processes the erasure coding one order of magnitude faster than CPU on a variety of block sizes (except for encoding 16MB block size: 6X faster). Therefore, we expect to significantly reduce the coding time in Gest by GPU acceleration, which consequently improves the overall end-to-end I/O throughput.

We re-run the 8-nodes experiments listed in Table III on the

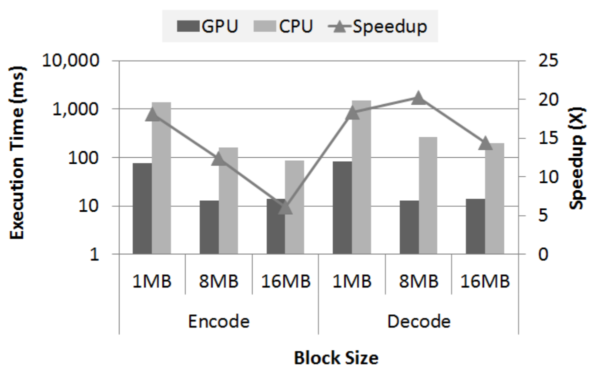


Fig. 12. Gest coding time on a single Sirius node

Sirius cluster. The number of replicas is set to the same number of redundant parities of erasure coding for a fair comparison of reliability, just like what we did in Fig. 11. The results are reported in Fig. 13, where we see for both (5:3) and (3:5) cases, GPU-powered erasure coding delivers higher throughput (read and write combined). Recall that Fig. 10 shows that CPU erasure-coding outperforms file replication in some scenarios; now Fig. 13 indicates that GPU accelerates erasure-coding to outstrip all the replication counterparts.

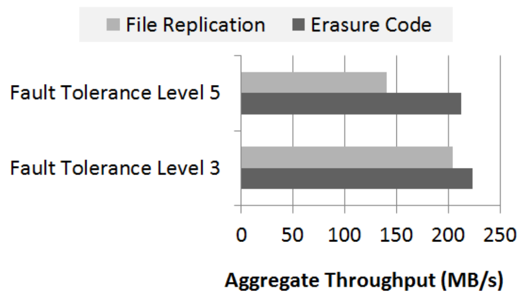


Fig. 13. Performance on the Sirius cluster

## VI. Related Work

Key-value stores have been extensively studied in recent years. Orbe [10] is a distributed key-value store with scalable causal consistency built with newly proposed protocols based on dependency matrices. Muninn [22] is a versioning key-value store that leverages non-volatile memory and an object-based storage model. Gest, to the best of our knowledge, is the first distributed key-value store with GPU acceleration.

Fault tolerance is one of the most challenging part in distributed systems. Much research has been focusing on replacing the full-size replicas. For example, partial fault tolerance was shown to be highly effective in [20]. Another angle (for example, [25]) is to have a non-static number of replicas in the conventional wisdom, so that more replicas could support higher throughput for popular data and fewer replicas are allocated for unpopular data to save the storage cost. In distributed networks, location-aware replication and independent service-structure failure recovery was proposed for group communication services in [46]. In Grid Computing, a transparent fault tolerance architecture was proposed for distributed workflows [12]. In Cloud Computing, data locality and checkpoint placement were extensively studied for fault tolerance in [43]. To achieve graceful degradation of quality of service in case of system overloading, a distance-based priority algorithm was studied in [26]. Recently, a new idea [3] was proposed to expose the internal states and checkpoint them in order to achieve the fault tolerance in stream processing systems. None of the aforementioned systems takes the radical change to break the replica as Gest does.

Recent GPU technology has drawn much research interest of applying these many-cores for data parallelism. For example, GPUs are proposed to parallelize the XML processing [39]. In high performance computing, a GPU-aware MPI was proposed to enable the inter-GPU communication without changing the original MPI interface [45]. Nevertheless, GPUs do not necessarily provide superior performance; GPUs might suffer from factors such as small shared memory and weak single-thread performance as shown in [2]. Another potential drawback of GPUs lies in the dynamic instrumentation that introduces runtime overhead. Yet, recent study [13] shows that the overhead could be alleviated by information flow analysis and symbolic execution. In this paper, we leverage GPUs in key-value stores—a new domain for many-cores.

Besides GPUs, other hardware advances open the door to further performance improvement. For instance, RDMA and InfiniBand are demonstrated to significantly boost the I/O throughput of HBase key-value stores [18]. A hybrid of SSD and HDD was proposed to improve the performance of key-value stores in [31].

## VII. Conclusion and Future Work

This paper presents Gest, a distributed key-value store whose reliability is based on GPU-accelerated coding as opposed to the conventional full-size replication. To the best of our knowledge, Gest is the first distributed key-value store taking advantage of modern GPU parallelism. We provide detailed analysis on how the parameters impact the system. Gest demonstrates that a key-value store's reliability can be achieved as the same level as conventional file replication but with superior space efficiency and I/O performance.

Gest is agnostic about the underlying computing chips, as long as the interfaces are implemented. Therefore, there is nothing architecturally preventing us from leveraging new computing devices to further accelerate the coding process. We also plan on conducting experiments at larger scales, such as the 3,000-GPU Blue Waters supercomputer [1].

## Acknowledgement

This work was supported in part by the National Science Foundation under awards OCI-1054974 (CAREER). This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

## References

[1] Blue Waters. http://ncsa.illinois.edu/enabling/bluewaters, 2014.

[2] R. Bordawekar, U. Bondhugula, and R. Rao. Believe it or not!: Mult-core cpus can match gpu performance for a flop-intensive application! In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, 2010.

[3] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, 2013.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *J. Parallel Distrib. Comput.*, 68(10), 2008.

[5] M. L. Curry, A. Skjellum, H. Lee Ward, and R. Brightwell. Gibraltar: A reed-solomon coding library for storage applications on programmable graphics processors. *Concurr. Comput. : Pract. Exper.*, 23(18), 2011.

[6] B. Debnath, S. Sengupta, and J. Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2), Sept. 2010.

[7] B. Debnath, S. Sengupta, and J. Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[9] Y. Ding, H. Tan, W. Luo, and L. Ni. Exploring the use of diverse replicas for big location tracking data. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, June 2014.

[10] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[11] J. N. England. A system for interactive modeling of physical curved surface objects. pages 336–340, 1978.

[12] O. Ezenwoye, M. Blake, G. Dasgupta, L. Fong, S. Kalayci, and S. Sadjadi. Managing faults for distributed workflows over grids. *Internet Computing, IEEE*, 14(2):84–88, March 2010.

[13] N. Farooqui, K. Schwan, and S. Yalamanchili. Efficient instrumentation of gpgpu applications using information flow analysis and symbolic execution. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, 2014.

[14] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124), Aug. 2004.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[16] Y. Gu and R. L. Grossman. Supporting configurable congestion control in data transport services. In *ACM/IEEE Conference on Supercomputing*, 2005.

[17] J. L. Hafner, V. Deenadhayalan, K. K. Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. 2005.

[18] J. Huang, X. Ouyang, J. Jose, M. Wasi-ur Rahman, H. Wang, M. Luo, H. Subramoni, C. Murthy, and D. Panda. High-performance design of hbase with rdma over infiniband. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, May 2012.

[19] Intrepid. https://www.alcf.anl.gov/user-guides/intrepid-challenger-surveyor, Accessed September 5, 2014.

[20] G. Jacques-Silva, B. Gedik, H. Andrade, K.-L. Wu, and R. K. Iyer. Fault injection-based assessment of partial fault tolerance in stream processing applications. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, DEBS '11, 2011.

[21] H. Jin, X. Yang, X.-H. Sun, and I. Raicu. Adapt: Availability-aware mapreduce data placement for non-dedicated distributed computing. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, 2012.

[22] Y. Kang, T. Marlette, E. L. Miller, and R. Pitchumani. Muninn: a versioning key-value store using object-based storage model. In *Proceedings of the 7th International Systems and Storage Conference (SYSTOR 14)*, June 2014.

[23] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[24] T. Kobus, M. Kokocinski, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, 2013.

[25] K. R. Krish, A. Khasymski, A. R. Butt, S. Tiwari, and M. Bhandarkar. Aptstore: Dynamic storage management for hadoop. In *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01*, CLOUDCOM '13, 2013.

[26] J. Li, Y. Song, and F. Simonot-Lion. Providing real-time applications with graceful degradation of qos and fault tolerance according to (m, k)-firm model. *Industrial Informatics, IEEE Transactions on*, 2(2), 2006.

[27] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, 2013.

[28] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence database scanning on a gpu. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, 2006.

[29] P. Marandi, C. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, June 2014.

[30] A. J. McAuley. Reliable broadband communication using a burst erasure correcting code. pages 297–306, 1990.

[31] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen. Optimizing Key-Value Stores for Hybrid Storage Architectures. In *Proceedings of CASCON*, 2014.

[32] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, Mar. 2008.

[33] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, June 2012.

[34] J. S. Plank. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Technical report, University of Tennessee, 2007.

[35] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proccedings of the 7th Conference on File and Storage Technologies*, 2009.

[36] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society of Industrial and Applied Mathematics*, (2), 06/1960.

[37] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *SIGCOMM Comput. Commun. Rev.*, (2):24–36, Apr.

[38] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, July 2013.

[39] L. Shnaiderman and O. Shmueli. A parallel twig join algorithm for XML processing using a GPGPU. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2012.

[40] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies*, 2010.

[41] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, Aug. 2001.

[42] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[43] X. Su. *Efficient Fault-tolerant Infrastructure for Cloud Computing*. PhD thesis, Yale University, 2013.

[44] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*, pages 531–532. Prentice Hall; 2nd edition, 2006.

[45] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. Panda. Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation. *Parallel and Distributed Systems, IEEE Transactions on*, 25(10), 2014.

[46] Y.-H. Wang, Z. Zhou, L. Liu, and W. Wu. Fault tolerance and recovery for group communication services in distributed networks. *Journal of Computer Science and Technology*, 27(2):298–312, 2012.

[47] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 328–338, 2002.

[48] H. Xia and A. Chien. RobuSTore: a distributed storage architecture with robust and high performance. In *ACM/IEEE Conference on Supercomputing*, 2007.