Enabling Space Elasticity in Storage Systems

Pétur Orri Ragnarsson

Thesis of 60 ECTS credits
Master of Science (M.Sc.) in Computer Science

January 2016

Enabling Space Elasticity in Storage Systems

Thesis of 60 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

January 2016

Supervisor:

Ýmir Vigfússon, Supervisor
Assistant Professor, Emory University, USA

Examiner:

Mahesh Balakrishnan, Examiner
Associate Professor, Yale University, USA

Marcel Kyas, Examiner
Assistant Professor, Reykjavik University, Iceland

Enabling Space Elasticity in Storage Systems

Pétur Orri Ragnarsson

January 2016

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. Thus, an opportunity for elastic storage, where system administrators 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 space elasticity of storage by allowing applications to describe how soft state can be regenerated. Harmonium is a system that uses motifs to dynamically change the storage footprint of applications. Harmonium is implemented as a runtime and a collection of shim layers that interpose between applications and specific storage systems; we describe shims for a filesystem (HarmFS) and a key-value store (HarmKV). We explore some strategies on how to select data to remove when space is scarce and how Harmonium might be used to on an increasingly unreliable storage backend. Finally, we show that HarmFS allows us to dynamically alter the storage footprint of a VM, while we use HarmKV to build a graph database that accelerates common queries when given extra storage space.

Geymslurými gagna gert sveigjanlegt

Pétur Orri Ragnarsson

janúar 2016

Útdráttur

Gagnageymslur eru gerðar til að tapa aldrei gögnum. Nútímakerfi nýta hins vegar aðgengilegt pláss til að auka afsökst með því að vista hverful gögn eins og niðurstöður, forsótt gögn eða forreiknuð föll. Upp úr því rís hugmyndin um teygjanlegt geymslupláss, þar sem kerfisstjórar geta breytt plássnýtingu forrita með því að fjarlægja eða endurskapa hverful gögn eftir framboði auðlinda og notkunarmynstrum. Við leggjum til nýja hlutfirringu (e. abstraction), kallaða mótíf, sem gerir geymslupláss teygjanlegt með því að leyfa forritum að lýsa því hvernig hægt er að endurskapa hverful gögn. Harmonium er kerfi sem nýtir mótíf til að breyta plássnýtingu forrita sjálfvirkt. Harmonium er útfært sem bakgrunnsforrit og safn af millilögum sem koma milli forrita og geymslukerfa. Við sýnum millilög fyrir skráakerfi (HarmFS) og lyklageymslu (HarmKV). Við skoðum hvernig er best að velja gögn til að fjarlægja þegar skortur er á plássi og hvernig nýta mætti Harmoinium þegar geymslukerfið verður óáreiðanlegra með aldrinum. Að lokum sýnum við hvernig er hægt að nota HarmFS til að aðlaga plássnýtingu sýndarvélar og notum HarmKV til að smíða grafa-gagnagrunn sem svarar algengum fyrirspurnum hraðar þegar aukapláss er til staðar.

Enabling Space Elasticity in Storage Systems

Pétur Orri Ragnarsson

Thesis of 60 ECTS credits submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science (M.Sc.) in Computer Science

January 2016

Student:

.....

Pétur Orri Ragnarsson

Supervisor:

.....

Ýmir Vigfússon

Examiner:

.....

Mahesh Balakrishnan

.....

Marcel Kyas

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing has gained tremendous popularity in recent years. One of its main features is how easy it is to increase computing power at a moment's notice, and conversely to give it up when it's no longer needed. No spare capacity has to be kept around for times of high resource usage, lying idle inbetween. Only the resources used are paid for. These cloud resources come in the form of virtual machines, which come in all sizes, ranging from less powerful than a cheap smartphone to monsters with dozens of cores and hundreds of gigabytes of memory. Depending on circumstances resources can be adjusted either by modifying individual virtual machines by adding or removing cores or memory (vertical scaling) or by spinning up or down entire machines (horizontal scaling).

However, in a typical scenario, the storage space these machines use can only be adjusted in one direction; to increase. The reason for this limitation is the traditional maxim of storage systems: No data must ever be lost. According to this promise, all bytes written to permanent storage are of equal and paramount importance. However, modern applications don't always behave in accordance to this principle. They use durable storage media for increased performance, rather than durability. This shift is driven by hardware trends: Larger disks allow application developers to think of creative ways to use the extra space, while solid state drives can function as a cheaper alternative to RAM.

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 [1], 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 an anecdotal example, Figure 1.1 shows that between 25-55% of storage on our own workstations is consumed by caches and ephemeral contents. In addition, durability may not be critical for some data either because new applications (such as big data analytics) can provide useful answers despite missing data [2], or because the data may be duplicated across multiple locations [3], [4].

We have exposed an inefficient dynamic between storage systems and applications: Applications opportunistically use any space available to them for nonessential data, while the storage systems struggle to make sure none of the data is lost. The applications do this for valid reasons, to increase their own efficiency. However, there is little or no cooperation between applications and hard disks fill up with data that, while useful, isn't crucial and can be safely removed at the cost of some performance. We would therfore like to build a system that has more oversight over data on the system. When free space is scarce the system should be able to identify less important data and remove it. On the other hand, the system

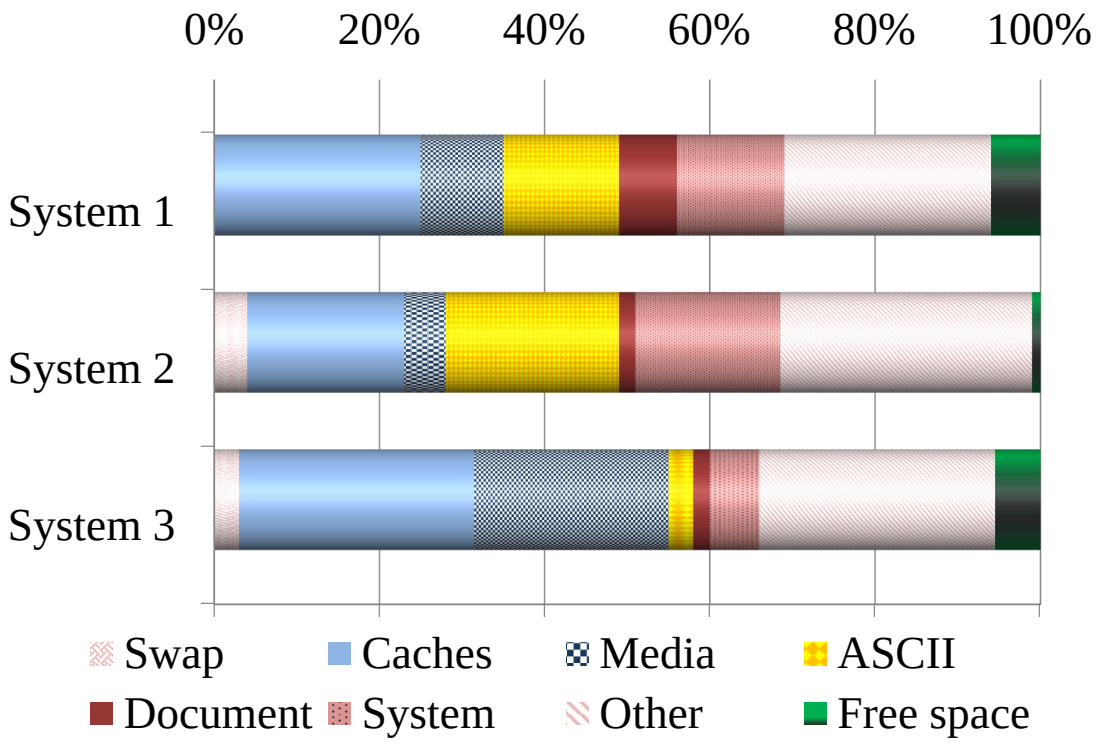


Figure 1.1: Classification of data types stored on three representative developer systems, showing large volume of ephemeral content. The sampled systems have 256 to 512 GB total storage space each.

should be able to go in the other direction and recreate removed data when space becomes plentiful.

To refine our requirements, the system needs the following properties: 1) Identifies which data is safe to remove. 2) Removes data that's suspected not to be in use. 3) Recreates removed data when it's accessed.

In addition we would like our system to have the following properties: 4) Opaque to existing applications for ease of integration. 5) Low overhead.

Such a system would enable elasticity of storage, at least when there is data that's safe to remove.

To this end, 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 data by fetching data over the network from a URL, or via computation over other data (e.g., sort a file, merge multiple input files, generate an index, or even expand a compressed input file). Motifs allow us to fulfill properties 1 and partially number 3, by assigning them to files that are safe to remove and we know how to regenerate.

We implement the motif abstraction within a system called Harmonium, 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, enabling property 2 and the rest of 3. 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, fulfilling property 4. 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 Harmonium shims and their corresponding real-world applications. We implement a filesystem shim (Harmonium-FS) using FUSE with a regular file system as a backing store. We execute Harmonium-FS as a guest filesystem within multiple virtual machines, interacting with corresponding Harmonium runtime instances running on the host OS. With the help of file-based motifs, Harmonium-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, showing that we fulfill property 5 as well.

We also implement a key-value store shim (Harmonium-KV) that runs over – and exposes the API of – LevelDB [5]. Above Harmonium-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 Harmonium-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, Harmonium-KV was able to reduce latencies 10× through the use of motifs.

This paper makes the following contributions:

- 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 Harmonium that implements the motif abstraction. Harmonium can be extended with thin shims to add space elasticity to any existing storage system.
- We describe two Harmonium 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.

Chapter 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 (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 2.1, *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 2.1, 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

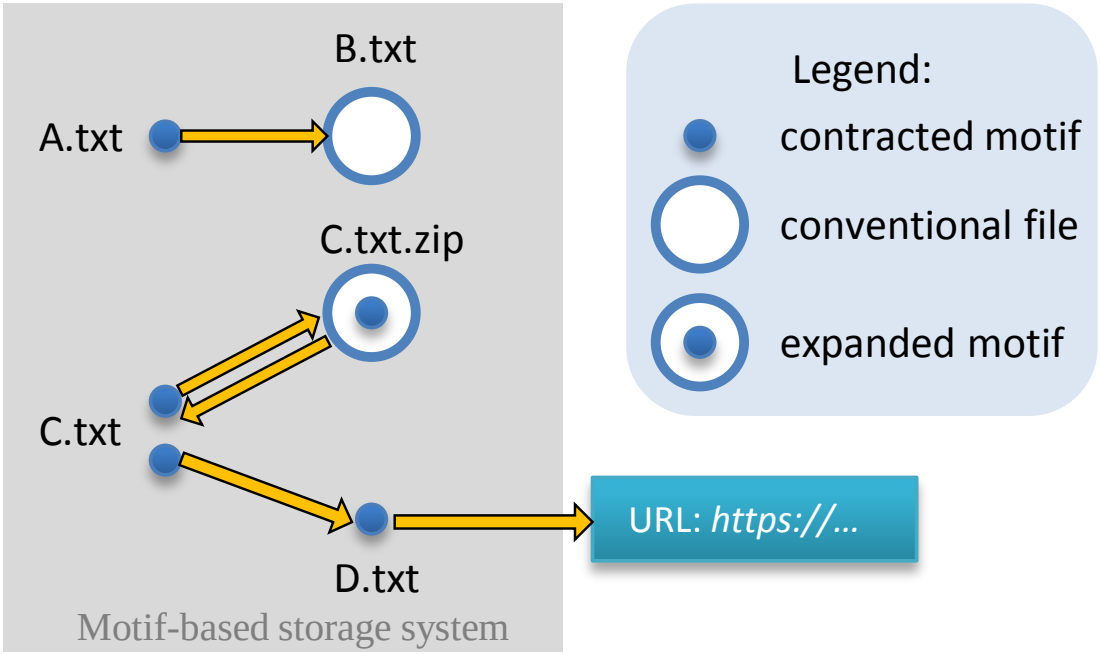


Figure 2.1: **Motifs** exist in expanded or contracted form; depend on other files, motifs, and external resources; and can have circular dependencies.

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 instance 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 2.1 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 2.1, *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 2.1, *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. Motifs that depend on external sources like URLs on the web are not automatically invalidated.

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.

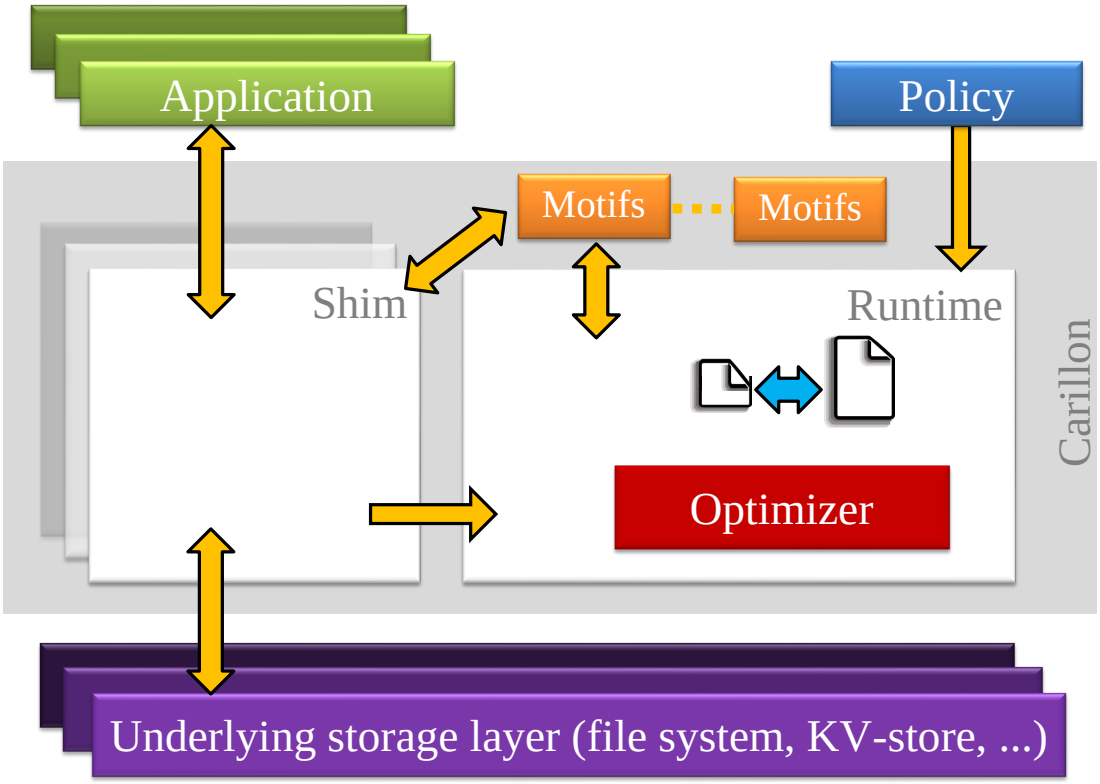


Figure 3.1: **The Harmonium architecture**. The yellow arrows show interface calls and callbacks.

Chapter 3

Harmonium Design

Harmonium 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, Harmonium uses a two-part design (as shown in Figure 3.1), 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 Harmonium runtime and shim. In addition, motifs are specific to shims, even if they have substantially similar functionality; this is because they

```
  //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 3.2: **API exposed by Harmonium runtime to shim**.

need to interact with the shim to access and write out the appropriate data units (e.g., files or key-value pairs).

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

The Harmonium 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 Harmonium runtime using the API shown in Figure 3.2. We use the running example of a filesystem shim (Harmonium-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 Harmonium runtime exposes three API calls (shown in Figure 3.2) 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, Harmonium 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, albeit 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 3.2) 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 Harmonium instance.

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 Harmonium 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 item 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).

Chapter 4

Implementation

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

4.1 Motif Implementations

In our Harmonium 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 4.1. A remote server has some files available and motifs are created for them. To *expand* a file they are copied to the local file system. If the file is read-only it can be simply 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. 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 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 Harmonium 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.

```
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 4.1: **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 code listing ignores error handling and security issues for clarity.

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.

4.2 The Harmonium Runtime

The workhorse of Harmonium is the runtime, which we implemented as a daemon. The runtime is implemented in 2563 lines of C++ code. The runtime interacts with the shim via IPC, exposing the API described in Figure 3 for creating and managing motifs. The IPC mechanism used is Apache Thrift RPC [6], 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 Harmonium 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 [7].

Within the runtime is an optimization module that decides what items 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 The Optimizer module

At the heart of the Harmonium 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 (item-name, time) pairs specifying items that are to be opened by some application. Our challenge is to ensure that each item 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 item-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 item. An approximation algorithm for knapsack will then contract the items that give the most profit without violating resource constraints.

In a typical scenario in Harmonium, thousands of items are being considered for contraction making exact solutions to the NP-hard knapsack optimization problem impractical. Unfortunately, even calculating approximate solutions for multi-resource knapsack is prohibitively expensive [8]. 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 items 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.

We evaluated the following standard cache replacement algorithms.

Random Contract an item at random when needed.

LRU Contract the least recently used item.

LFU Contract the least frequently used item.

FIFO Contract the oldest item.

CLOCK Maintain a circular list of items and traverse it using a hand, decrementing the counter of an item whenever the hand passes the item, and resetting the counter if the item is used. Items are contracted when the counter reaches 0 [9].

S4LRU Contract least recently used item among four segments of LRU, where items first appear in the lowest LRU segment, and are then promoted to higher segments upon repeated access, with evictions from segments working as ordinary LRU [10].

The results of our evaluation Figure 6.3 show that the very high locality of filesystems accesses [11] means that the Harmonium optimizer that contracts files in an LRU fashion will gain little by factoring future expansion costs into account.

4.4 Security challenges

Normally, Harmonium 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 [1], [12]. All motifs are currently run under the privileges and capabilities of the user of the process that makes the request to Harmonium.

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.

Chapter 5

Applications

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

5.1 Application: Harmonium-FS

Harmonium-FS is a Harmonium-based POSIX filesystem implemented using Linux FUSE [13]. The implementation comprises 685 lines of C++ code. Harmonium-FS is a Harmonium shim: files in Harmonium-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 Harmonium 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.

Harmonium-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 Harmonium. The Harmonium 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, Harmonium-FS must ensure the file exists in a fully expanded form. To satisfy this request, we must therefore send an *expand* request to Harmonium. 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 Harmonium runtime must be made aware of the new name for this particular motif. The shim can implements `rename` by calling *detach-motif* and then *attach-motif* on the new filename. In our traces (Sec. 6), we found that rename was an

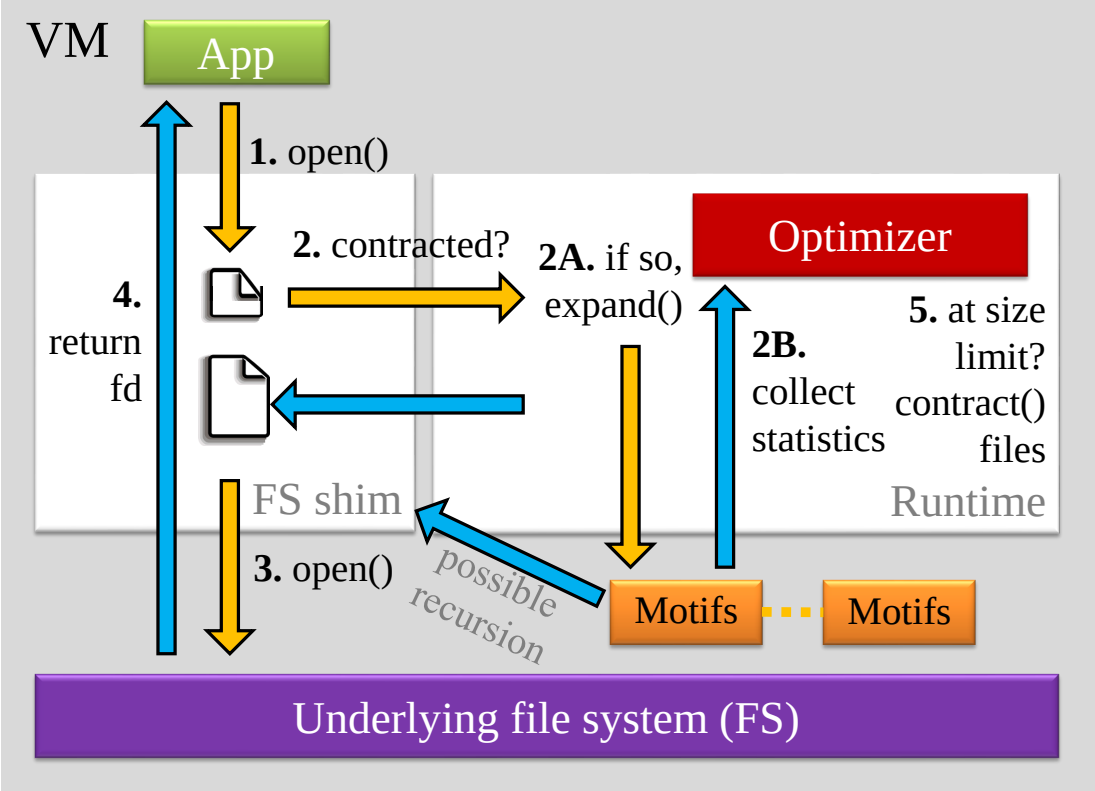


Figure 5.1: **Look-up in an elastic filesystem**: timeline of an `open` request to Harmonium-FS. The Harmonium-FS layer informs Harmonium that it is opening a file. Contracted files are expanded via the corresponding motif, and the time required for expansion is recorded for future reference. Expansions may in turn trigger other expansions. Once the file is expanded, the original `open` request proceeds.

extremely rare operation (fewer than 0.01%) and thus omitted the functionality 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 Harmonium. 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 Harmonium.

5.2 Example look-up in Harmonium-FS

Figure 5.1 illustrates the steps taken when a contracted file stored in Harmonium-FS is opened by an application. Harmonium-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, Harmonium-FS send an RPC query to the Harmonium runtime to verify that the file has

been contracted and to prepare for expansion. Harmonium 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 Harmonium-FS's backing storage system. After expansion, the Harmonium RPC successfully returns. Harmonium-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.

5.3 Application: Harmonium-KV

We also implemented a key-value shim on top of Harmonium called Harmonium-KV. Harmonium-KV runs over a LevelDB instance, and exposes the LevelDB API to applications. It also adds Thrift code [6] to interact with the runtime. The total shim size is 670 lines of C++ code.

Graph database: To facilitate experiments with the elasticity afforded by Harmonium-KV, we built a simple graph database on top of Harmonium-KV. The graph store stores a weighted digraph in Harmonium-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.

Against this graph database, we ran a basic route planning workload, 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 entire road map of the State of California, (21K vertices, 43K edges), for our experiments. Crucially, this simple service illustrates how applications can benefit from elastic storage: retaining intermediate computations will reduce 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. 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 modify the shortest path algorithm to create motifs for its internal state in the Harmonium-managed key-value store at the end of each run instead of throwing it all away. During subsequent runs some required values will correspond to expanded motifs and don't require full calculation. This allows Harmonium to manage the storage footprint of the route planning application.

5.4 Potential application: Decaying media

Some storage media deteriorate over time. In particular, SSD cells are known to become increasingly likely to record an erroneous bit as they are written and erased over time. This opens up an interesting use case for Harmonium. A user requires his disks to be correct with some high probability, and when they can no longer provide this probability they must be discarded. It is common for SSDs to cross this threshold after 5000 or 10000 write/erase cycles [14].

SSDs currently include a fixed amount of error correction bits for each cell, but when the errors become too numerous for these to correct, the drive must shut down. However, if it were possible to add extra error correction bits to the data on an aging drive the overall file would still be correct within the required probabilty. These extra bits would consume some

of the available space while increasing the reliability of data. This is clearly something right up Harmonium's alley.

The scenario looks like this: A system is running Harmonium on a slowly deteriorating SSD. Once a certain unreliability threshold is reached a program runs [1] and adds error correction bits to every file on the drive [2]. This operation may require more space than is available so Harmonium kicks in and reduces files according to whatever motifs are available. As time goes on further unreliability thresholds are reached more and more space is devoted to error correction until eventually so little space is left for real data that the drive is effectively unusable.

This demonstrates a use case for Harmonium where it can be used for a gradual tradeoff between space efficiency and performance.

A simple EC scheme

Let us assume that the EC bits on a block can detect more errors than it can fix, as is common in EC schemes. If the preexisting EC can correct a particular error, we need not do anything. When the error is too large we mark that block as damaged.

In the following paragraph we devise and evaluate a simple scheme to provide the dynamic error correction outlined above. Readers familiar with the RAID5 disk storage scheme will see a close resemblance, with the major differences that this is on a single physical drive and the amount of parity blocks is increasing over time.

A naive way to add EC to a device that can already detect its errors is to split it up into chunks of n blocks and for each chunk store a parity block with the XOR of all the blocks in the chunk. This way it is possible to recover the loss of a single block in each of the chunks by XORing all the remaining blocks and the parity block. When the device ages n is reduced and the parities rewritten to reflect the new chunk layout. This scheme sacrifices $1/n$ of the total available space to EC.

The main problem with this approach is that it causes two actual writes to the disk for each write the user performs, one for the data and one for the parity. This accelerates the aging of the disk and therefore reduces the increase in lifespan by half.

The worst case age acceleration is doubling the writes: Each time the user writes a block we update the parity. Fortunately it is common for the user to update many consecutive blocks at the same time. If these all have the same parity block we can save some writes by only updating the parity once for the entire request. To evaluate how large these savings are we aquired some block device traces from various types of servers [3], including a database server, build server, authentication server and more.

We ran the entire collection of traces through a block device simulator and calculated what the actual write overhead would be. The results are shown in figure 5.2.

From the figure we can see that the write overhead begins at about 20% for $n = 100$ and slowly increases until $n = 20$ where it is approximately 24%. With lower n the overhead increases rapidly. From this data we can calculate the extension in the usable lifetime of the disk. To formalize this we define the usable life of a disk to be the total number of block writes possible before it is no longer usable. We can then see that the usable life is the integral of the graph in Figure 5.3, which shows the usable amount of space of the disk for

[1] The details of this program are left as an exercise to the reader. Two options include building it into Harmonium or mounting another FUSE layer on top of Harmonium.

[2] This cannot be done incrementally with current drives since the FTL is an inaccessible black box that may shuffle bits around without the user touching them.

[3] http://iotta.snia.org/tracetypes/3

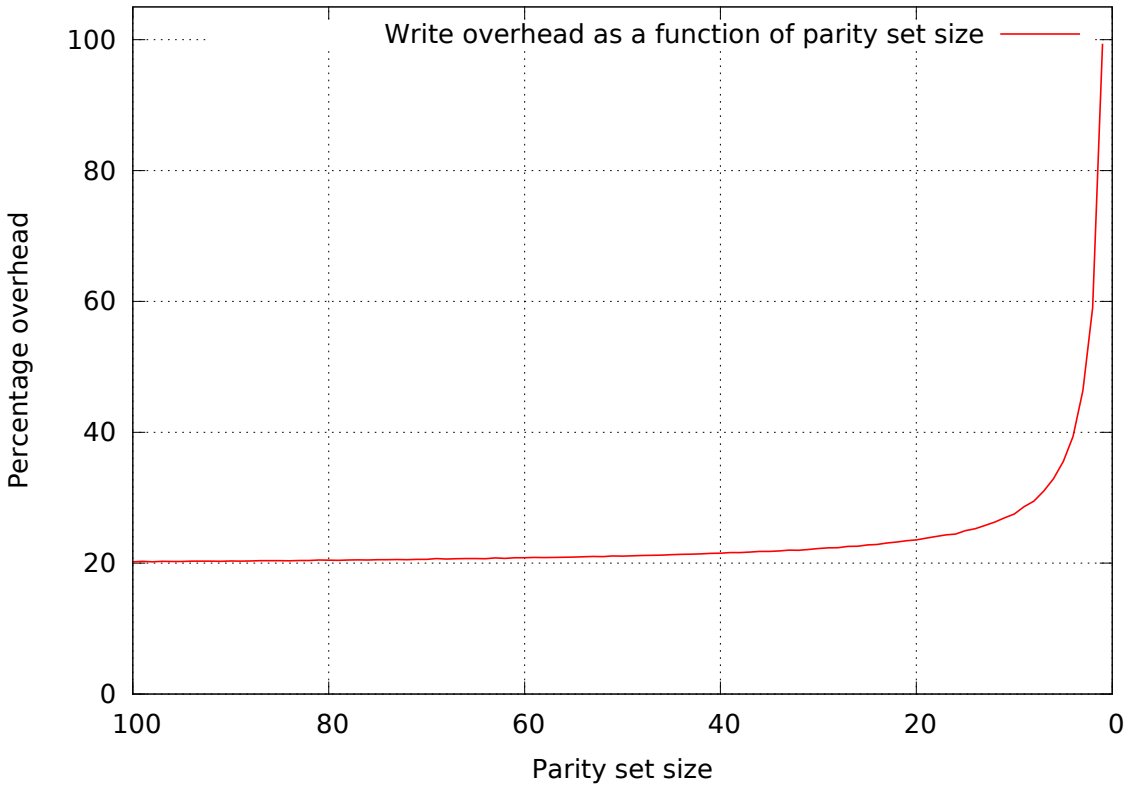


Figure 5.2: Average write overhead for each level of error correction.

each write cycle by the user on the disk. From this graph we conclude that the increase in lifetime is slightly over 40%.

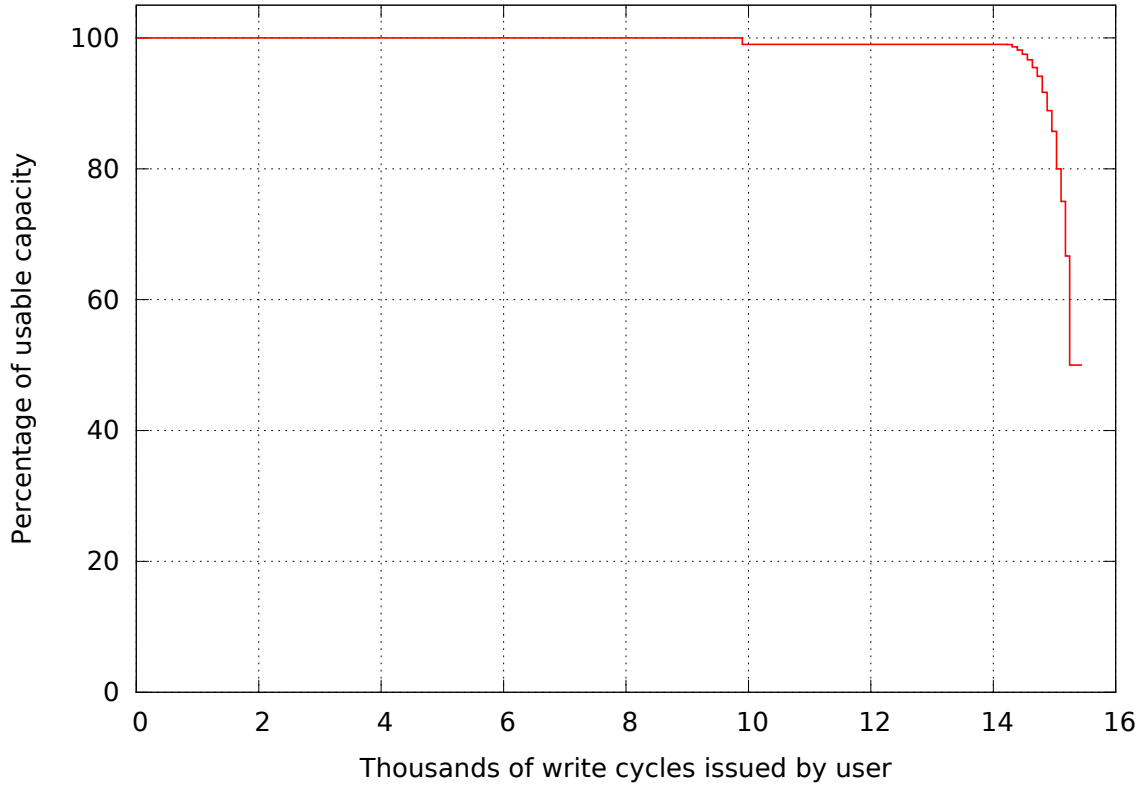


Figure 5.3: Usable percentage of space for each write cycle by the user

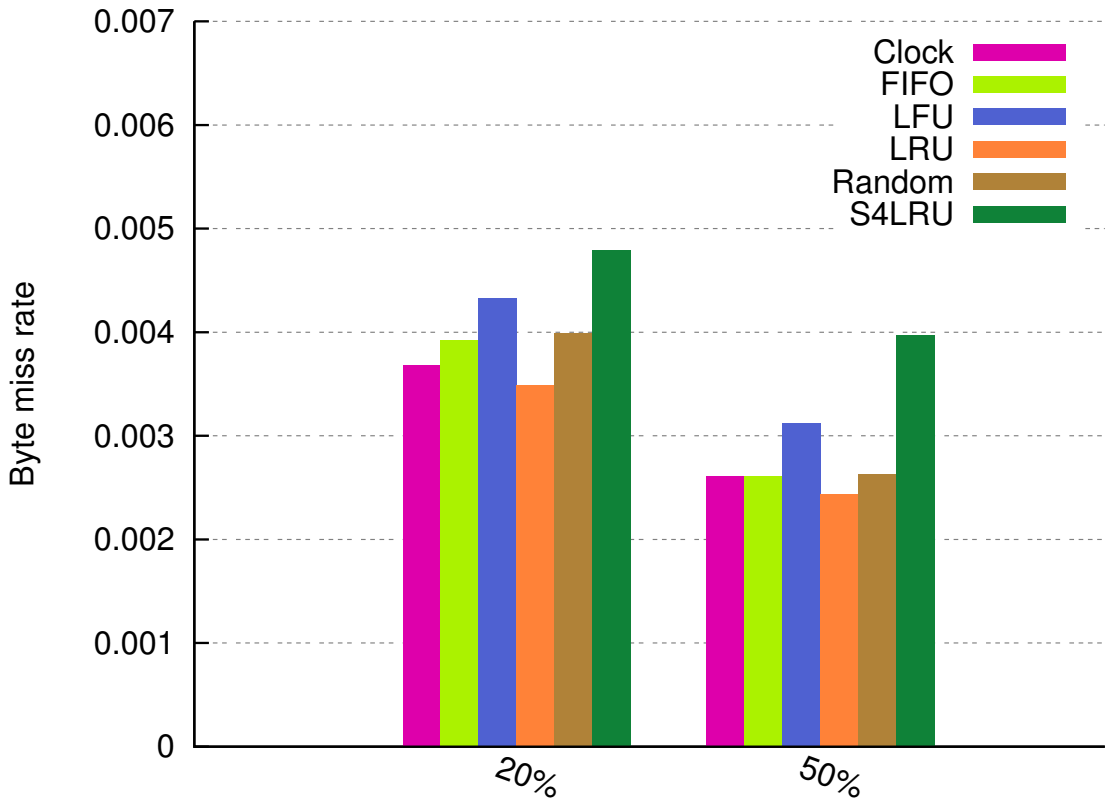


Figure 6.1: **Choosing files to contract.** Byte hit rate performance of several cache replacement algorithms on two traces with storage constraints of 20% and 50% of the total available storage.

Chapter 6

Evaluation

Our evaluation aims to answer the following three questions.

- What is empirically a good method for choosing files to contract?
- Does Harmonium provide space elasticity at a reasonable cost?
- What is the performance overhead of Harmonium-enabled services?

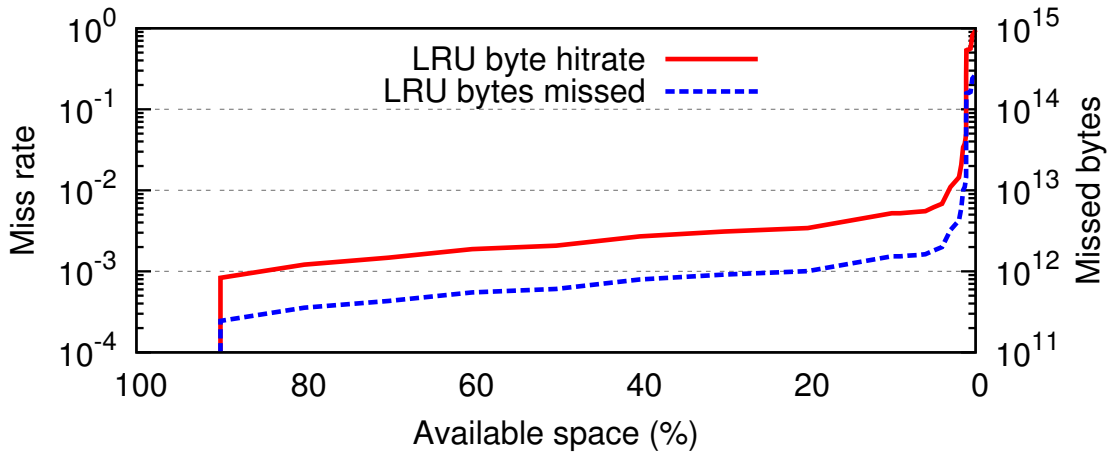


Figure 6.2: **Working set size.** LRU histogram showing byte miss rate and total bytes missed for LRU caches for different cache sizes.

6.1 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 [13]. The one exception is that the filebench benchmark suite [15] 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.

6.2 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` **sysdig** 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).

6.3 What data should we contract?

We evaluated different cache-replacement algorithms for choosing items to contract. All algorithms were run on two different traces, each with two different resource constraints.

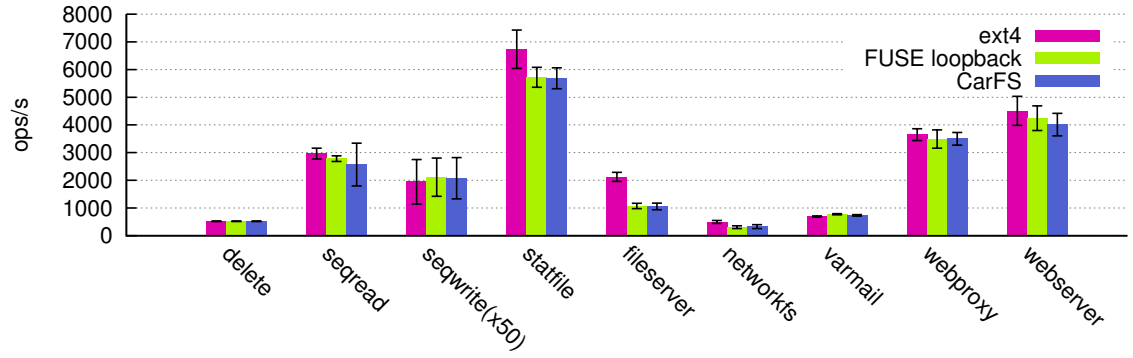


Figure 6.3: **Performance overhead.** Benchmarks on Harmonium-FS using filebench [15]. Error bars represent one sample standard deviation.

We run simulations over all open calls in the DevTrace and CourseTrace traces. For our policy, we set a target storage capacity to be either 30% or 50% of the system's total storage space. We run the traces under these constraints and record two metrics.

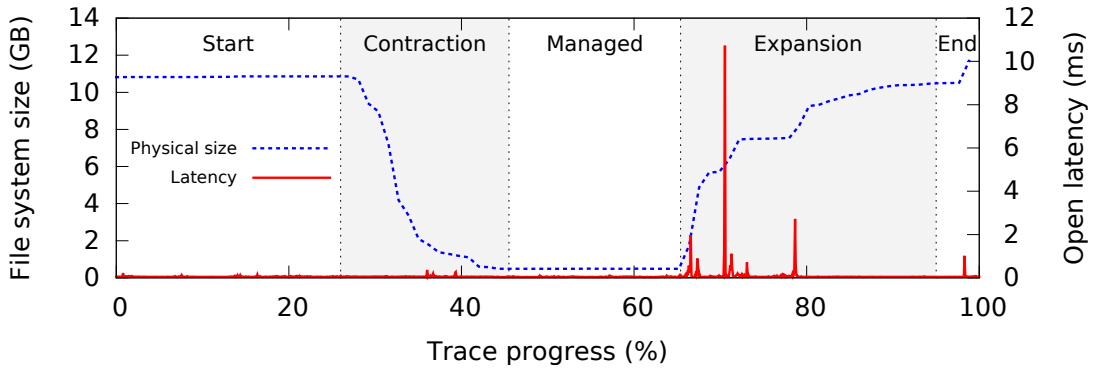
First, the *byte miss rate* gauges on the effectiveness of a cache replacement algorithm with variable size items. Second, we record a statistic closely related to the definition of an optimal contraction-expansion solution: the *total bytes missed* by the policy. We thus record the full size of each requested file and sum up the 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 graphs in Figure 6.1 show that the standard cache replacement algorithms all have similar performance for the two metrics. The small working set (Figure 6.2) means that all algorithms have a low miss rate. While this is characteristic of filesystem traces [11], ***the strong locality implies that recency of access is a significantly more important factor to decide on contraction than the anticipated expansion overhead.*** Indeed, LRU gives the most competitive performance – the exception being CourseTrace (50%). We ran an LRU simulation on DeveloperTrace for various space constraints to create an LRU histogram (Figure 6.2) 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 Harmonium and use it in the following evaluations.

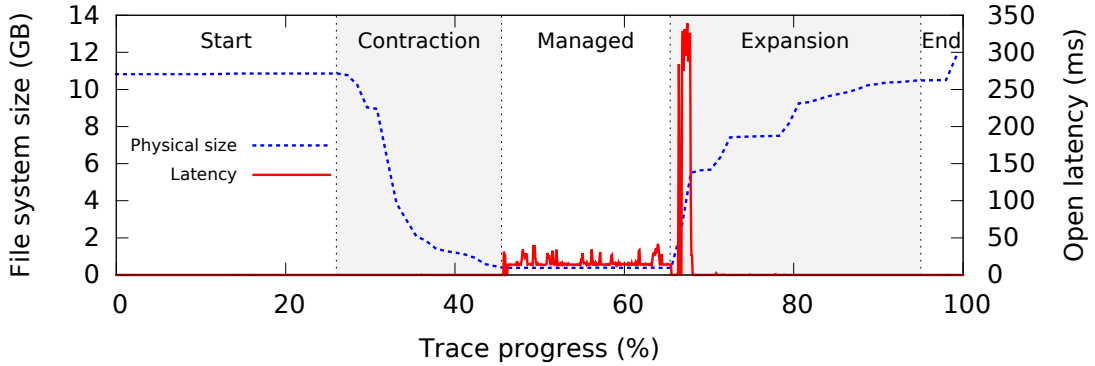
6.4 How much overhead is imposed by Harmonium ?

For our experimental evaluation of Harmonium services, we begin by subjecting Harmonium-FS to standard benchmarks. We compare the performance of Harmonium-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 [15]. 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 6.3 show that the ***overhead of Harmonium in Harmonium-FS is less than 5% compared to the FUSE loopback driver***.



(a) 500MB available to Harmonium-FS.



(b) No space available to Harmonium-FS.

Figure 6.4: **Harmonium-FS performance.** Average latency of open calls and Harmonium-FS filesystem size when running on CourseTrace. The storage limit policy is changed from unrestricted to restricted and then back during the trace, where the size restriction is (6.4a) 500MB, and (6.4b) 0MB.

6.5 Does Harmonium-FS achieve space elasticity?

Phase	Count	Stdev	Mean	Min	5%	Median	95%	Max
Start	3402497	228	45	7	33	39	63	271437
Managed	12038815	3801	74	17	29	42	68	1375868

(a) 500MB available to Harmonium-FS.

Phase	Count	Stdev	Mean	Min	5%	Median	95%	Max
Start	3402497	223	45	8	33	40	64	206646
Managed	453780	27200	10985	20	35	63	34721	1199208

(b) No space available to Harmonium-FS.

Table 6.1: Statistics for **Harmonium-FS**. Each number except *Count* is given in microseconds. *Count* is the number of open requests during that phase.

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 Harmonium-FS as the Coordinator process changes our resource policy over time to adjust the space partition. 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, Harmonium 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 6.4a). 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.

The spikes in overhead after the space constraints are lifted are due to eager background expansion of files when more space is made available to prevent those files from incurring overhead when they are next accessed. We intend to throttle the rate of background expansion to minimize the impact on `open` latency owing to context-switches.

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 6.4b. 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.

Table 6.1 shows some extra statistics about the Harmonium-FS performance evaluation during the *Start* and *Managed* phases, where we expect the system performance to be in a steady state. (Ie. not performing extra work to do active contraction or expansion.)

6.6 Does Harmonium-KV achieve space elasticity?

We argued that Harmonium allows motifs to elastically use excess space to reduce use of other resources. To evaluate this statement, we take the illustrative application built on top of Harmonium-KV for finding shortest-paths in a graph, subject it to an experiment and measure the application performance and the storage space elasticity.

As input, we run shortest path calculations between random pairs of Californian cities with gradually greater preference for larger populations (Figure 5.3).

Figure 6.5 shows the results of our evaluation on over 100,000 source-destination pairs. We make three policy 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 Harmonium 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 gradually 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 Harmonium maintains the useful entries in memory for the application. This experiment

illustrates a scenario where motifs allow Harmonium to optimize for *storage capacity* during the first and third phases, and for *CPU cycles* in the second phase.

Table 6.2 shows some additional statistics for this evaluation. The three lines correspond to the three different policies we set during the experiment. We ignore the adjustment phases (marked *Expansion* and *Contraction* in Figure 6.5).

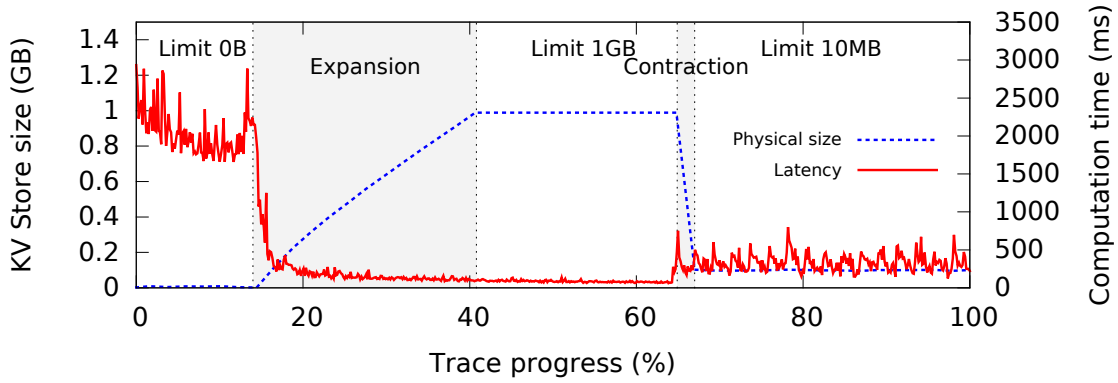


Figure 6.5: **Elastic Graph Store.** Computation time and space usage of a shortest-path application on top of our Harmonium-KV shim. The application calculates shortest paths in a route network as we vary storage resources available to cache intermediate computations.

Limit	Count	Stdev	Mean	Min	5%	Median	95%	Max
0B	980	611	2004	< 1	1585	1714	2957	6401
1GB	39776	149	84	< 1	< 1	1	171	2889
100MB	15525	521	297	< 1	< 1	63	1620	2870

Table 6.2: Statistics for the **Elastic Graph Store**. Each number except *Count* is given in milliseconds. *Count* is the number of requests made to the graph store during that phase.

Chapter 7

Related Work

Exploiting excess storage. Trade-offs between storage and computation time are fundamental in computer science and have been extensively investigated by the algorithms [16] and cryptography [17], [18] communities, as well as other subfields.

Data caches for avoiding expensive recomputation have been the staple of the memory hierarchy for more than half a century. 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 [19].

Excess space can also be used to provide redundancy for stronger durability of data, thus implicitly saving the cost of recreating files. The Elephant filesystem by Santry *et al.* leverages surplus storage to allow accidentally deleted files to be recreated [20]. A central question is which information to forget when storage space is scarce, resembling the decision when to transform between representations of data in Harmonium.

Using context to reduce storage footprint. The proliferation and rapidly growing data collection threatens to offset the exponential increases in storage capacity we have enjoyed for three decades. Hasan and Burns claim that unintentional and unneeded data, so-called *waste data*, is growing rapidly and call for digital waste data management [21]. They advocate for systems to reuse, recycle, recover and even disposing of files into a *digital landfill* to minimize waste. Such strategies could be implemented as motifs and automatically carried out by Harmonium.

Zadok *et al.* detail how automatically reducing storage consumption can decrease management overhead and device lifetimes in a multi-user environment [12]. They modify a filesystem to maintain elastic disk quotas by enforcing compression, downsampling or removal policies of certain file types when a user's quota is exceeded. These policies are akin to our motifs, except without programmability or support for precomputation to alleviate loads by using more storage.

Sigurbjarnarson *et al.* presented a Python filesystem prototype for enabling elastic cloud storage [7]. They define a notion of a motif on which we build, but their design does not enable elasticity in arbitrary storage systems.

Nectar is a distributed system that manages the storage volumes used for dataset computation within data centers [1]. It automatically and transparently removes unneeded intermediate datasets that fill up space, and recomputes them from the original dataset and a LINQ program later if needed. Computation in Nectar involves a series of functional transformations from original datasets, resembling stream processing. Nectar explicitly explores the data-computation trade-off, but unlike Harmonium, 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).

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 [3], [4], [22], [23]. The savings can be substantial: Meyer and Bolosky found that over 20% of files on the desktop machines at Microsoft were redundant – 50% if all the computers were grouped together [3]. With more aggressive deduplication of blocks across files, the figures increased to respectively 35% and 70%. Harmonium supports applications using motifs with deduplication techniques for condensing or compressing data at the computational cost of needing to piecing the data back together in the future.

Network storage and archival filesystems have long been used to store data no longer required to live on primary storage. These systems are commonly LAN-based [24], [25] but personal cloud storage services such as DropBox [26] have gained popularity. Most of these systems replicate the content on both systems, commonly for backup and robustness purposes, and do not automatically offload content to save space on the primary storage.

Several distributed systems balance performance with storage overhead. SpringFS [27] changes the number of active storage servers depending to meet elasticity and performance targets, passively migrating data in the background. Sierra [28] and Rabbit [29] 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 Harmonium-FS allows between zero and one copies to exist of a file. Further, Harmonium-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.

Sources for motifs. The systems and database communities have made significant progress on making complete histories of data modifications and movements – *data provenance* – practical on modern machines For instance, Devecsery *et al.* recently built a system that tracks complete lineage of all state on a computer with modest computational (8% CPU) and space (4TB for 4 years of use) overheads [30]. A motif can leverage such histories to create alternate representations for how a given piece of state could be derived. A Harmonium-based system could then under storage pressure, for instance, choose to discard history of old files partially or completely.

Chapter 8

Conclusion

Many modern applications put ephemeral data into permanent storage, causing wasted effort while conserving unimportant data. We proposed a system, Harmonium to alleviate this waste and enable applications to scale down their storage footprint when required. Harmonium allows application developers to specify which data is ephemeral and create motifs that detail how to reconstruct the data, if needed.

We evaluated Harmonium in two different scenarios. We created a Harmonium-enabled file system and ran a recorded trace from a grading server. During the run we changed the space available to the system and demonstrated that Harmonium was capable of adjusting its storage footprint. Very low available space caused performance to degrade indicating a tradeoff: Low space usage came at the cost of increased latency in filesystem operations and increased network traffic. Likewise, we created a Harmonium-enabled key-value store on top of which we created a route planning program. In this case the tradeoff for low space usage was increased CPU usage while answering queries in addition to slower responses. In both cases, Harmonium clearly reached its goal of enabling space elasticity.

There are several avenues for future work. Identifying more opportunities for space elasticity and creating motifs to enable them would increase the applicability and usefulness of the system. Allowing files to be only partially contracted, instead of always being either fully contracted or expanded, could greatly increase performance when only accessing portions of a file (eg, the `head` command). Modifying a hypervisor to be Harmonium-aware, could allow easier use of space elasticity. (Currently, an administrator needs to set a policy in Harmonium, wait for it to reach the goal, and only then is the space actually freed and hypervisor settings can be edited.)

Bibliography

[1] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: automatic management of data and computation in datacenters.", in *OSDI*, 2010, pp. 75–88.

[2] 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 *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 29–42.

[3] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication", *ACM Transactions on Storage (TOS)*, vol. 7, no. 4, p. 14, 2012.

[4] 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*, ser. USENIX'09, San Diego, California: USENIX Association, 2009, pp. 8–8. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1855807.1855815`.

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

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

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

[8] A. Kulik and H. Shachnai, "There is no eptas for two-dimensional knapsack", *Information Processing Letters*, vol. 110, no. 16, pp. 707–710, 2010.

[9] F. J. Corbato, "A paging experiment with the multics system", DTIC Document, Tech. Rep., 1968.

[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*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 167–181, ISBN: 978-1-4503-2388-8. DOI: `10.1145/2517349.2522722`.

[11] M. Mitzenmacher, "Dynamic models for file sizes and double pareto distributions", *Internet Mathematics*, vol. 1, no. 3, pp. 305–333, 2004.

[12] 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 *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.*, B. Kobler and P. C. Hariharan, Eds., IEEE, 2004, pp. 193–197. [Online]. Available: http://storageconference.org/nasa/conf2004/Papers/MSST2004-21-Zadok-a.pdf.

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

[14] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential raid: rethinking raid for ssd reliability", *ACM Transactions on Storage (TOS)*, vol. 6, no. 2, p. 4, 2010.

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

[16] Z. Galil and J. Seiferas, "Time-space-optimal string matching (preliminary report)", in *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, ser. STOC '81, Milwaukee, Wisconsin, USA: ACM, 1981, pp. 106–113. DOI: 10.1145/800076.802463. [Online]. Available: http://doi.acm.org/10.1145/800076.802463.

[17] M. Hellman, "A cryptanalytic time-memory trade-off", *IEEE Transactions on Information Theory*, vol. 26, no. 4, pp. 401–406, Jul. 1980, ISSN: 0018-9448. DOI: 10.1109/TIT.1980.1056220.

[18] J. Borst, B. Preneel, J. Vandewalle, and J. V, "On the time-memory tradeoff between exhaustive key search and table precomputation", in *Proc. of the 19th Symposium in Information Theory in the Benelux, WIC*, 1998, pp. 111–118.

[19] 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*, ser. Middleware '14, Bordeaux, France: ACM, 2014, pp. 289–300, ISBN: 978-1-4503-2785-5. DOI: 10.1145/2663165.2663317. [Online]. Available: http://doi.acm.org/10.1145/2663165.2663317.

[20] 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*, ser. SOSP '99, Charleston, South Carolina, USA: ACM, 1999, pp. 110–123, ISBN: 1-58113-140-2. DOI: 10.1145/319151.319159. [Online]. Available: http://doi.acm.org/10.1145/319151.319159.

[21] 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)*, London, Aug. 2013.

[22] P. Christen, "A survey of indexing techniques for scalable record linkage and deduplication", *IEEE Trans. on Knowl. and Data Eng.*, vol. 24, no. 9, pp. 1537–1555, Sep. 2012, ISSN: 1041-4347. DOI: 10.1109/TKDE.2011.127. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2011.127.

[23] 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*, ser. FAST'10, San Jose, California: USENIX Association, 2010, pp. 17–17. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855528.

[24] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières, "Replication, history, and grafting in the Ori file system", in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 151–166, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522721. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522721.

[25] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat, "Cimbiosys: a platform for content-based partial replication", in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09, Boston, Massachusetts: USENIX Association, 2009, pp. 261–276. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558995.

[26] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: understanding personal cloud storage services", in *Proceedings of the 2012 ACM Conference on Internet Measurement Conference*, ser. IMC '12, Boston, Massachusetts, USA: ACM, 2012, pp. 481–494, ISBN: 978-1-4503-1705-4. DOI: 10.1145/2398776.2398827. [Online]. Available: http://doi.acm.org/10.1145/2398776.2398827.

[27] 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 *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, 2014, pp. 243–255.

[28] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: practical power-proportionality for data center storage", in *Proceedings of the sixth European Conference on Computer systems*, 2011, pp. 169–182.

[29] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan, "Robust and flexible power-proportional storage", in *Proceedings of the 1st ACM Symposium on Cloud computing*, 2010, pp. 217–228.

[30] D. Devecsery, M. Chow, X. Dou, J. Flinn, and P. M. Chen, "Eidetic systems", in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO: USENIX Association, Oct. 2014, pp. 525–540, ISBN: 978-1-931971-16-4. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/devecsery.

School of Computer Science
Reykjavík University
Menntavegur 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.ru.is