Enabling Space Elasticity in Storage Systems

Helgi Sigurbjarnarson
University of Washington
helgi@cs.washington.edu

Petur O. Ragnarsson
Reykjavik University
peturr13@ru.is

Juncheng Yang
Emory University
juncheng.yang@emory.edu

Ymir Vigfusson
Emory University / Reykjavik University
ymir@ymsir.com

Mahesh Balakrishnan
Yale University
mahesh@cs.yale.edu

Abstract

Storage systems are designed to never lose data. However, modern applications increasingly use local storage to improve performance by storing soft state such as cached, prefetched or precomputed results. Required is elastic storage, where cloud providers can alter the storage footprint of applications by removing and regenerating soft state based on resource availability and access patterns. We propose a new abstraction called a motif that enables storage elasticity by allowing applications to describe how soft state can be regenerated. Carillon is a system that uses motifs to dynamically change the storage space used by applications. Carillon is implemented as a runtime and a collection of shim layers that interpose between applications and specific storage APIs; we describe shims for a filesystem (Carillon-FS) and a key-value store (Carillon-KV). We show that Carillon-FS allows us to dynamically alter the storage footprint of a VM, while Carillon-KV enables a graph database that accelerates performance based on available storage space.

Categories and Subject Descriptors D4.2 [***Operating Systems***]: Storage Management

1. Introduction

The promise of cloud computing lies in elasticity: the property that applications can ramp up or dial down resource usage as required, eliminating the need to accurately estimate service load and resource cost a priori. Elasticity can usually be achieved easily for CPU or RAM, either by spinning up or down more virtual machines (i.e., horizontal scaling), or by adding cores or RAM to individual virtual machines (i.e., vertical scaling) [21]. However, storage is typically *inelastic* in the dimension of space or capacity; the cloud provider cannot easily modulate the storage footprint of an application.

The need – and opportunity – for space elasticity in storage arises from the fact that modern applications often store data on durable media for performance rather than durability. In the 00s, system designers were faced with slow, stagnant network speeds that stayed within 1 Gbps for over a decade; architectural paradigms such as peer-to-peer and client-server computing that forced applications to operate over bandwidth-constrained WANs; and hierarchical topologies within data centers that created bandwidth bottlenecks. In addition, CPU cycles were relatively scarce in the single-core era. In contrast, disks were large, inexpensive, and fast for sequential I/O at 100 MB/s or more. Such constraints pushed designers to creatively use excess storage capacity in order to avoid network I/O or CPU usage.

As a result, much of the data stored by applications on secondary storage is volatile data that does not fit in RAM; usually, it can be thrown away on a reboot (e.g., swap files), reconstructed via computation over other data (e.g., intermediate MapReduce or Dryad files [8], image thumbnails, memoized results of computations, desktop search indices, and inflated versions of compressed files), or fetched over the network from other systems (e.g., browser and package management caches). As a case in point, up to 55% of storage on three of our own development VMs is consumed by caches and ephemeral content. In addition, durability may not be critical for a file either because new applications (such as big data analytics) can provide useful answers despite missing data [1], or because the data may be duplicated across multiple files [5, 13].

Such behavior by applications is problematic because the balance between network, compute and storage has shifted in recent years: applications now execute in the cloud within

SYSTOR '16, June 06-08, 2016, Haifa, Israel.

DOI: http://dx.doi.org/10.1145/2928275.2928291

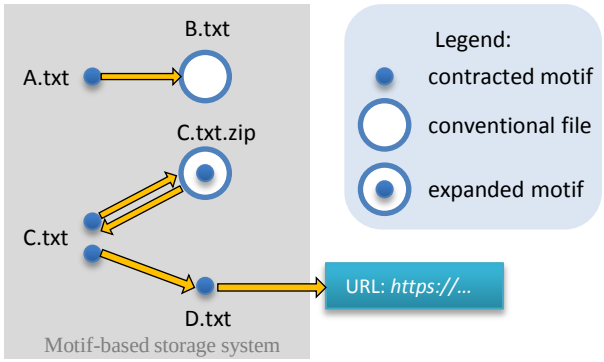


Figure 1: ***Motifs*** *exist in expanded or contracted form; depend on other files, motifs, and external resources; and can have circular dependencies.*

data centers that provide full-bisection bandwidth; networks are faster in general, with the emergence of 10 Gbps Ethernet and the resurgence of RDMA; compute cycles are plentiful and often wasted on multi-core machines; and hard disks are increasingly supplanted by fast, small and expensive SSDs as default secondary storage media. Cloud providers now face the need to multiplex scarce storage resources among multiple tenants, but their job is made difficult by applications that treat storage as a cheaper resource than network or compute. Required are interfaces and mechanisms that allow cloud providers to control the storage behavior of applications, reclaiming space occupied by ephemeral or reconstructable data as needed.

Unfortunately, existing storage systems treat durability as a sacred covenant: all data is equally important, and no data must be lost. The assumption is that applications will only store data on a durable medium if they actually require durability, and the task of the storage system is to preserve that durability at all costs. As a result, modern storage systems exhibit an inefficient dynamic: applications opportunistically use persistent storage to store data that is ephemeral, whereas storage systems struggle heroically to ensure that this data is not lost.

In this paper, we present the *motif* abstraction: a code fragment attached by the application to a unit of data (i.e. a file, a key-value pair, etc.) that tells the storage system if and how that data can be reconstructed. The motif can be *expanded* to generate the bytes constituting the data item, or *contracted* back. For example, a motif might generate the file by fetching data over the network from a URL, or via computation over other files (e.g., sort a file, merge multiple input files, generate an index from a larger file, or even expand a compressed input file).

We implement the motif abstraction within a system called Carillon, which consists of two components. The first is a runtime that manages motifs, deciding when to expand and contract them based on resource availability and access patterns. The second is a thin shim layer that interposes between the application and an unmodified storage system (e.g. a filesystem or a key-value store). The API exposed by the shim layer to the application can be identical to that of the underlying storage system (e.g., a POSIX filesystem API), with the addition of an interface to allow applications to install or remove motifs. The shim layer and the runtime interact with each other via an IPC mechanism. This two-part design enables developers to easily add motif support to any target storage system simply by implementing a new shim layer.

We demonstrate the end-to-end utility of motifs via two Carillon shims and their corresponding real-world applications. We implement a filesystem shim (Carillon-FS) that exposes a POSIX API and runs over ext4. We execute Carillon-FS as a guest filesystem within multiple virtual machines, interacting with corresponding Carillon runtime instances running on the host OS. With the help of file-based motifs, Carillon-FS allows the storage footprint of each VM to change over time. The performance overhead of our prototype is less than 5% beyond standard FUSE overheads on the `filebench` benchmark.

We also implement a key-value store shim (Carillon-KV) that runs over – and exposes the API of – LevelDB [11]. Above Carillon-KV, we implement a graph database that stores its state in the form of adjacency lists in the key-value store; for each node in the graph, there is a key-value pair where the key is the node ID and the value is a list of neighboring node IDs. The graph database proactively calculates answers to popular queries (e.g., the path between two nodes) and stores them in Carillon-KV, while providing a motif describing how these cached results were computed. When evaluated on a simple shortest-paths application on top of graph database, Carillon-KV was able to reduce latencies $10\times$ through the use of motifs.

We make the following contributions in this paper:

- We propose the motif abstraction as a way for storage systems to achieve space elasticity, by understanding how the data they store can be reconstructed via computation, network accesses, and other data.
- We describe the design and implementation of a system called Carillon that implements the motif abstraction. Carillon can be extended with thin shims to add space elasticity to any existing storage system.
- We describe two Carillon shims – a POSIX filesystem and a LevelDB-based key-value store – and show that they enable space elasticity in real-world applications such as VM hosting and a graph database, respectively.

2. The Motif Abstraction

A motif is a code fragment capable of regenerating a data item (such as a file or a key-value pair). It exposes a single *expand* method which generates the content of that item

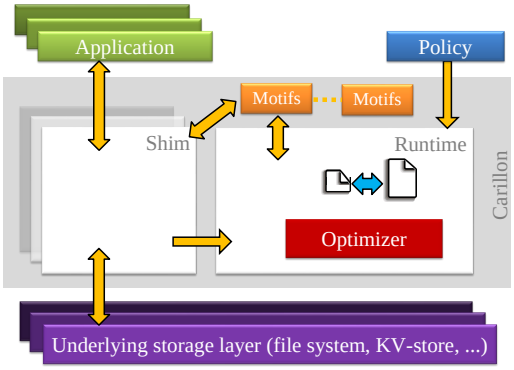


Figure 2: ***The Carillon architecture****. The yellow arrows show interface calls and callbacks.*

(i.e., the raw bytes corresponding to it). A motif's *expand* method consists of arbitrary code: it can fetch data across the network, run computations, or access local storage. We describe motifs in the context of a filesystem for ease of exposition. When a file is associated with a motif, it can exist in contracted form as the motif, or its contents can be generated using that motif. The following are several key properties of motifs.

Motifs are recursive. A motif's *expand* method can access other files. For example, in Figure 1, A.txt is a contracted motif which expands by performing some computation over B.txt. The file being accessed could be a conventional file (like B.txt); alternatively, it could also be a motif. For example, in Figure 1, a file C.txt is an index generated by scanning a data file D.txt, which in turn is a local copy of a remote file accessed via a URL. Expanding the index file requires the data file to be expanded first; accordingly, the motif for the index file depends on the motif for the data file. Motifs can also execute binaries (such as `zip` or `curl`), which in turn could exist as motifs.

Motifs are stateful. A motif consists of two components: the code executed to generate the file contents, and a small amount of metadata used as input for this code. For example, the motif for D.txt in Figure 1 consists of code to actually download the file over the network, along with the URL of the remote source.

Motifs can define circular dependencies. Files that can be generated from each other lead to circular motif dependencies. One example of this is compression: in Figure 1, C.txt.zip is the compressed version of a file C.txt, which means the bytes for C.txt can be generated by uncompressing C.txt.zip; conversely, the bytes for C.txt.zip can be generated by compressing C.txt. Accordingly, the motifs for C.txt and C.txt.zip depend on each other. Another example of a circular dependency involves two files storing the same data sorted on different columns; each file can be generated by sorting the other. A third example involves files storing different data structures with the same data: for instance, a hash map and a tree representation of a set of items.

Files can have multiple motifs. In cases where a file can be reconstructed via more than one method, multiple motifs can be associated with it. For instance, in Figure 1, C.txt can be generated by uncompressing C.txt.zip, or by generating an index over D.txt; depending on the load on the network, storage system and CPU, as well as whether D.txt is expanded or contracted, it might be faster to use one motif versus the other.

Motifs can be invalidated. If a motif depends on other files in the same filesystem – either conventional files or other motifs – it is automatically contracted when those files change. It must be expanded again before it can be read. As a result, motifs do not expose stale data to applications. This does not apply to motifs that depend on external sources like URLs on the web.

Motifs can support writes. A motif can optionally contain a *contract* method. For read-only files, contraction requires no extra code; it merely involves deleting the raw bytes of the file and retaining the motif. However, in some cases, an expanded file can be modified by the application, and these changes have to be relayed upstream to the original source of the data. For example, if a motif expansion involves fetching data over the network, its contraction might involve writing that data back to the remote location, effectively making the local file a write-back cache. The *contract* method is not allowed to change other files in the same filesystem, to prevent the consistency snarl that can arise if writes occur in motif dependency cycles.

3. Carillon Design

Carillon is a system that implements the motif abstraction. A primary design goal is to add motif-based storage elasticity to existing storage services – such as filesystems and key-value stores – with minimal effort. To achieve this goal, Carillon uses a two-part design (as shown in Figure 2), consisting of a runtime and a shim. The runtime is agnostic to the target storage system, while the shim is tailored to it; each new storage service requires a new shim that exposes its API to applications. A single runtime/shim pair operates in concert with a single storage service. If multiple storage services are executed on the same machine, each one requires its own Carillon runtime and shim. In addition, motifs are specific to shims, even if they have substantially similar functionality; this is because they need to interact with the shim to access and write out the appropriate data units (e.g., files or key-value pairs).

The Carillon runtime is responsible for managing motif metadata, including the mapping between opaque data identifiers (i.e., filesystem filenames or key-value store keys) and motifs. It accepts policies from the administrator regarding the target size of the storage system, and tracks access and size statistics about units of data (such as individual files). Based on the policies and statistics, it triggers motif expansion and contraction to change the footprint of the storage

```
  //create a motif for a new data item
t_motif create_motif(t_ID oid, t_templateID mtmp,
                     void *motifstate);
  //attach a motif to an existing data item
t_motif attach_motif(t_ID oid, t_templateID mtmp,
                     void *motifstate);
  //detach a motif from an existing data item
t_motif detach_motif(t_ID oid, t_templateID mtmp);
  //notify of open and close
void notify_open(t_ID oid);
void notify_close(t_ID oid);
```

Figure 3: ***API exposed by Carillon runtime to shim.***

service on the fly. Further, it executes motifs within its own address space.

The Carillon shim intercepts calls to the target storage system and exposes the corresponding API to applications, along with motif-specific calls which we'll describe shortly. It interacts with the Carillon runtime using the API shown in Figure 3. We use the running example of a filesystem shim (Carillon-FS) which exposes a POSIX API to applications.

3.1 The Runtime API

We now describe the interaction of the shim (using the filesystem as an example) with the runtime. When a file is opened, the shim calls *notify-open* on the runtime. The runtime returns immediately if the file is either an expanded motif or a conventional file; else, it expands the motif before returning. When the file is closed, the shim calls *notify-close* on the runtime.

When the motif executes, it is responsible for writing out the generated bytes to the underlying storage system. To do so, it interacts with the shim's upstream API (i.e., the POSIX API in the case of the filesystem) but uses a special PASS_THROUGH flag to indicate that it wants to directly write to the underlying storage system. When the motif wants to access other files in the filesystem, it uses the shim without the PASS_THROUGH flag, to ensure that required motif expansions are triggered for its dependencies. The PASS_THROUGH flag is also used on contraction, either by the runtime or the motif's *contract* method, to delete the bytes inside the file.

To allow the installation of motifs in the system, the Carillon runtime exposes three API calls (shown in Figure 3) to the shim, which in turn exposes them to the application. The application invokes these calls on the shim with parameter types specific to a storage API (e.g., with filenames); the shim routes the calls to the runtime in a form that's independent of storage API (e.g., passing filenames as opaque identifiers).

First, applications can call *create-motif* to create a new mapping between a data identifier and a motif. In doing so, they provide both the code for the motif and its metadata. To simplify motif creation, Carillon provides a library of motif templates from which individual motif instances can be created. An example call to *create-motif* might pass in a filename (e.g., `/tmp/abc`), a motif template (e.g., one which downloads a URL), and the metadata for the motif (e.g., the URL to download from). Consider the filesystem example; when the *create-motif* call returns from the runtime to the shim, and from the shim to the application, at that point a new file exists in the filesystem, but in contracted motif form.

Second, applications can call *attach-motif* to attach a new motif to an existing data identifier. This is similar to *create-motif*, except the data unit already exists; in the context of a filesystem shim, this attaches a motif to the filename, but leaves the file in expanded form. This file can now be contracted – i.e., its contents can be deleted – since a motif exists to reconstruct it. Finally, applications can call *detach-motif* to dissociate a motif from an identifier.

In addition, the runtime provides APIs (not shown in Figure 3) that allow the shim to update it with access statistics, either eagerly or lazily, as files are read and written. The runtime then uses these statistics to choose which files to contract. Finally, the policy API exposed by the runtime is currently very simple – it accepts the target size of the Carillon instance.

4. Implementation

We now discuss in more depth our implementation of the various Carillon components.

4.1 Motif Implementations

In our Carillon implementation, motif templates are dynamically loadable C++ modules that implement the *expand* method (and an optional *contract* method). The modules are pluggable and can be installed or upgraded during run-time. Individual motifs are created by passing a motif template and motif-specific metadata to the *create-motif* call described in the previous section. If the motif reads other data units in the same storage system (that could also be motifs), it is required to explicitly specify dependencies in the metadata at creation time.

We now detail some of the motif templates that are implemented in our system. We describe the motifs we used for the filesystem shim.

Network-storage motif: An example of a network-storage motif template implementation used by our system is shown in Figure 4. The remote server is expected to contain copies of files, as discussed earlier. Therefore, read-only files can simply be removed during contraction. For mutable files, however, the *contract* method of the network-storage motif ensures that the remote site contains an up-to-date copy of the data before removing its local copy. The *expand* method then copies the file back from the remote server, in our implementation using `scp`. Notice that any motif created with the network-storage motif template has a dependency on the `scp` binary.

Compression motif: The next motif template uses file-level compression to save storage space at the expense of higher

```
int contract(struct context *ctx) {
 int res = execute(
    "ssh %s \"mkdir -p '`dirname \"%s%s\"`'\"",
    IP, PATH, ctx->path);
 if(res == 0)
   res = execute("scp \"%3$s\" '%s:\"%s%s\"'",
                 IP, PATH, ctx->path);
 return res;
}

int expand(struct context *ctx) {
 return execute(
    "scp '%1$s:\"%2$s%3$s\"' \"%3$s\"",
    IP, PATH, ctx->path);
}

static struct motif m = {
    .name     = "compress-motif",
    .contract = contract,
    .expand   = expand,
};

struct motif *init() { return &m; }
void cleanup() { }
motif_init(init);
```

Figure 4: ***Network-storage motif example.****. A file is retrieved from remote server during expand, and mutable files that may have changed are uploaded before local removal by the contract routine. The listing omits error handling and security issues for clarity.*

CPU utilization. File compression is implemented as two motifs, *Compress-Motif* and *Decompress-Motif*, that induce a dependency cycle of length 2 between an original file and its compressed version, since one file can be created from the other. Given a file A, the application first creates a new compressed file A.zip with a *Compress-Motif* with A as its argument. Next, the application attaches the *Decompress-Motif* to the original file A specifying a name (here A.zip) of the compressed file that was created and exposed on the underlying storage system. Consequently, the *Decompress-Motif* for A can decompress the dependent file A.zip to recreate the content of A, and the *Compress-Motif* for A.zip can recreate the content of A.zip by compressing the original A file. In our implementation, we use `gzip` as the compression utility.

Browser downloads motif: Recalling that workstations often use storage for caches of various kinds, the next motif template illustrates how Carillon facilitates better use of such caches in the context of a web browser. Many major web browsers, such as Mozilla Firefox and Google Chrome, store copies of downloaded files in a "Downloads" folder. These files are removed from the folder only manually, causing a tendency for the folder to fill up over time. The contents of the Downloads folder are tracked by an internal database of the browser. In the case of Firefox, a `places.sqlite` database maintains information about what files have been fetched and the URL from where they were initially downloaded.

We built *Downloads-Motif*, a simple motif template that is parameterized with a URL. Expanding the motif causes the URL to be fetched. To use the *Downloads-Motif* with the Firefox Downloads folder, we wrote a script to scan the Downloads database and create a *Downloads-Motif* for each motif. Before deciding to contract a file, the *Downloads-Motif* sends an HTTP HEAD request (with no cookies or authentication tokens) to verify that the file can be retrieved later.

4.2 The Carillon Runtime

The workhorse of Carillon is the runtime, which we implemented as a daemon. The runtime stores its metadata – the mapping from filenames to motifs – in a single file on the base filesystem (*i.e.*, the filesystem outside the Carillon universe, containing the runtime executable). The size of this metadata is proportional to the number of installed motifs; each motif is quite compact, since it consists only of an object identifier, a motif template identifier, and metadata for the motif. The metadata is typically a small number of identifiers that the motif depends on, either pointing to other data units or to external objects (e.g., URLs).

The runtime is implemented in 2563 lines of C++ code. The runtime interacts with the shim via IPC, exposing the API described in Section 3 for creating and managing motifs. The IPC mechanism used is Apache Thrift RPC [3], which allows for easy development of shims for new storage systems. The runtime also exposes a policy API to management tools, which can be used to set resource limits (i.e., the total space that can be used by the Carillon instance) and to collect statistics. These statistics can in turn feed into an automatic management tool, such as to automatically partition storage space across VMs [16].

Within the runtime is an optimization module that decides what objects to expand or contract based on external resource pressure while trying to minimize user-experienced latency. The details of the process are discussed below. Contractions are performed in the background, whereas expansions are triggered upon request from the shim as the application accesses data units.

4.3 Security challenges

Normally, Carillon runtime and all motifs are configured and managed by the system administrator. Multiuser systems can be an exception, however, since individual users may choose to use motifs and contribute to optimize resources on a shared system [8, 24]. All motifs are currently run under the privileges and capabilities of the user of the process that makes the request to Carillon.

To prevent the situation where user u compromises the security or privacy of another user u' by registering malicious motif code, we require that users specifically vet motif codes contributed by regular users u who have fewer privileges or capabilities than u' before they use the code. Motif templates installed by an administrator are thus always enabled.

Motifs may contain errors that could cause the system to hang or damage files. We currently take a *laissez-faire* approach and assume that developers provide correct motifs. We intend to improve our rudimentary sandbox around motifs to help mitigate stability and security concerns.

5. The Optimizer Module

At the heart of the Carillon runtime is an optimization module that decides what resources should be consumed through the use of motifs to save on other more valuable resources. For example, disk space could be saved by contracting a rarely used file on a filesystem by a motif at the penalty of longer wait times on the next `open` system call. There are several challenges in determining the impact of different choices and making a good choice.

First, we must estimate when the file will next be needed. Proper estimates guard against wasted efforts of contracting files that shortly after require additional resources to be expanded again. An optimal estimate would depend on knowledge of future accesses, putting the problem in the same class as cache replacement policies. Second, we must model which and how many resources are used and saved through contraction and then later consumed during future expansion. The resources span network, storage and computation resources and thus depend on dynamic usage patterns. Third, a file may be contracted by one of multiple applicable motif templates. Different motifs have different resource profiles, so whereas one motif may save on storage space by consuming network resources, another motif may reduce storage space in exchange for higher CPU load.

To model the problem, the input consists of an online sequence of $(\text{object-name}, \text{time})$ pairs specifying objects that are to be opened by some application. Our challenge is to ensure that each object is in expanded (readable) form when it is accessed, potentially waiting for the expansion to complete, while simultaneously adhering to the specified restrictions on resource consumption.

We can make online decisions without future knowledge by reformulating the problem in the knapsack framework. We assign a *profit* and *size* value to each object-motif pair that may be contracted. The profit is calculated as a difference between resource savings and the expected latency during future expansion. The latter term encapsulates both the estimated time until reuse as well as assessing the latency for that access. We calculate the size of each pair in the knapsack to be the current resource usage of the object. An approximation algorithm for knapsack will then contract the objects that give the most profit without violating resource constraints.

In a typical scenario in Carillon, thousands of objects are being considered for contraction. Unfortunately, even calculating approximate solutions for multi-resource knapsack is prohibitively expensive [12]. We are forced to simplify to make progress.

A natural approach to our original problem is to consider local storage as a cache, available for the expenditure of other resources. This perspective lets us tap into the extensive cache-replacement algorithm literature to decide what objects to contract. However, cache replacement strategies may potentially discard too much of the information provided by the system. To determine the impact, we will investigate the performance of both simple and more complex strategies on realistic workloads.

6. Applications

To illustrate how Carillon enables elastic storage, we implemented two applications: Carillon-FS, a file-system shim that uses motifss to manage storage, and Carillon-KV, a key-value store whose data can be preloaded and removed using motifs. Further, we built a simple graph database application on Carillon-KV to show how elasticity can accelerate practical applications. Below, we discuss key aspects of these implementations.

6.1 Application: Carillon-FS

Carillon-FS is a Carillon-based POSIX filesystem implemented using Linux FUSE [18]. The implementation comprises 685 lines of C++ code. Carillon-FS is a Carillon shim: files in Carillon-FS are stored on an underlying filesystem, which in our set-up was `ext4`. When a file is contracted, its bytes are removed from the underlying filesystem, while the Carillon runtime maintains it as a motif. An empty, 0-byte token file is left behind on the filesystem. The motif can subsequently be expanded to repopulate the file.

Carillon-FS passes most operations directly to the underlying filesystem implementation in the kernel with a few important exceptions detailed below.

stat If a file is contracted, we can not rely on the underlying filesystem to fulfill the `stat` request, since the size of the token file is zero bytes. Expanding the file on a `stat` is wasteful. In this case, we issue a lookup RPC to Carillon. The Carillon stored metadata contains full `stat struct` about the last expanded state of the file, which we return to the caller.

open When opening a file, Carillon-FS must ensure the file exists in a fully expanded form. To satisfy this request, we must therefore send an *expand* request to Carillon. When *expand* returns, we can assume the file is fully expanded even if was previously contracted.

unlink If a file is contracted we must take special care to clean up any state when a file is permanently unlinked. A motif's *contract* method may do arbitrary operations with various side-effects, including storing metadata on a remote site. We must therefore forward this call to the motif responsible for the file.

rename The Carillon runtime must be made aware of the new name for this particular motif. The shim currently

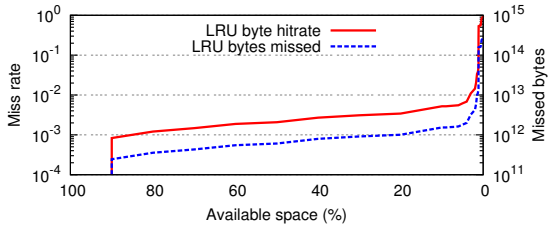


Figure 5: ***Working set size.*** *LRU histogram showing byte miss rate and total bytes missed for LRU caches for different cache sizes.*

implements `rename` by calling *detach-motif* and then *attach-motif* on the new filename, thus retaining the atomicity of the underlying file system. An optimized version would affect only metadata in the daemon, thus speeding up patterns such as the creation of temporary files. However, we found that rename was a rare operation in our traces ($< 0.01\%$) and thus take the detach-attach approach in our prototype.

truncate Truncate removes an arbitrary portion of a file. In the general case such a request can not be fulfilled except by calling *expand* first.

utime This method modifies the `stat struct`. If a file is in a contracted form, we must update the stat metadata stored by Carillon. We do this by first calling `Lookup` to provide the current `stat` for the file, and then notifying the runtime of an access to the file, which updates the relevant metadata stored by Carillon.

Example look-up in Carillon-FS. Figure 8 illustrates the steps taken when a contracted file stored in Carillon-FS is opened by an application. Carillon-FS first checks if the file is present on its backing storage system, say an ext4 filesystem. If the file is already expanded, the data are in place and the call just proceeds as normal. Assuming the file is instead in a contracted state, Carillon-FS send an RPC query to the Carillon runtime to verify that the file has been contracted and to prepare for expansion. Carillon consults an internal database and locates the motif for the file to be expanded. The metadata in the motif contains sufficient information to expand the file. The motif's expand function now runs and writes the expanded file to the appropriate path on Carillon-FS's backing storage system. After expansion, the Carillon RPC successfully returns. Carillon-FS now attempts to open the file on the backing storage system, which will now succeed. It returns the file descriptor to the user program, and the user reads or modifies the file.

6.2 Application: Carillon-KV

We also implemented a key-value shim on top of Carillon called Carillon-KV. Carillon-KV runs over a LevelDB instance, and exposes the LevelDB API to applications. It also adds Thrift code [3] to interact with the runtime. The total shim size is 670 lines of C++ code.

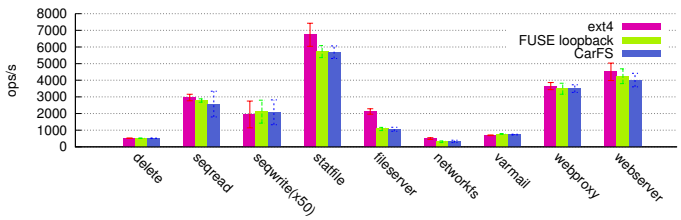


Figure 6: ***Performance overhead.*** *Benchmarks on Carillon-FS using* filebench *[22]. Error bars represent one sample standard deviation.*

Graph database: To facilitate experiments with the elasticity afforded by Carillon-KV, we built a simple graph database on top of Carillon-KV. The graph store stores a weighted digraph in Carillon-KV using an adjacency list representation: each vertex is associated with a map between vertices and their weights. The API includes the basic graph operations including enumerating all vertices in the graph, finding outgoing edges from a given vertex, and to add an edge between a pair of vertices of a given weight.

7. Evaluation

Our evaluation aims to answer the following three questions. **(1)** What is empirically a good method for choosing files to contract? **(2)** Does Carillon provide space elasticity at a reasonable cost? **(3)** What is the performance overhead of Carillon-enabled services?

7.1 Experimental Setup

We conducted our experiments on a dual-core 2.0GHz Intel i5 4310U processor machine with 16GiB DRAM running Arch Linux 3.18.6-1 and FUSE 2.9.3-2 [18]. The one exception is that the filebench benchmark suite [22] was run on a QEMU 0.12.1.2 virtualized machine with one Intel E5-2695v2 2.4GHz core and 4GiB DRAM running CentOS 6.6 with 64-bit Linux 2.6.32-431 and FUSE 2.8.3 using a 7x900GB 10K RPM SAS drives configured in RAID-6. To accommodate the network storage motif, we used another virtualized machine on the same LAN with the same specification as the one above as an upstream server.

7.2 Traces

Our evaluation relies on two real-world traces.

DevTrace: The first trace is a longitudinal log of all system calls on a developer's Ubuntu Linux laptop. The trace spans 7 months of daily use and contains approximately 77M system calls.

CourseTrace: Using `Sysdig` [17], we also collected two weeks of system call data from a live code autograding server for a 100 person computer science university course. The trace contains over 300M system calls, of which 39M are `open` calls that can trigger expansion. We also took a snapshot of the directory structure, filename and file sizes at the beginning of the trace. For our experiments, we created a

copy of the filesystem directory structure and filled each file with random bytes of the appropriate size.

To avoid triggering spurious kernel calls and polluting the kernel caches when parsing a trace file from disk, we automatically generated and compiled C code from each trace that successively generates every system call related to the filesystem in the trace (the longest code is 10MLOC).

7.3 What data should we contract?

We evaluated several cache-replacement algorithms for choosing objects to contract, including random replacement, LRU, LFU, FIFO, CLOCK and S4LRU. CLOCK maintains a circular list of objects and traverse it using a hand, decrementing the counter of an object whenever the hand passes the object, and resetting the counter if the object is used. Objects are contracted when the counter reaches 0 [6]. S4LRU divides an LRU into four segments, where items are first brought into the lowest segment. An object which is repeatedly accessed is promoted to a higher segment, whereas idle objects are demoted to lower segments or evicted altogether as in ordinary LRU [10]. The algorithms were run with many different parameter values and we report only on the parameters that yielded the best performance.

We ran simulations over all `open` calls in the DEVTRACE and COURSETRACE traces. We set a target storage capacity to be either 20% or 50% of the system's total storage space as a policy. We recorded the *byte miss rate*, gauging the effectiveness of the algorithm with variable size objects, and the *total bytes missed*. In the latter, we recorded the full size of each requested file and sum up sizes of missed files. This aggregate acts as a proxy for the expected latency of the expansion of the file. For example, the duration of copying a file to a remote server depends linearly on the file size. The policy that minimizes total missed bytes will thus best approximate the optimal strategy.

The algorithms all had similar performance for the two metrics and hit rates of over 99.5% (figure omitted due to space constraints). LRU gave the most competitive performance – the small working set (Figure 5) means that all algorithms have a low miss rate. While this is characteristic of filesystem traces [14], ***the strong locality implies that recency of access is a significantly more important factor to decide on contraction than the anticipated expansion overhead.*** We ran an LRU simulation on DEVELOPERTRACE for various space constraints to create an LRU histogram (Figure 5) to investigate the locality of the traces. The figure shows that the miss rate stays low until about 1% space when it enters a cliff, showing that byte accesses are concentrated around a very small set of files. We adopted LRU as the default algorithm in Carillon and used it in the subsequent experiments.

7.4 How much overhead is imposed by Carillon ?

For our experimental evaluation of Carillon services, we begin by subjecting Carillon-FS to standard benchmarks. We compare the performance of Carillon-FS against `ext4` and a FUSE loopback implementation which forwards all system calls directly to the kernel to highlight the overhead FUSE incurs for an extra context-switch into user space.

We use filebench, a filesystem and storage benchmark suite that can generate both micro and macro benchmarks [22]. The micro-benchmarks issue common filesystem specific system calls to large files. The macro workloads are synthesized to emulate the behavior of common applications like web and mail servers. We ran four micro benchmarks and five macro benchmarks. The results in Figure 6 show that the ***overhead of Carillon in Carillon-FS is less than 5% compared to the FUSE loopback driver***.

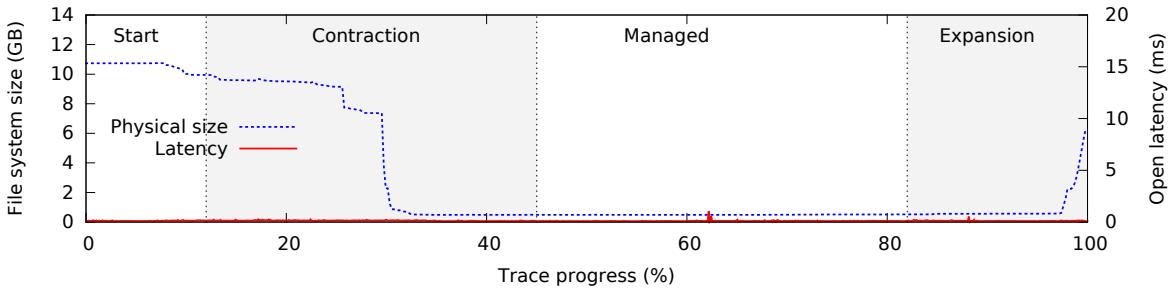
7.5 Does Carillon-FS achieve space elasticity?

We evaluate the elastic cloud storage application by focusing on elasticity and performance within a single VM. We replay COURSETRACE on a volume managed by Carillon-FS as the Coordinator process changes our resource policy over time to adjust the space partition. To focus on the additional overhead imposed by Carillon-FS, the storage volume is backed by RAM. Initially, no restrictions are put on the storage use. After approximately $1/3$ of the trace has run, we significantly reduce the space available to the system. After $2/3$ of the trace, we lift the constraints again. In the experiments, Carillon exclusively uses a network storage motif. We repeated the experiment with various kernel and buffer cache settings and observed minimal differences in performance.

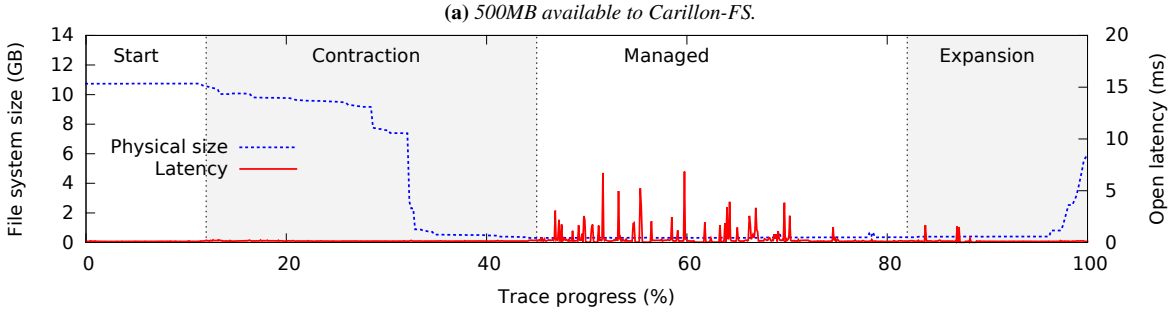
Our performance metric is the time to complete `open` system calls, since this reflects the principal overhead of file expansion. We expect limited overhead during the first third of the trace, some increases during the era of constrained capacity, and finally low overhead in the last third.

On the first run, we constrain the system to 500 MB during the middle phase (Figure 7a). The blue dashed line represents the total size of the system at each point in time. The red line represents the average time of an `open` call over each 20 second period.

We ran the trace with different settings to investigate the causes of the relatively low overhead seen in the figure. We changed the policy and constrained the middle managed phase of the trace to enforce 0 bytes managed space. This change cause most `open` operations to trigger an expansion, and thus increasing overhead. The behavior is confirmed in Figure 7b. The graph further shows that the total system size never outgrows approximately 400 MB, reaffirming the conventional wisdom that real-world traces exhibit high locality: a small set of files on the systems are responsible for most activity on the system. The distribution of `open` latencies confirm that vast majority of files do not cause significant overhead. Hence, as long as space is available to keep these popular files expanded, the overhead of the occasional expansion of rarely accessed files is minor.



(a) *500MB available to Carillon-FS.*



(b) *No space available to Carillon-FS.*

Figure 7: ***Carillon-FS performance.*** *Average latency of* `open` *calls and Carillon-FS filesystem size on* COURSETRACE. *The storage limit policy is changed from unrestricted to restricted and then back during the trace, where the size restriction is (7a) 500MB, and (7b) 0MB.*

7.6 Does Carillon-KV achieve space elasticity?

We argued that Carillon allows motifs to elastically use excess space to reduce use of other resources.

We ran a basic route planning workload against Carillon-KV, which stores a large weighted network representing road map data, and finds the shortest driving distance between a given source and destination. We used the road map of the State of California, (21K vertices, 43K edges), for our experiments. Crucially, this simple service illustrates how applications may benefit from elastic storage: retaining intermediate computations reduces computation cost of future queries but at the cost of storage consumption.

The graph database uses Dijkstra's shortest-path algorithm to compute an optimal path between s and t. As input, we run shortest path calculations between random pairs of Californian cities with gradually greater preference for larger populations (Section 7.6). The intermediate state calculated by the algorithm – sets of predecessors and distance estimates – can be useful to accelerate future queries, but is commonly discarded. We use the graph database to address this issue.

We modified the Dijkstra calculations to memoize intermediate paths within the graph store, thus enabling Carillon to manage the storage footprint. The planar nature of the road map workload ensures that memoization works very well in this use case. We attach a motif to each memoized path that regenerates it by executing the shortest path query between the source and the destination.

Figure 9 shows the results of our evaluation on over 100,000 source-destination pairs. We make three policy

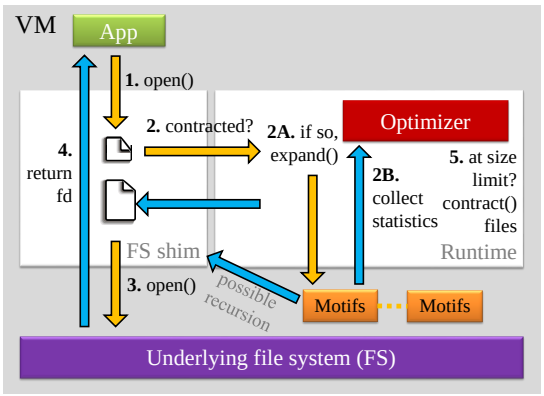


Figure 8: ***Look-up in an elastic filesystem****: timeline of an* `open` *request to Carillon-FS. The Carillon-FS layer informs Carillon that it is opening a file. Contracted files are expanded via the corresponding motif, and the time required for expansion is recorded for future reference. Expansions may in turn trigger other expansions. Once the file is expanded, the original* `open` *request proceeds.*

changes during the trace. We begin with an unrealistic limit of no excess storage, causing every request to be answered by a full call to Dijkstra's algorithm. At 15%, we allow up to 1GB of intermediate calculations to be stored. The immediate consequence is a higher CPU load since Carillon automatically precomputes state for the most popular cities using information about the most recently evicted entries from its internal LRU. This choice accentuates the dramatic drop in computation time, all the while storage usage grad-

ually increases to store more intermediate state. At 65%, we reduce the storage capacity limit to 10MB. Space used for intermediate calculations is quickly released and the latency increases accordingly, although remaining lower than at the beginning since Carillon maintains the useful entries in memory for the application. This experiment illustrates a scenario where motifs allow Carillon to optimize for *storage capacity* during the first and third phases, and for *CPU cycles* in the second phase.

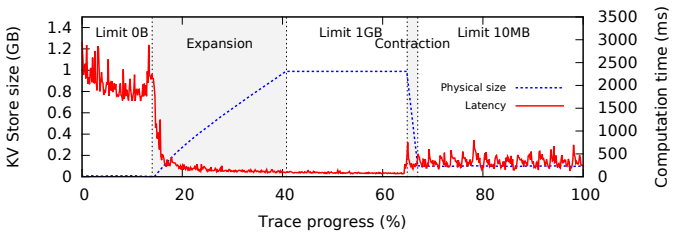


Figure 9: ***Elastic Graph Store.*** *Computation time and space usage of a shortest-path application on top of our Carillon-KV shim. The application calculates shortest paths in a route network as we vary storage resources available to cache intermediate computations.*

8. Related Work

Exploiting excess storage. Data caches for avoiding expensive recomputation have been the staple of the memory hierarchy for many decades. Most caches are of fixed size and are fully utilized in the steady state. When a cache fills up, eviction decisions are predominantly made based on the recency and frequency of accesses, ignoring the often variable resource costs of cache misses for different items [7].

Excess space can also be used to provide redundancy for stronger durability of data, thus implicitly saving the cost of recreating files (e.g., the Elephant filesystem by Santry *et al.* [15]). A central question is which information to forget when storage space is scarce, resembling the decision when to transform between representations of data in Carillon.

Using context to reduce storage footprint. Hasan and Burns claim that unintentional and unneeded data, so-called *waste data*, is growing rapidly and call for digital waste data management strategies [9]. Such strategies could be implemented as motifs and automatically carried out by Carillon.

Zadok *et al.* detail how automatically reducing storage consumption can decrease management overhead and device lifetimes in a multi-user environment [24], introducing compression, downsampling or removal strategies when a user's quota is exceeded. These policies are akin to our motifs, except motifs allow for programmability and support for precomputation to alleviate loads by using more storage.

Nectar is a distributed system that manages the storage volumes used for dataset computation within data centers [8]. Nectar explicitly explores the data-computation tradeoff, but unlike Carillon, it makes fundamental assumptions that restrict the generality of input data (only streams), the execution environment (all programs are LINQ programs) and the generality of transformations (more restricted than motifs). Apache Spark uses a resilient distributed dataset abstraction, which tracks the lineage of data so that it may be rebuilt when errors occur [25].

The systems and database communities have made significant progress on making complete histories of data modifications and movements – *data provenance* – practical on modern machines. A motif can leverage such histories to create alternate representations for how a given piece of state could be derived. A Carillon-based system could then under storage pressure, for instance, choose to discard history of old files partially or completely.

An earlier workshop version of this paper proposes a filesystem for space elasticity, but does not generalize it to a framework [16].

Other space saving techniques. Identifying and removing redundant parts of files within a system, *data deduplication*, has received significant attention in both academia and industry as a method for decreasing storage costs [4, 5, 13, 20]. Deduplication can be implemented within Carillon via motifs.

Several distributed systems balance performance with storage overhead. SpringFS [23] changes the number of active storage servers depending to meet elasticity and performance targets, passively migrating data in the background. Sierra [19] and Rabbit [2] seek to reduce power consumption of their systems by manipulating storage. These systems maintain one or more copies of each file to achieve their goals, whereas Carillon-FS allows between zero and one copies to exist of a file. Further, Carillon-aware storage systems achieve storage elasticity through application-level information via programmable motifs, any one of which could implement the replication logic used by these systems.

9. Conclusion

Modern applications routinely store soft state on durable media to accelerate performance, which can be counterproductive if storage is scarce or slow compared to the network and CPU. The motif abstraction allows applications to specify to the storage system how soft state can be reconstructed via network I/O or computation. Carillon makes it easy to build new storage systems that use motifs. Carillon-based systems can dial down and dial up resource usage on the fly, enabling elasticity in the new and vital dimension of storage capacity.

Acknowledgments

We are grateful to anonymous reviewers for constructive feedback on the paper and earlier submissions. Our work is partially supported by NSF CAREER award #1553579 and funds from Emory University and Reykjavik University.

References

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[2] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan. Robust and flexible power-proportional storage. In *SoCC*, pages 217–228, 2010.

[3] Apache Foundation. Apache thrift framework. `https://thrift.apache.org/`, 2015.

[4] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Trans. on Knowl. and Data Eng.*, 24(9):1537–1555, Sept. 2012. ISSN 1041-4347. . URL `http://dx.doi.org/10.1109/TKDE.2011.127`.

[5] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 8–8, Berkeley, CA, USA, 2009. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855807.1855815`.

[6] F. J. Corbato. A paging experiment with the multics system. Technical report, DTIC Document, 1968.

[7] S. Ghandeharizadeh, S. Irani, J. Lam, and J. Yap. CAMP: A cost adaptive multi-queue eviction policy for key-value stores. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 289–300, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2785-5. . URL `http://doi.acm.org/10.1145/2663165.2663317`.

[8] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, pages 75–88, 2010.

[9] R. Hasan and R. Burns. The life and death of unwanted bits: Towards proactive waste data management in digital ecosystems. In *Proceedings of the 3rd International Conference on Innovative Computing Technology (INTECH)*, August 2013.

[10] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. .

[11] Jeffrey Dean, Sanjay Ghemawat. LevelDB on-disk key-value store. `https://github.com/google/leveldb`, 2011.

[12] A. Kulik and H. Shachnai. There is no eptas for two-dimensional knapsack. *Information Processing Letters*, 110 (16):707–710, 2010.

[13] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14, 2012.

[14] M. Mitzenmacher. Dynamic models for file sizes and double pareto distributions. *Internet Mathematics*, 1(3):305–333, 2004.

[15] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 110–123, New York, NY, USA, 1999. ACM. ISBN 1-58113-140-2. . URL `http://doi.acm.org/10.1145/319151.319159`.

[16] H. Sigurbjarnarson, P. O. Ragnarsson, Y. Vigfusson, and M. Balakrishnan. Harmonium: Elastic cloud storage via file motifs. In M. A. Kozuch and M. Yu, editors, *6th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud '14, Philadelphia, PA, USA, June 17-18, 2014*. USENIX Association, 2014. URL `https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/sigurbjarnarson`.

[17] Sysdig. Sysdig: system troubleshooting tool. `http://www.sysdig.org`, 2015.

[18] M. Szeredi. Filesystem in userspace. `http://fuse.sf.net`, 2003.

[19] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *EuroSys*, pages 169–182, 2011.

[20] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855511.1855528`.

[21] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[22] A. Wilson. The new and improved FileBench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies*, 2008.

[23] L. Xu, J. Cipar, E. Krevat, A. Tumanov, N. Gupta, M. A. Kozuch, and G. R. Ganger. SpringFS: Bridging agility and performance in elastic distributed storage. In *FAST*, pages 243–255, 2014.

[24] E. Zadok, J. Osborn, A. Shater, C. P. Wright, K. Muniswamy-Reddy, and J. Nieh. Reducing storage management costs via informed user-based policies. In B. Kobler and P. C. Hariharan, editors, *21st IEEE Conference on Mass Storage Systems and Technologies / 12th NASA Goddard Conference on Mass Storage Systems and Technologies, Greenbelt, Maryland, USA, April 13-16, 2004.*, pages 193–197. IEEE, 2004. URL `http://storageconference.org/nasa/conf2004/Papers/MSST2004-21-Zadok-a.pdf`.

[25] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.