# Harmonium: Elastic Cloud Storage via File Motifs

Helgi Sigurbjarnarson*, Petur Orri Ragnarsson*, Ymir Vigfusson*, Mahesh Balakrishnan†
*School of Computer Science & CRESS, Reykjavik University
†Microsoft Research

## Abstract

Modern applications expand to fill the space available to them, exploiting local storage to improve performance by caching, prefetching and precomputing data. In virtualized settings, this behavior compromises storage elasticity owing to a rigid contract between the hypervisor and the guest OS: once space is allocated to a virtual disk and used by an application, it cannot be reclaimed by the hypervisor. In this paper, we propose a new guest filesystem called Harmonium that exploits the ephemeral or derivative nature of application data. Each file in Harmonium optionally has a *motif* that describes how the file can be reconstructed via computation, network accesses, or operations on other files. Harmonium expands files from their motifs when space is available, and contracts them back to their motifs when it is scarce. Given a target size, the system selects files to expand or contract based on the load on the CPU, network, and storage, as well as expected access patterns. As a result, Harmonium enables elastic cloud storage, allowing the hypervisor to dynamically balance storage across multiple VMs.

## 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) [7]. However, storage elasticity is more challenging; cloud providers cannot easily allocate extra capacity or reclaim it from applications.

In this paper, we focus on storage elasticity in infrastructure-as-a-service clouds, where the hypervisor on a single physical machine partitions its resources across many virtual machines. In such a setting, each virtual machine is provided with a virtual disk of fixed capacity, which in turn resides as a variable-sized file in the filesystem of the hypervisor. As the virtual machine stores data in its virtual disk, its backing file expands. However, the virtual disk cannot expand beyond its fixed capacity; and once the virtual machine stores data in it, the physical machine cannot reclaim space from the virtual disk without destroying the virtual machine. As a result, the physical machine cannot respond elastically to changes in the storage use patterns of its virtual machines.

A key reason for the inelasticity of storage is that 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.

However, modern applications increasingly store data on durable media for reasons other than durability. This shift is primarily driven by hardware trends: larger disks have pushed application designers to think of creative ways to use excess storage capacity to improve performance, while the emergence of flash has provided 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 [3], image thumbnails, desktop search indices, and inflated versions of compressed files), or fetched over the network from other systems (e.g., browser and package management caches). 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 [4]. On three typical developer machines (Table 1), we found 19-28% of the space occupied by various caches (on one machine, only 41% of this space was accessed in the past month), 5-23% by media that also existed on remote servers, 0-2% by source files with primary copies in online repositories, and 0-4% by swap files. While this is preliminary data, we believe the make-up to be fairly representative of modern filesystems.

As a result, virtualized systems exhibit an inefficient dynamic: applications opportunistically use persistent storage to store data that is ephemeral, whereas storage

**Table 1:** *Breakdown of file types on three developer filesystems.*

| | Space Usage | | |
|---|---|---|---|
| **File Type** | System 1 | System 2 | System 3 |
| ASCII | 2% | 21% | 12% |
| Cache | 28% | 19% | 25% |
| Document | 2% | 2% | 7% |
| Media | 23% | 5% | 10% |
| Source | 1% | 0% | 2% |
| System | 6% | 18% | 13% |
| Swap | 3% | 4% | 0% |
| Other | 28% | 31% | 25% |
| Free space | 5% | 1% | 6% |

systems struggle heroically to ensure that this data is not lost. The situation is exacerbated by the address space abstraction, which does not allow the host system to differentiate between the maximum possible size of a virtual disk and the physical storage capacity currently available to it.

To eliminate this inefficiency, we present Harmonium, a new guest filesystem for virtual machines that treats durability as a spectrum rather than a binary property. In Harmonium, each file is associated with an optional *motif* which tells the filesystem how the file can be reconstructed. The motif for a file is an arbitrary piece of code: for example, it might fetch data over the network from a URL, or generate the file via computation over other files (e.g., sort a file, merge multiple input files, or even expand a compressed input file), or reconstitute the file from duplicate chunks in other files. A file can be *expanded* from its motif or *contracted* back into its motif.

Figure 1 shows the Harmonium architecture. A typical deployment consists of a single physical machine running multiple VMs, each of which runs an independent Harmonium filesystem. A single, administrator-controlled process on the host machine acts as a coordinator and is responsible for assigning each Harmonium instance a specific target size. Given this target size and a set of file motifs, Harmonium decides which files to expand or contract based on access patterns as well as the load on the CPU and network. For instance, if the system usually exhibits high CPU load and low network utilization, Harmonium will retain files with CPU-heavy motifs in expanded form and contract files with network-heavy motifs. This ensures that a subsequent access to a contracted file (and the resulting expansion from the motif) is more likely to utilize network bandwidth rather than CPU cycles.

Existing work on elastic cloud storage provide similar trade-offs between storage footprint and performance, usually in a distributed setting. Systems such as Sierra [6], Rabbit [2], and SpringFS [9] rely on techniques such as variable replication factors and write offloading to scale

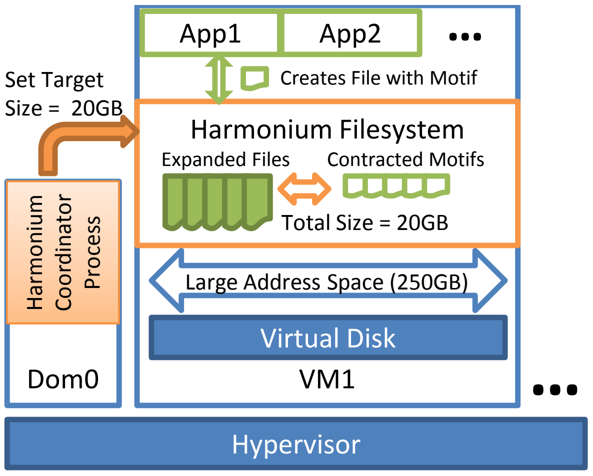


**Figure 1:** *A Harmonium instance within each VM expands and contracts motifs in response to a target allocation set by a coordinator.*

up the performance or scale down the power consumption of distributed storage clusters. However, these systems typically maintain $1$ to $N$ copies of each file, while Harmonium chooses between storing $0$ or $1$ copy of each file.

In the remainder of this paper, we describe the motif abstraction and our initial system prototype, and outline the challenges in realizing a full-fledged implementation.

## 2 The Motif Abstraction

When an application creates a new Harmonium file, it optionally provides a motif. The motif is an arbitrary code fragment that provides an *expand* method to generate the data in the file. Harmonium can then obtain the raw bytes of the file by running the motif's expand method. A motif expansion can fetch data across the network, run computations, and operate over other files in the filesystem.

In our current prototype, Harmonium is implemented as a FUSE filesystem [5] running in user-space, and motifs are fragments of Python code. Figure 2 shows an example motif that fetches data from a remote URL.
Motifs have a number of significant properties:

**Motifs can be recursive.** A motif's expand method can open other files and read from them. These other files could themselves exist in contracted state as motifs. In this case, Harmonium recursively expands the required files from their motifs.

**Motifs can support writeable files.** A motif can optionally contain a *contract* routine. For read-only files, contraction requires no extra code; it merely involves deleting the raw bytes of the file and retaining the motif code. However, in some cases, an expanded file can be modified by the application, and these changes have to be

```
import os

class SCPMotif(object):
  def expand(self, fname, meta=None):
    p = fname.bypass()
    os.popen('scp \
       fileserver1:storage%s "%s"' % (p,p))

  def contract(self, fname, meta=None):
    p = fname.bypass()
    os.popen('ssh fileserver1\
      "mkdir -p storage%s"' %\
      os.path.dirname(p))
    if os.popen('scp "%s"\
       fileserver1:storage%s' % (p,p)) == 0:
      open(p, 'w').close()
```

**Figure 2:** *An example motif: expand fetches data from a remote location, while contract writes it back to the remote location.*

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 default motif is compression.** For conventional files that truly require durability and cannot be fetched over the network or generated via computation over other files, Harmonium uses a default compression motif. In this case, the contract routine compresses data, and the expand routine decompresses it. Compression can be viewed as a special case of generating data via computation over other input files; in this case, the input file is simply the compressed copy of the data.

**Motifs can define circular dependencies.** Compression adds an additional wrinkle to the system: it is wasteful to retain both the compressed and uncompressed versions of a file at the same time. To optimize for space, Harmonium creates a circular dependency between the two versions: the expand routine of the compressed version decompresses it to create the raw file, while the contract routine of the raw file compresses it to create the compressed version. Circular dependencies can save space for other types of motifs: for example, if data is stored redundantly in multiple files depending on the index by which it is sorted, only one of these files needs to exist in expanded form at any given point in time.

**Files can have multiple motifs.** In cases where a file can be reconstructed via more than one method, the application can associate multiple motifs with it. Harmonium then picks the best motif to use for expanding the file. Compression is an obvious alternative motif for any file; if the network is heavily oversubscribed, it might be faster to contract/expand the file via compression/decompression than to access a remote network location.

## 3 Implementation Challenges

We have implemented the basic motif abstraction as a FUSE filesystem, minus dependencies on other files. Going from our simple prototype to a realistic implementation requires tackling a number of challenges:

**Interfaces:** Harmonium requires some modification to the interface between the hypervisor and the guest filesystem. In one direction, each Harmonium instance needs a way for the coordinator process on the host to signal to it the amount of space it is required to use, independent of the size of the virtual disk address space; this can be done via some simple communication channel such as a socket. In the reverse direction, it needs a way to efficiently tell the hypervisor which addresses in the virtual disk address space are no longer in use (i.e., a TRIM command). Such interfaces are increasingly common on cloud platforms; for instance, Windows Azure allows the guest OS to issue TRIM commands to the underlying virtual disk when an application explicitly deletes a file, allowing the hypervisor to reclaim the storage allocated to that file [8].

**Security:** Since a motif is an arbitrary piece of code, applications can cause the system to hang, crash, corrupt data, and consume resources wastefully. Our current FUSE implementation is particularly vulnerable, since the motifs execute within the same process as the filesystem. Executing the motif within the process that created the associated file is not an option, since files typically outlast their creating processes. The coarse-grained isolation provided by virtualization is helpful but too heavy-handed. Required is a lightweight sandboxing mechanism for executing individual motifs that can guard against rogue motifs.

**Access Latency Prediction:** To decide which motifs to expand or contract, Harmonium needs to predict the future. In particular, it needs to know two pieces of information for each file: when it will be accessed next, and how long expansion will take at that point in time. For the first question, Harmonium is no better or worse off than any caching scheme in existence, and can use similar techniques (such as LRU policies) for prediction. Answering the second question accurately is harder, since it requires Harmonium to predict the load on the system at the expected time of the access, as well as understand how that impacts the execution time of the motif. In our current implementation, we make the simplifying assumption that the load profile of the system is relatively stationary over time, and generate execution time estimates for the motifs by executing them proactively and taking black box measurements. These measurements are plugged into an optimization framework, which we describe next.

## 4 Expansion/Contraction Strategies

At the heart of Harmonium is an optimization question: *what files should be contracted (or expanded) when the target size of the filesystem is reduced (or increased)?* The problem is somewhat analogous to the well-known cache replacement challenge of evicting (i.e., contracting) files that are least likely to be used in the near future. However, the issue is further complicated by the additional dimensions of latency and space utilization. When a contracted file is expanded on-demand to accommodate an access, the latency depends on the execution time of the motif. Further, if no motif exists for a file, the default contraction mode of compression can result in different space savings for different files, depending on the compressibility of each file's contents. Our optimization framework does not currently model motifs that depend on other contracted files.

Before we discuss the heuristics used in the Harmonium optimization algorithms, we begin by defining the problem domain. Each Harmonium instance exposes an administrative API that allows an administrator to set the target size of the system: specifically, `contract(`$S$`)` and `expand(`$S$`)` denote that $S$ bytes of space must be released or may now be consumed by the file system.

Let $F$ denote the set of files existing in expanded form in the Harmonium filesystem, and $M$ denote the set of files existing as contracted motifs. Each file $i \in F$ can be contracted from its expanded form and moved to $M$ for expected disk space savings of $s_i$ bytes. Later, the file can be re-expanded according to its motif in $e_i$ expected time, which factors in all sources of latency including CPU time, anticipated network accesses and so forth. The expansion latency variable should also explicitly factor in the time until the expansion is likely to take place, providing some notion of discount for items that will not be used for a long time. In addition, when a file has multiple motifs, $e_i$ is set to the minimum expected expansion time across all the motifs.

**Integer program.** We formulate the Harmonium optimization problem of determining what files should be contracted as the following integer program.

$$\min \quad \sum_{i=1}^{n} e_i x_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} s_i x_i \geq S, \tag{2}$$

$$x_i \in \{0,1\} \quad , \quad 1 \leq i \leq n \tag{3}$$

Here, $n = |F|$ is the number of files and $S$ denotes the amount of space that the filesystem needs to shed.

This problem is equivalent to the 0-1 KNAPSACK problem in combinatorial optimization, which is $NP$-complete. In 0-1 KNAPSACK we are given a collection of items that each has an associated profit and weight, and the goal is to find a subset of an item collection that maximizes the total profit for chosen items in the subset without the total weight exceeding the capacity $W$ of the knapsack. By maximizing the expansion latencies $e_i$ of the files *not* chosen by our integer program subject to the standard weight upper bound constraint, the equivalence between our problem and 0-1 KNAPSACK is evident.

A similar integer program exists for the dual problem of determining which motifs to expand when the filesystem's target size is increased, which we omit for brevity.

Harmonium incorporates the estimates for $e_i$ and $s_i$ in approximation algorithms for the underlying optimization problem. We implemented several approaches and compared against a baseline that contracts files in FIFO order until enough space has been reclaimed.

**APX-KNAPSACK.** We adapted the standard pseudo-polynomial time dynamic programming algorithm for 0-1 KNAPSACK for our problem to improve on accuracy in exchange for performance. However, the $O(nW)$ running time where $W = \left(\sum_{i \in F} s_i\right) - S$ is the maximum space for *remaining* files, is prohibitive on systems with a large number of files $n = |F|$, even when we restrict the algorithm to files that have not recently been used. Hence we exclude this algorithm from our evaluation.

**GREEDYSPACE.** A natural approach to the optimization problem is to disregard the expansion delay altogether and greedily contract files that have not been recently used until at least $S$ bytes of space have been recovered. We implemented GREEDYSPACE by repeatedly contracting files from the LRU list of expanded files.

**GREEDYRATIO.** We modified the greedy approach to incorporate the expansion latency. In the approach, we consider a set $A$ of the least recently used $K$ items of the LRU queue, doubling the value of $K$ as required to obtain until at least $S$ bytes of space can be recovered. We sort $A$ by the ratio of $\frac{s_i}{e_i}$ and greedily select high ratio files from $A$ until we reach or exceed the $S$ threshold.

## 5 Evaluation

Our initial prototype of Harmonium is a user-space filesystem written using FUSE [5]. In Figure 3, we show the performance and space footprint of a Harmonium instance as we elastically size it up and down. In this experiment, all files are replicated on a remote storage server; accordingly, all local files have a motif that expands the file by fetching a copy from across the network. We use the latency to access the first byte of a file as a performance metric. Our workload consists of a set of 54,000 patch files applied in chronological order to the Linux kernel source code.

In the graph, we start by allowing the filesystem to op-

**Table 2:** *Comparison of optimization algorithms.*

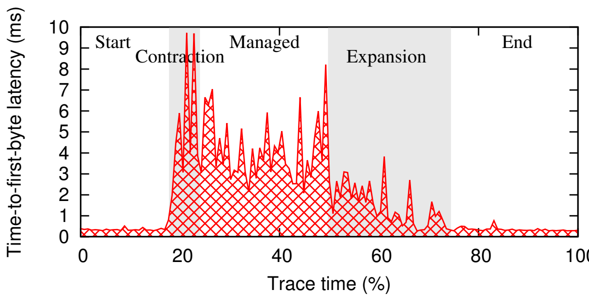
| Algorithm | Expansion latency ($s$) | Running time ($s$) |
|---|---|---|
| FIFO (Baseline) | 866.8 | 0.8 |
| GREEDYSPACE | 807.6 | 2.2 |
| GREEDYRATIO, $K = 10^3$ | 794.9 | 13.0 |
| GREEDYRATIO, $K = 10^4$ | 657.6 | 59.9 |

erate without any size constraints (the segment marked 'start'). We then constrain the filesystem to use less than 450 MB of space ('contraction'). Once contraction finishes, the system enters a stable state ('managed') where the space footprint is stationary and performance is relatively poor. At some point we reset the target size to 650 MB, and the filesystem expands motifs ('expansion'). Finally, we turn off the target size constraint ('end'). When the constraints are off, the filesystem increases in size over time since the trace adds around 200 MB more data than it removes over its lifetime.

**Optimization.** We also used the same workload of patch file applications to test our different optimization algorithms. As input to the optimization framework, we use the access sequence of files, the estimated expansion latency for each file should it be contracted, and the estimated space savings from the contraction of each file. The estimates are generated by measuring an actual execution of the motif. In the experiment, we execute this trace while trying to keep the filesystem size under a 400 MB limit. Whenever the size exceeds the limit, we run the optimization to find files to contract. To measure the efficacy of the algorithm, we keep track of the total expansion latency of contracted files; this is the quantity we need to minimize. Table 2 shows this number for different algorithms, as well as the total time spent executing each algorithm. We found that GREEDYRATIO with $K = 10,000$ reduces the expansion latency significantly, or by nearly 25% compared to the FIFO baseline, at the cost of longer running time for the algorithm. This trade-off is desirable since the expansion latency is on the critical path for user applications whereas the optimization algorithm for contracting space could be executed by a background process as needed.

## 6 Conclusion

Applications have evolved to opportunistically use secondary storage for data that can be recomputed, refetched, or abandoned. However, storage stacks are still designed to provide durability at any cost and treat all files as first-class citizens. In this paper, we proposed a new filesystem called Harmonium where each file is optionally associated with a motif, a piece of code that allows the filesystem to reconstruct the file via computation, network I/O and accesses to other files. In a virtualized system with multiple Harmonium instances running within VMs, administrators can flexibly redistribute physical storage between the instances, providing an elastic performance/capacity trade-off.



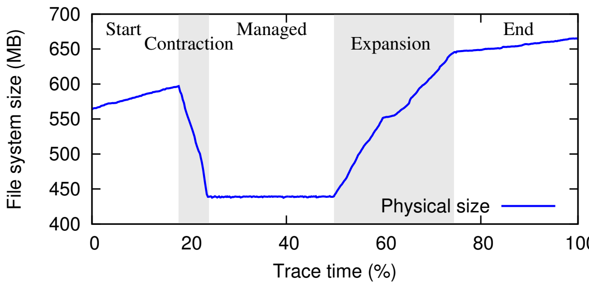


**Figure 3:** *Performance and space footprint of a Harmonium instance as it undergoes contraction and expansion.*

## 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] 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.

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

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

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

[7] 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.

[8] M. Wood. Trim Support comes to Windows Azure Virtual Machines. `http://bit.ly/1kPXicF`, 2013.

[9] 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.