Toward High-performance Key-value Stores through GPU Encoding and Locality-aware Encoding

Dongfang Zhao*, Ke Wang†, Kan Qiao‡, Tonglin Li§, Iman Sadooghi§, Ioan Raicu§
*Pacific Northwest National Laboratory †Intel ‡Google §Illinois Institute of Technology

Corresponding author: Dongfang Zhao, Phone: (509)375-3635, Email: dongfang.zhao@pnnl.gov

***Abstract*—Although distributed key-value store is becoming increasingly popular in compensating the conventional distributed file systems, it is often criticized due to its costly full-size replication for high availability that causes high I/O overhead. This paper presents two techniques to mitigate such I/O overhead and improve key-value store performance: GPU encoding and locality-aware encoding. Instead of migrating full-size replicas over the network, we split the original file into smaller chunks and encode them with a few additional parity codes using GPUs before dispersing them onto remote nodes. The parity code is usually much smaller than the original file, which saves the extra space required for high availability and reduces the I/O overhead. Meanwhile, the compute-intensive encoding process is largely accelerated by the massive number of GPU cores. Yet, splitting the original file into smaller chunks stored on multiple nodes breaks data locality from application's perspective. To this end, we present a locality-aware encoding mechanism that allows a job to be dispatched as finer-grained tasks right on the node where the required chunk resides. Therefore, the data locality is preserved at the finer granularity of sub-job (i.e., task) level. We conduct an in-depth analysis of the proposed approach and implement a system prototype named Gest. Gest has been deployed and evaluated on a variety of testbeds demonstrating that high data availability, high space efficiency, and high I/O performance could be collectively achieved at the same time.**

I. Introduction

Nowadays, distributed key-value store (KVS) is becoming one of the most important storage paradigms that compensate the conventional file systems on large-scale clusters. For instance, cloud vendors have widely adopted distributed KVS such as Cassandra [22], Dynamo [8], and Memcached [13]. High-performance computing follows a similar trend by leveraging distributed KVS for either performance improvement or unprecedented services, some of which were demonstrated in our prior work [24, 25, 32, 38, 44, 45]. The state-of-the-art approach to achieve high reliability and availability for distributed KVS is replication, which however, by nature has the following drawbacks that have side effects on storage and I/O performance: (1) significant space overhead, (2) additional disk I/O, and (3) more network bandwidth consumption.

This work is orthogonal to previous studies that are focused on algorithms, protocols, and models for better manipulating replicas [2, 9, 20, 27, 33]: we propose to replace full-size replication by more space-efficient parity coding along with GPU acceleration. It is, to the best of our knowledge, the first study that explores the feasibility of taking advantages of both parity coding and GPU acceleration in distributed KVS. While the computational overhead during the coding process can be largely compensated by GPUs, one key challenge is how to preserve the data locality of the scattered value (or, file) because the original value is dispersed to different nodes—the file-level locality from application's perspective is completely lost. To this end, we propose a locality-aware encoding mechanism to directly assign the sub-job (i.e., task) to the node where the requested chunk resides.

We justify the effectiveness of the proposed approach from three perspectives.

First, we abstract and model the proposed approach with environmental and system parameters, and derive the analytical results of the expected space utilization and performance speedup when parity coding is accelerated by GPUs. Our analysis shows that (1) the space efficiency of parity coding is roughly m higher than the conventional replication and (2) the end-to-end I/O throughput could be improved by mk times, where m denotes the number of tolerable failures and k denotes the number of chunks per file, respectively.

Then, we design and implement a distributed KVS prototype named Gest: a GPU-accelerated encoded store. Gest has been deployed and evaluated on a variety of platforms, ranging from a commodity Linux cluster to a leadership-class supercomputer. Experimental results confirm that GPU-boosted coding is a promising approach to achieve data availability, space efficiency, and I/O performance collectively at the same time.

Finally, we evaluate the performance benefit from the locality-aware encoding of tasks, which are split from the application-level job and dispatched onto the chunks dispersed to different physical nodes. We replay two popular MapReduce [7] workloads (i.e., sort and grep) when enabling locality-aware encoding with our distributed scheduler named MATRIX [39]. Results show that splitting jobs into smaller tasks only incurs 5% overhead on 128 nodes and achieves almost linear scalability when more nodes are available.

In summary, this paper makes the following contributions.

- We propose an approach to achieve high data availability, low space overhead, and high I/O performance of distributed key-value stores. The approach takes advantages of modern many-core architecture of GPUs to parallelize and accelerate the parity coding of large data objects. Together with a locality-aware distributed scheduler, the proposed approach is able to significantly reduce the I/O overhead.

- We conduct an in-depth analysis (Section IV) of the proposed mechanism with a large space of parameters. Theoretical analysis demonstrates that the model abstracted from the proposed approach would achieve both high space efficiency and high end-to-end I/O throughput.
- We design and implement a GPU-encoded distributed KVS prototype, namely Gest (Section III). Extensive evaluations (Section V) of Gest with both micro-benchmarks and real-world applications on a variety of testbeds demonstrate Gest's effectiveness in practice.

II. Motivation and Background

A. A Motivating Example

To make matters more concrete, Fig. 1 shows an example of a distributed KVS 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 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 4 times faster because each chunk is roughly a quarter of the original value `V1` in size.

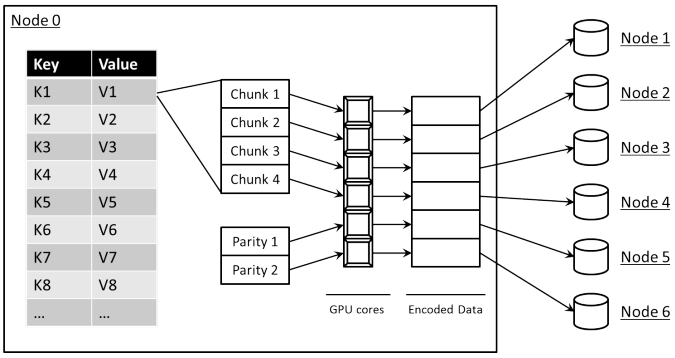

Fig. 1. A distributed key-value store, which can tolerate up to two failures, is deployed on six nodes with GPU acceleration.

B. Distributed Key-Value Stores

A distributed key-value storage usually has a simpler interface than a POSIX filesystem. Most file operations are implemented as `set(key, value)` and `value` ← `get(key)` in KVS, rather than conventional POSIX API such as `fh` ← `fopen(fname)`, `fwrite(ptr, sz, cnt, fh)`, `fread(ptr, sz, cnt, fh)`, and `fclose(fh)`. It greatly simplifies the code complexity and allows developers to focus more on the business logic rather than the I/O syntax.

Most distributed key-value store systems use replication to achieve reliability. In spite of its simplicity, data replication maintains several full-size 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 reliability in distributed KVS.

ZHT [25] has a similar ring-shaped look as the traditional DHT [35]. 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 on the client side maps a string to an ID that can be retrieved by a `lookup(k)` operation at a later point. Besides common KVS 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 writes. The replicas in ZHT should not be confused with the file replicas in Gest; ZHT replicas are used for the fault tolerance of metadata, and have no direct impact on Gest's reliability.

C. Erasure Coding

Erasure coding has been studied by the computer communication community since the 1990's [26, 31], as well as in storage and filesystems [16, 19, 29, 42]. Plank et al. [29] conduct 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 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 [41] show that for total N machines out of which M are unavailable, 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 the encoding process. At first glance, the scheme looks similar to file replication, as it allocates additional disks as backups. Nevertheless, the underlying rationale of erasure coding is completely different from file replication because of its complex matrix computation. As a case in point, one popular erasure code is Reed-Solomon coding [30], 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.

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 use either 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 on 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.

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

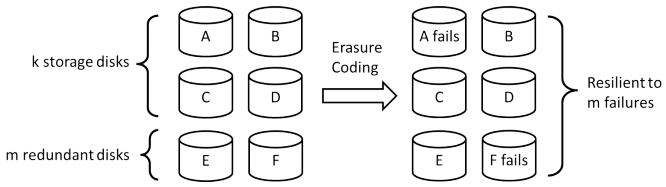

Fig. 2. Encoding k chunks into $n = k + m$ chunks so that the system is resilient to m failures

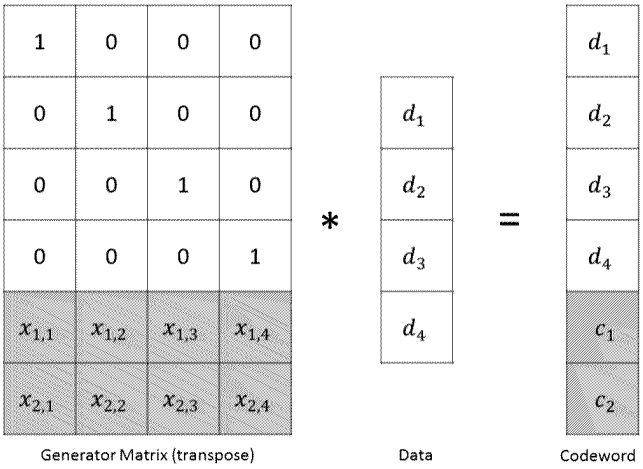

Fig. 3. Encoding 4 files into 6 codewords with Reed-Solomon coding

D. 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.

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 $\frac{384}{8} = 48$ times. 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	GeForce GT640	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.

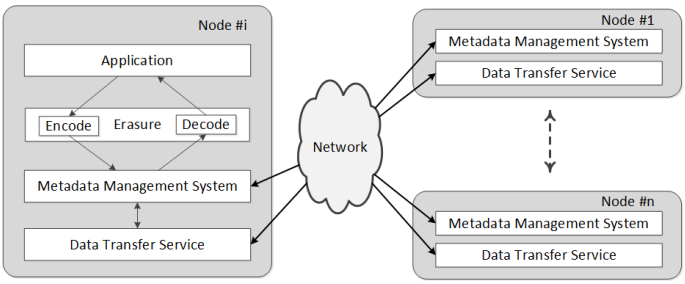

Fig. 4. Architectural overview of Gest deployed on an n-nodes distributed system. End users run applications on the i^{th} node where files are encoded and decoded by coding algorithms.

To make matters more concrete, Fig. 5 illustrates the scenario of writing and reading a file with $k = 4$ 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 that 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.

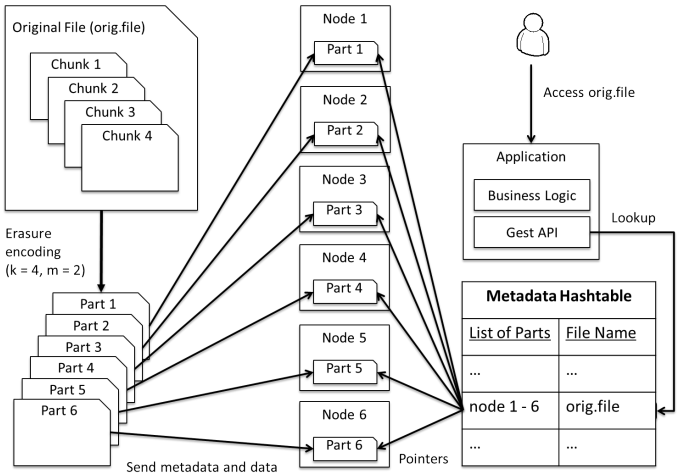

Fig. 5. An example of file writing and reading on Gest

A. Metadata Management

The traditional way of handling metadata for distributed systems is to manipulate them on one or a few nodes. The rationale is that metadata contains only high level information, thus is small in size. Therefore, a centralized repository usually meets the requirement. Most distributed storage systems

in production employ centralized metadata management; for instance the Google file system (GFS [14]) keeps all its metadata on the master node. This design is easy to implement and maintain, yet exposes a performance bottleneck for the workloads that generate a large amount of small files: the metadata accessing rate and memory footprint of a great number of small files can easily saturate the limited number of metadata servers. As a GFS tech lead mentioned, "a single master soon became a bottleneck of GFS ... Therefore, new and scalable schemes are needed for metadata management ..." [43].

In contrast, we manage Gest's metadata in a completely distributed fashion. Specifically, all metadata are dispersed into a distributed hashtable (ZHT [25]). While there are many choices of distributed hashtable implementations such as Memcached [13] and Dynamo [8], we chose ZHT as the underlying infrastructure because it is implemented in C/C++, takes the lowest (constant, in most cases) routing time, and supports both persistence (through periodical flushing) and dynamic membership. ZHT is installed as daemon services on Gest nodes.

In Gest, clients have a coherent view of all the files (i.e., metadata) no matter if the file is stored locally or not. 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 location.

Because all clients of Gest share the same global namespace of the underlying files, from the API point of view there is nothing particularly different between these clients. In fact, Gest API is built upon ZHT's own client API, which manages all the mappings between file chunks and their locations. It is ZHT's client, deployed on each every node in the cluster, who retrieves the metadata information of all files and chunks.

Besides the conventional metadata information for regular files, a special flag indicates whether 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 moves data back and forth across the network, which should be efficient and reliable in the ideal case. The User Datagram Protocol (UDP) is efficient in transferring data, but is unreliable, while the Transmission Control Protocol (TCP) is reliable but inefficient.

We have developed our own data transfer service – Gest Data Transfer service (GDT) on top of UDP-based Data Transfer (UDT [15]), 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.

Technically, GDT serves in a similar way to TCP where to specify the destination and port number when transferring files. Nonetheless, it exposes a simpler interface customized for Gest. It should be noted that, however, users are agnostic of the network protocols in Gest (GDT in this case); they only need to know how to use the Gest client API as shown in Figure 5.

It should be clear that moving data is, usually, more costly than dispatching the job to the node where the data resides. In other words, a "locality-aware" encoding would be more desirable than simply migrating file chunks into a remote (random) node. To this end, we could leverage a locality-aware job scheduler (e.g., MATRIX [39]) to strive to execute the encoding process on the node holding the file chunk.

C. Coding Algorithms

In addition to daemon services running at the back end, Gest plugs in encoding and decoding modules on the fly. Currently, we support two built-in libraries, namely Jerasure [28] and Gibraltar [6], as the default CPU and GPU libraries, respectively. Gest is implemented to be flexible to adopt coding libraries.

D. Workflows

When an application writes a file, Gest splits the file into k chunks. Depending on which coding library the user chooses, these k chunks are encoded into $n = k + m$ chunks, which are sent to n different nodes with 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 with an in-memory hashmap whose contents are asynchronously persisted to the local disk.

Reading a file is the reverse of writing it. 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.

F. User Interface

Gest provides a completely customizable set of parameters for the applications to tune the behavior of Gest. In particular, users can specify the 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.

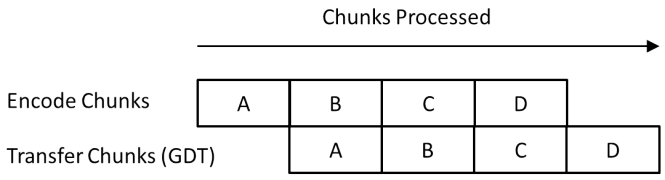

Fig. 6. Pipelining of encoding and transferring for a write operation in Gest

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 speed 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 [6] 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.

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:

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

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

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 in the coding process):

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

Again, since m is usually a small integer, and because in a large-scale system n (the total number of nodes) is usually significantly larger than m, the value of E_{gest} is highly close to 1. Also recall that the space efficiency of full-size replication is $\frac{1}{m}$. **Consequently, the space efficiency of parity coding is roughly m higher than the conventional replication where m indicates the number of tolerable failures.**

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

$$E^p_{gest} = \frac{s}{s + s \cdot \frac{k+m-1}{k}} = \frac{k}{2k+m-1}$$

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

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

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

$$Speedup \approx m \cdot k$$

In other words, **the end-to-end I/O throughput could be improved by mk times when full-size replications is replaced by concurrent parity coding, where m indicates the tolerable failures and k indicates the number of chunks per file.** 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. Testbeds

Gest has been deployed on four testbeds: a Sun-Oracle cluster (HEC), a high-performance GPU cluster (Sirius), the Amazon EC2 cloud, and an IBM Blue Gene/P supercomputer (Intrepid [1]). Each HEC node has 8 GB RAM, dual AMD Opteron quad-core processors (8-cores at 2 GHz), and an OS whose 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. Experiments on the Amazon EC2 Cloud are deployed on the "m3.large" instances up to 256 CPUs (i.e., 128 instances). Each instance has 2 CPUs, 6.5 ECUs, 7.5 GB memory, 16 GB SSD, and the Ubuntu 14.04 LTS OS. 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. In this paper we used up to 512 nodes for evaluating Gest.

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 with erasure coding algorithms of different parameter combinations at different scales. Table III summarizes the list of candidate mechanisms, as well as the the number of chunks for both the original file and the redundant data.

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.

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

Similarly, different erasure-coding parameters of Gest are listed as `Erasure[1..6]`, along with different file granularity and additional parities. We design experiments at different scales to study the scalability of Gest. 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>` means 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 picked 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 KVS without compromising the reliability. Therefore in all cases of erasure coding, 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 of 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.

First, a larger scale enables higher space efficiency. This is counter-intuitive to some extent, as it is a well-accepted practice to collect data on a small subset of nodes (e.g., collective I/O can batch small and dispersed I/Os to reduce the number of I/O calls). Fig. 7, however, demonstrates that when redundant parity stays the same, space efficiency is monotonically increasing with respect to more nodes. The reason is that with more nodes the redundant parity is in finer

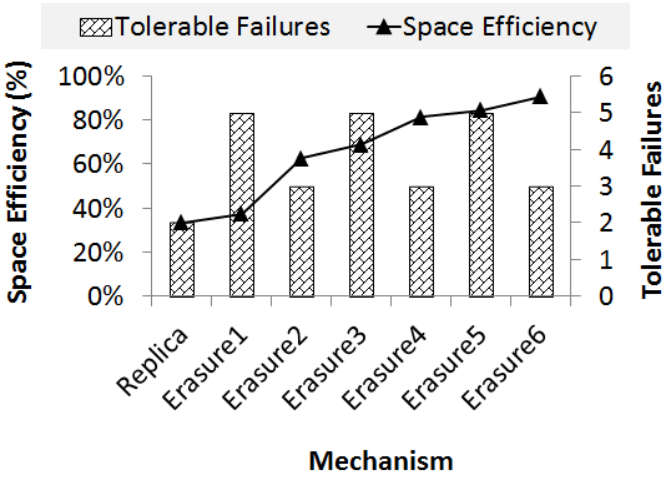

Fig. 7. Data reliability and space efficiency

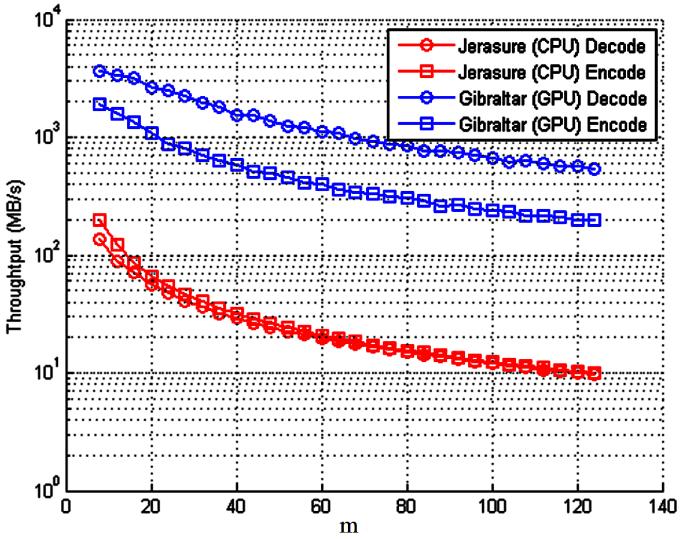

Fig. 8. Throughput with buffer size = 1 MB, storage efficiency = 33%

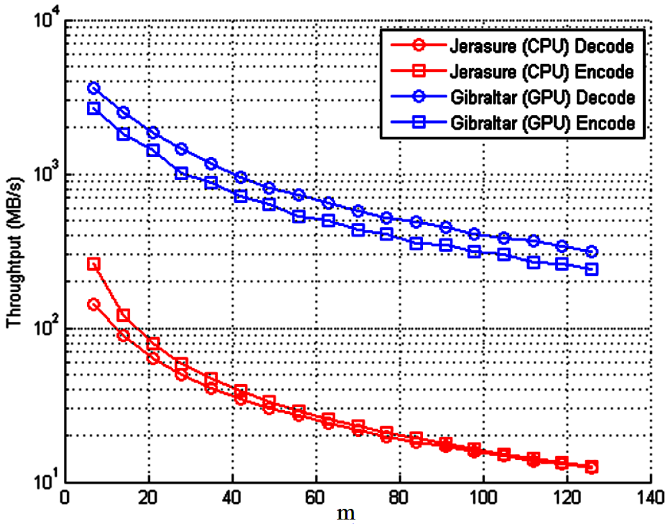

Fig. 9. Throughput with buffer size = 1 MB, storage efficiency = 75%

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 of our testbeds. 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 a fixed storage efficiency as 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.

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

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

E. I/O Performance

We compare both read and write throughput of all mechanisms listed in Table III on the HEC cluster at the scales of 8-nodes, 16-nodes, and 32-nodes. The files to be read and written are 1 GB per node, with 1 MB block size. Fig. 10 shows the results. One important observation 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 later 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, and to evaluate Gest at larger

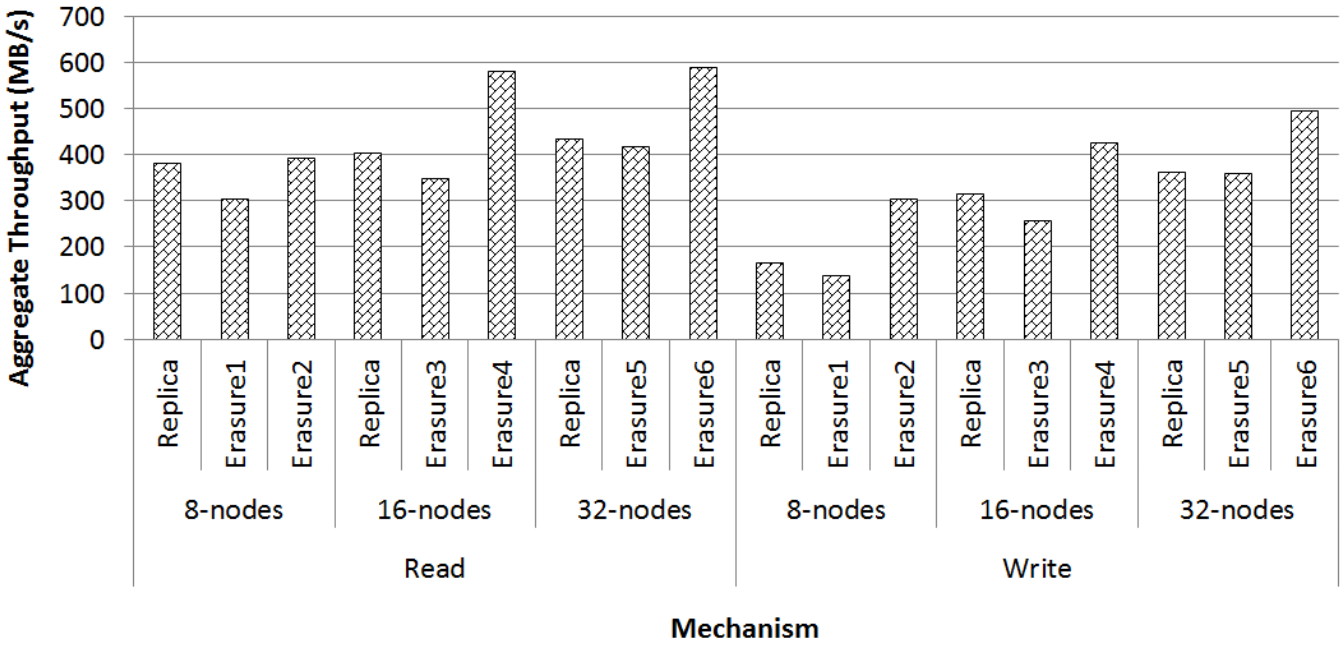

Fig. 10. Performance on the HEC cluster

scales, we deploy Gest on 512 nodes of Intrepid, where each node only has 4 cores at 850 MHz. For a fair comparison, we slightly change the setup of the previous 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 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?

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 16 MB block size: 6 times faster). Therefore, we expect to significantly reduce the

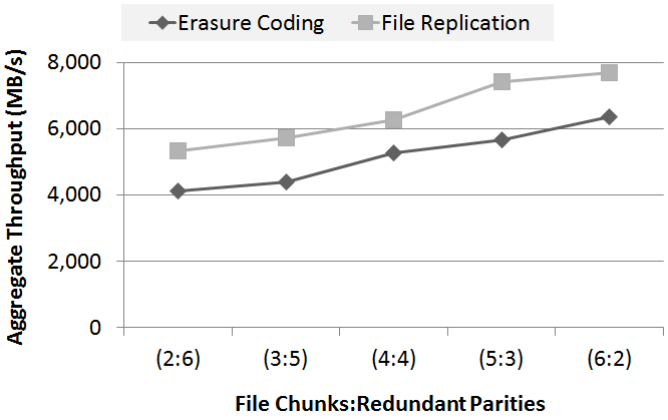

Fig. 11. Performance on Intrepid [1]

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 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.

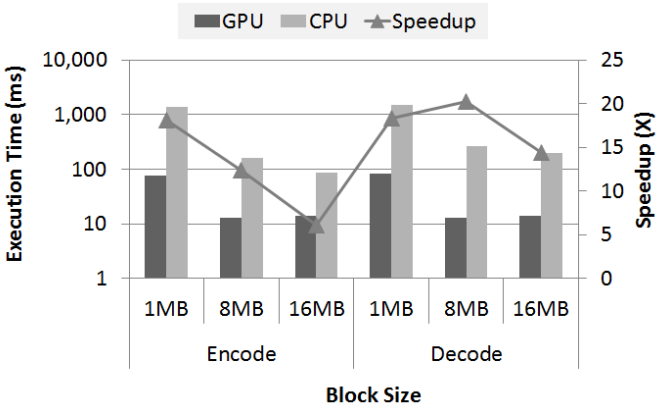

Fig. 12. Gest coding time on a single Sirius node

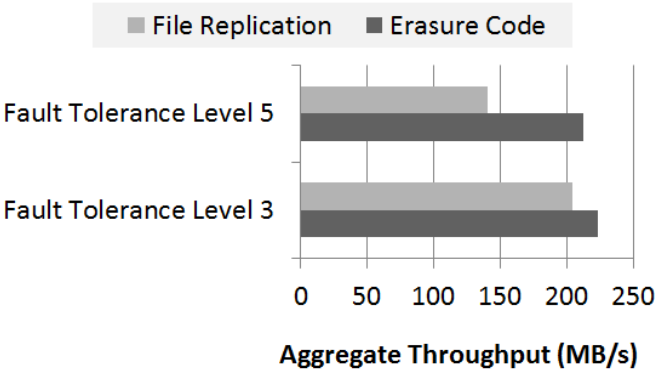

Fig. 13. Performance on the Sirius cluster

F. Locality-aware Encoding

We evaluate the performance benefit from locality-aware encoding by executing MapReduce workloads. The input data is 10 GB extracted from Wikipedia. We do weak-scaling experiments that process 256 MB data per instance. That is, at 128 instances the total data size is 32 GB (i.e., 3.2 copies of the 10 GB input data). The first application is "grep", which searches texts to match the given pattern in the file. The second application is "sort", which performs in-place sort of all the words of a given file. We set $k = 4$ and $m = 2$ in this experiment; that is, each file is split into 4 equal chunks and encoded with 2 additional parities with GPUs.

Fig. 14 shows, at different scales from 1 to 128 instances, the speedup and efficiency when 4 tasks are derived from a single job and then work on 4 chunks concurrently. The speedup is measured by considering the scalability, the wall time of the application when chunking files is disabled, and the overhead introduced by the finer granularity of the tasks (sub-jobs). The efficiency is defined as the ratio of the real speedup over the scalability, i.e., the number instances in this case.

We observe that the speedup is slightly decreased from 1 to 128 instances. The reason is that more instances incur higher overhead of spawning a larger number of small tasks.

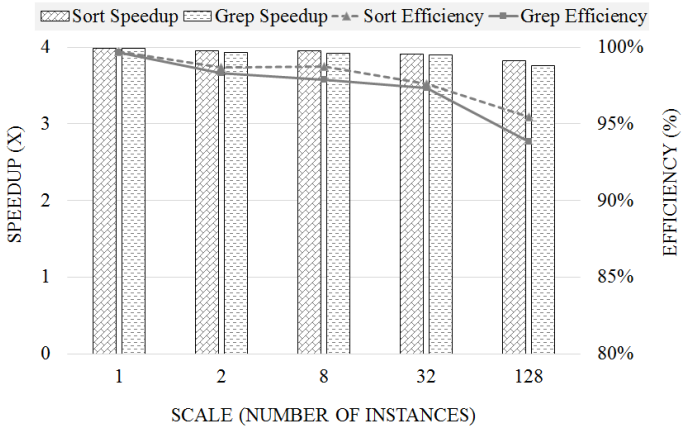

Fig. 14. Speedup and efficiency of two applications (sort and grep) when data-locality aware encoding is enabled

Nevertheless, this overhead is significantly smaller than the I/O gain from the data parallelism. As the efficiency plot shows, the speedup keeps around 95% at 128 instances—we only lose 5% efficiency after scaling up more than two orders of magnitude.

G. Comparison of Popular Key-value Stores

This section focused on the performance comparison between Gest and other popular key-value stores. In particular, we are most interested in two representative systems—Memcached [13] and Cassandra [22]. In this experiment, we configure all three systems to tolerate up to three failures. That is, for Gest it adopts the (5:3) erasure coding scheme while the other systems maintain 3 replicas.

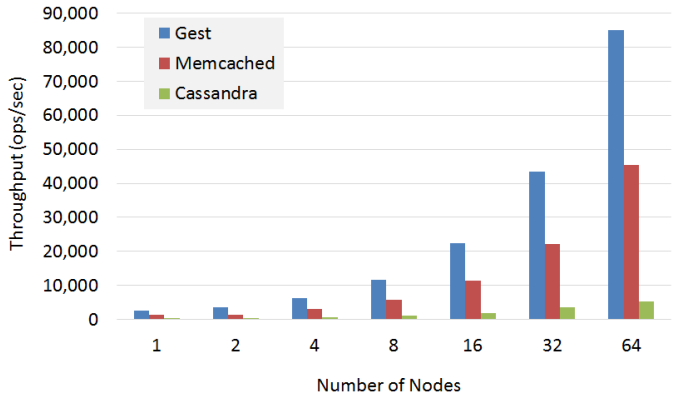

Fig. 15. Comparison of popular key-value stores

Figure 15 reports the throughput of all key-value stores at different scales from 1 node to 64 nodes. The results evidently shows the performance advantage of Gest over the others at all scales. Two more observations should be noted. First, Gest shows a strong scalability, which is highly desirable and essential to many applications. Second, although Memcached is a memory-based system, its performance could hardly

compete with Gest, which is largely due to Gest's highly efficient replication scheme and GPU's acceleration.

H. Case Study: a Building Block for Distributed File Systems

In order to demonstrate the effectiveness of Gest in real-world applications, we evaluate its performance impact to the FusionFS [46] distributed file system. The original design of FusionFS leverages a distributed KVS called ZHT [25] to manage its metadata. We replace ZHT by Gest and we expect a higher performance delivered by FusionFS.

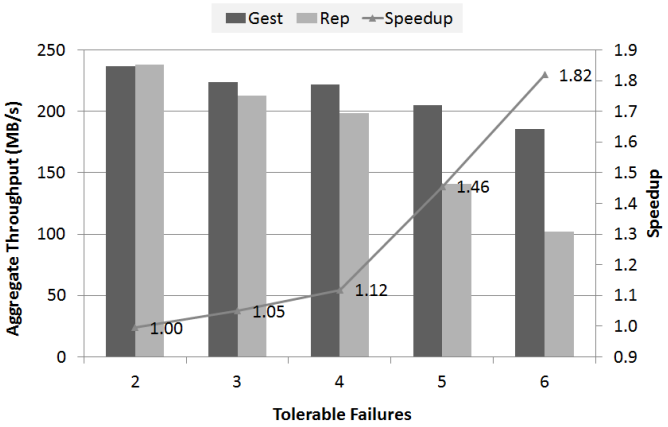

Fig. 16. Throughput of Gest and Rep on FusionFS (block size 1 MB)

In this experiment, FusionFS is deployed on the Sirius cluster. We keep the total number of nodes fixed (i.e., $n = k + m = 8$). The tolerated number of failures (i.e., m) ranges from 2 to 6. It is equivalent to having 3 to 7 replicas (including the primary copy) in the conventional data replication algorithms (Rep). Figure 16 shows the end-to-end I/O throughput of Gest- and Rep-based data redundancy in FusionFS. Only when $m = 2$, Rep slightly outperforms Gest, and the difference is almost negligible. Starting from $m = 3$, Gest clearly shows its advantage over Rep, and the speedup increases for larger ms. Particularly, when $m = 6$, i.e., to keep the system's integrity allowing 6 failed nodes, Gest throughput is 1.82 higher than the traditional Rep method.

VI. Related Work

Distributed key-value stores have been extensively studied in recent years. In addition the most popular ones such as Memcached [13] and Cassandra [22], there are a couple more representative systems. Orbe [10] is a distributed key-value store with scalable causal consistency built with newly proposed protocols based on dependency matrices. Muninn [18] 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 [17]. Another angle (for example, [21]) 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 [40]. In Grid Computing, a transparent fault tolerance architecture was proposed for distributed workflows [11]. In Cloud Computing, data locality and checkpoint placement were extensively studied for fault tolerance in [36]. To achieve graceful degradation of quality of service in case of system overloading, a distance-based priority algorithm was studied in [23]. Recently, a new idea [5] 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 [34]. In high performance computing, a GPU-aware MPI was proposed to enable the inter-GPU communication without changing the original MPI interface [37]. 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 [4]. Another potential drawback of GPUs lies in the dynamic instrumentation that introduces runtime overhead. Yet, recent study [12] 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.

VII. Conclusion

This paper presents Gest, the first distributed key-value store whose reliability is achieved through GPU-accelerated parity coding as opposed to the conventional full-size replication. Its data locality is uniquely achieved by dispatching tasks right on the chunk node rather than the conventional merge-decode paradigm. In addition to the theoretical analysis to justify the effectiveness of Gest, we also, from a system's perspective, showcase how to design and implement a system to solve the long-existing dilemma between data reliability, space efficiency, and I/O performance at the same time. In a more general sense, Gest demonstrates that a data-intensive problem can be transformed into a compute-intensive one (full-size replication vs. parity coding), which is then solved by compute-intensive devices (GPUs) and a more distributed architecture (i.e., locality-aware distributed scheduler).

Our future work on Gest is to deploy and tune it on more heterogeneous testbeds. Since Gest is agnostic about the underlying computing chips as long as the interfaces are implemented, there is nothing architecturally preventing us from leveraging new computing devices to further accelerate the coding process. Another direction is to evaluate it on extreme scales. For instance, we plan on conducting experiments on the

3,000-GPU Blue Waters supercomputer [3] at National Center for Supercomputing Applications.

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] Argonne's Intrepid, "https://www.alcf.anl.gov/user-guides/intrepid-challenger-surveyor," Accessed September 5, 2014.

[2] A. Benoit, H. Larcheveque, and P. Renaud-Goud, "Optimal algorithms and approximation algorithms for replica placement with distance constraints in tree networks," in *IEEE 26th International Symposium on Parallel Distributed Processing (IPDPS)*, 2012.

[3] Blue Waters, "http://ncsa.illinois.edu/enabling/bluewaters," Accessed April 17, 2015.

[4] 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*, ser. PACT '10, 2010.

[5] 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*, ser. SIGMOD '13, 2013.

[6] 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.*, vol. 23, no. 18, 2011.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of USENIX Symposium on Opearting Systems Design & Implementation*, 2004.

[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.*, vol. 41, no. 6, 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*, ser. SOCC '13, 2013.

[11] O. Ezenwoye, M. B. Blake, G. Dasgupta, L. L. Fong, S. Kalayci, and S. M. Sadjadi, "Managing faults for distributed workflows over grids," *Internet Computing, IEEE*, vol. 14, no. 2, pp. 84–88, March 2010.

[12] 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*, ser. GPGPU-7, 2014.

[13] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, Aug. 2004.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *ACM Symposium on Operating Systems Principles*, 2003.

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

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

[17] 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*, ser. DEBS '11, 2011.

[18] 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)*, Jun. 2014.

[19] 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.

[20] 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*, ser. ICDCS '13, 2013.

[21] R. K. 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*, ser. CLOUDCOM '13, 2013.

[22] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, Apr. 2010.

[23] 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*, vol. 2, no. 2, 2006.

[24] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu, "Exploring distributed hash tables in highend computing," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 3, Dec. 2011.

[25] 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.

[26] A. J. McAuley, "Reliable broadband communication using a burst erasure correcting code," in *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, 1990, pp. 297–306.

[27] S. Mu, K. Chen, Y. Wu, and W. Zheng, "When paxos meets erasure code: Reduce network and storage cost in state machine replication," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, 2014.

[28] J. S. Plank, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications," University of Tennessee, Tech. Rep., 2007.

[29] 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.

[30] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society of Industrial and Applied Mathematics*, no. 2, 06/1960.

[31] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *SIGCOMM Comput. Commun. Rev.*, no. 2, pp. 24–36, Apr.

[32] I. Sadooghi, J. H. Martin, T. Li, K. Brandstatter, K. Maheshwari, T. P. P. de Lacerda Ruivo, G. Garzoglio, S. Timm, Y. Zhao, and I. Raicu, "Understanding the performance and potential of cloud computing for scientific applications," *IEEE Transaction on Cloud Computing (TCC)*, 2015.

[33] 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*, 2013.

[34] 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.

[35] 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.*, vol. 31, no. 4, pp. 149–160, Aug. 2001.

[36] X. Su, "Efficient fault-tolerant infrastructure for cloud computing," Ph.D. dissertation, Yale University, 2013.

[37] 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*, vol. 25, no. 10, 2014.

[38] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Using simulation to explore distributed key-value stores for extreme-scale system services," in *Proceedings of ACM/IEEE International Conference on Supercomputing*, 2013.

[39] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proceedings of IEEE International Conference on Big Data (BigData Conference)*, 2014.

[40] 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*, vol. 27, no. 2, pp. 298–312, 2012.

[41] 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*, 2002, pp. 328–338.

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

[43] X. Yang, *Principles, Methodologies, and Service-Oriented Approaches for Cloud Computing*, 1st ed. Hershey, PA, USA: IGI Global, 2013.

[44] D. Zhao, K. Qiao, and I. Raicu, "Hycache+: Towards scalable high-performance caching middleware for parallel file systems," in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 267–276.

[45] D. Zhao, C. Shou, T. Malik, and I. Raicu, "Distributed data provenance for large-scale data-intensive computing," in *Cluster Computing, IEEE International Conference on*, 2013.

[46] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "FusionFS: Toward supporting data-intensive scientific applications on extreme-scale distributed systems," in *Proceedings of IEEE International Conference on Big Data*, 2014, pp. 61–70.