Enabling Space Elasticity in Storage Systems

Helgi Sigurbjarnarson
Pétur Orri Ragnarsson
Junchen Yang
Ymir Vigfusson
Mahesh Balakrishnan

Motivation

Elasticity for CPU and memory well known

Storage use typically hard to decrease

Motivation

00s:

- Single cores
- 1 Gbps networks
- Large HDDs

Motivation

A lot of data is volatile:

Swap files

Constructed from other data (thumbnails, indices, memoized computations)

Fetched over the network (browser and package manager caches)

Case in point: up to 55% of stored data on our dev VMs is ephemeral

Motivation

Today:

- Many cores
- 40 Gbps networks
- Smaller SSDs

Storage systems still promise never to lose data.

Our goal

Create a system that:

- Identifies data that isn’t really needed
- Removes this data when space needs to be recovered
- In case you *do* need some data, recover it

Motif:

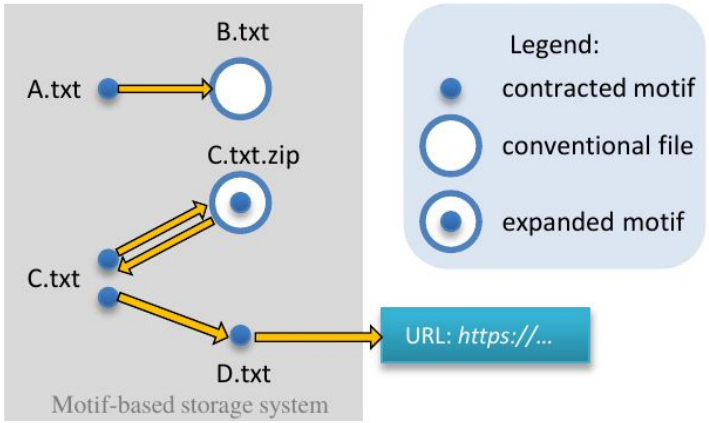
A piece of code that knows how to create a file.

Motifs

More specifically: An *expand* function and metadata

Key properties:

- A motif is stateful
- Motifs can be recursive
- A single file can have multiple motifs
- Can define circular dependencies
- Can be invalidated
- Support writes
  - Optional *contract* function



Carillon:

A system that utilizes motifs to provide space elasticity

Carillon

Two main components: **Runtime** and **storage shim**

Runtime is independent of the underlying storage layer
Shim is tailored to it

Operate in tandem to provide elasticity

Each different storage layer requires its own runtime/shim pair

Design goal: Add elasticity to existing storage with minimal effort

Carillon

The Carillon runtime is responsible for several things

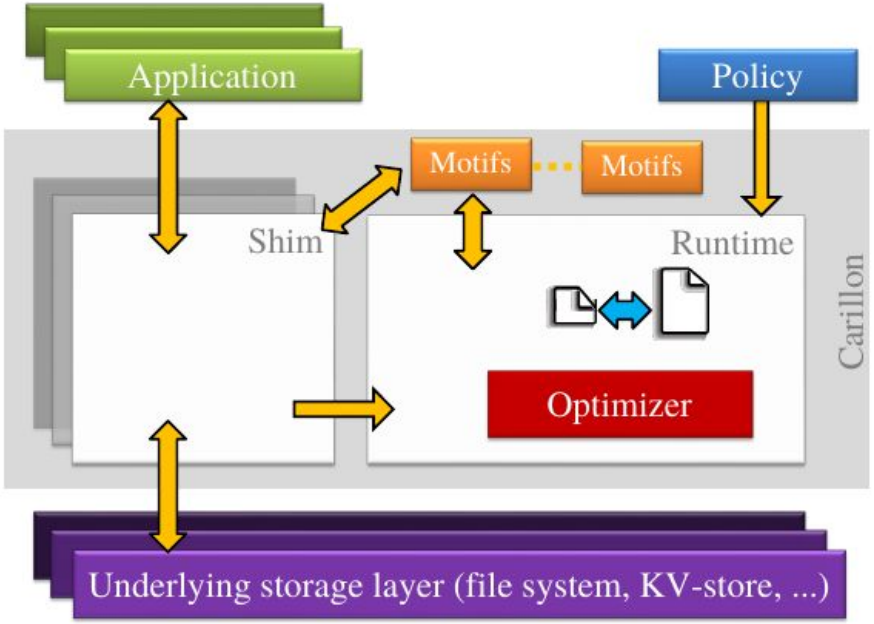
- Managing motif metadata
- Accept storage policies (eg. there is now less space available)
- Track statistics
- Execute motifs based on statistics and available space

Carillon

A Carillon shim, by contrast, does mostly one thing

- Intercept calls to the underlying storage layer and forward to runtime

Overview


Application
Policy
Motifs
Motifs
Shim
Runtime
Carillon
Optimizer
Underlying storage layer (file system, KV-store, ...)

What to delete?

Ideal goal: Never wait for expansion

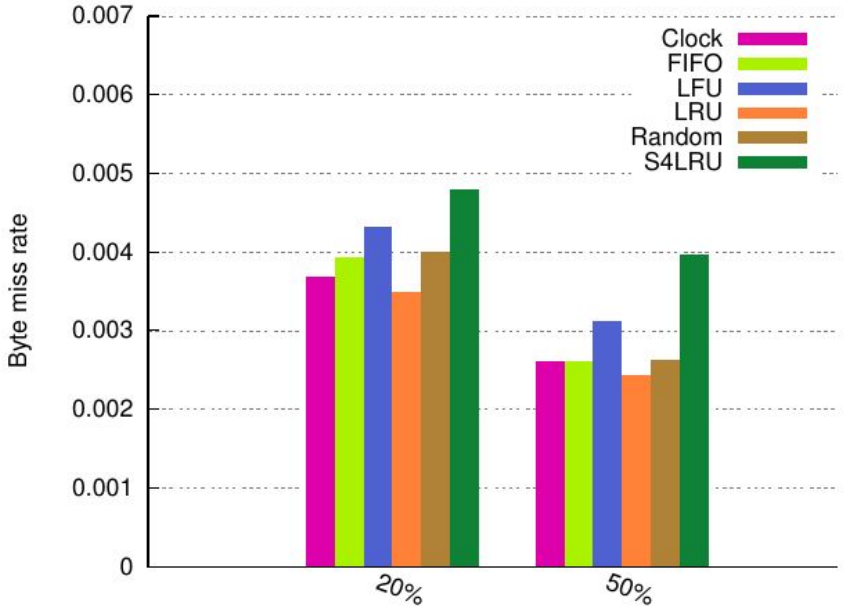
Can’t know the future

Actual goal: Minimize wait time

Model as a 0-1 knapsack problem; slow to solve

Cache algorithms!

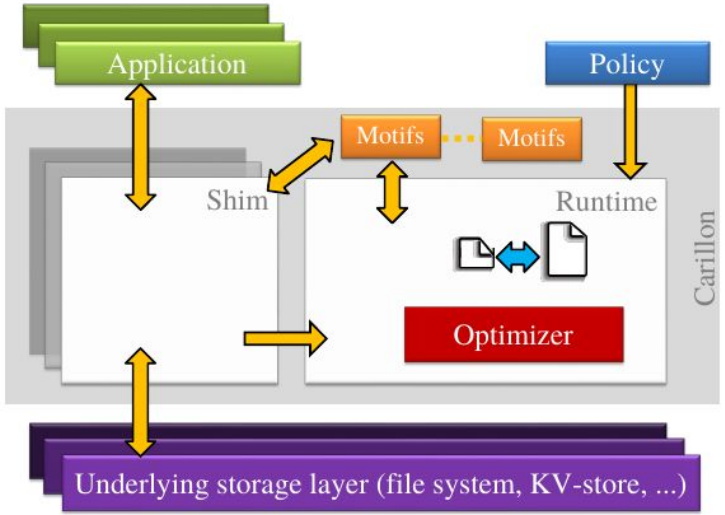
Cache algorithms



CarillonFS

Most operations forwarded without extra work.

Except: stat, open, unlink, rename, truncate, utime



CarillonKV

Key-value store

Graph database

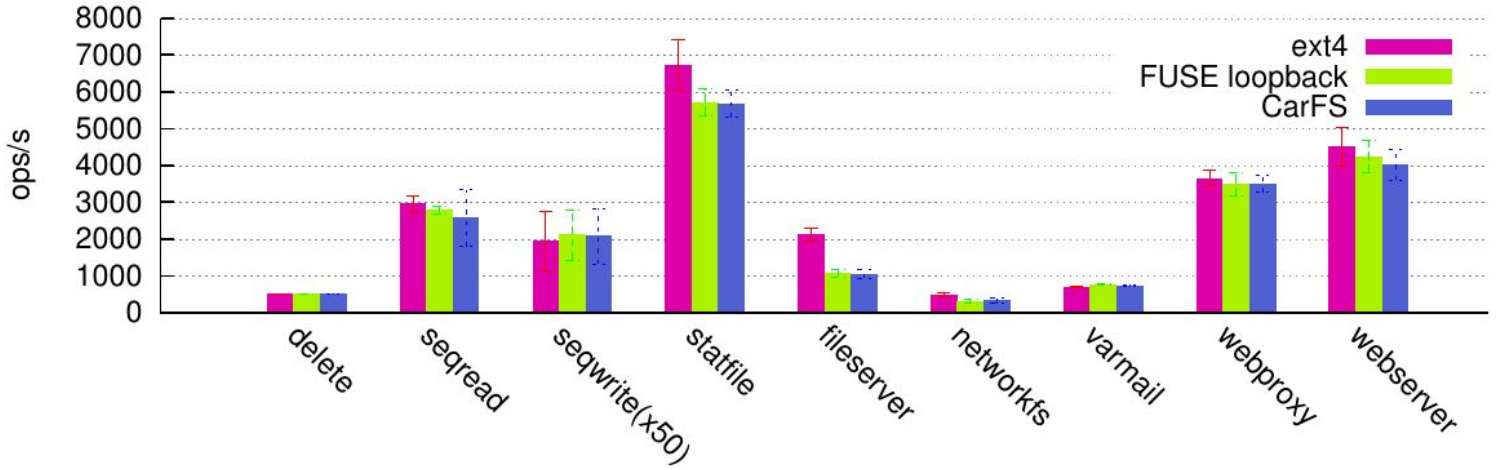
Route planner

Dijkstra’s algorithm has a lot of internal state that’s usually discarded

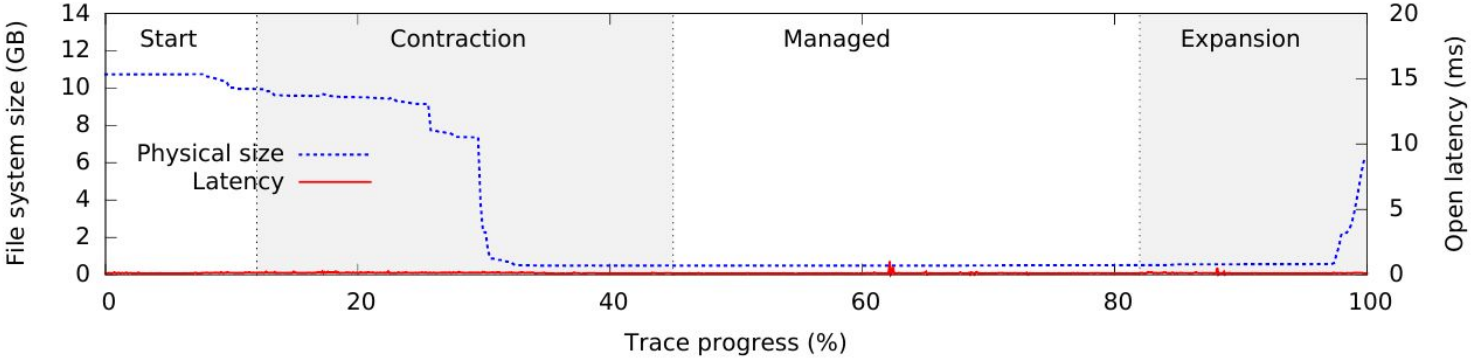
Motif-ize some of it to speed up future runs

Evaluation

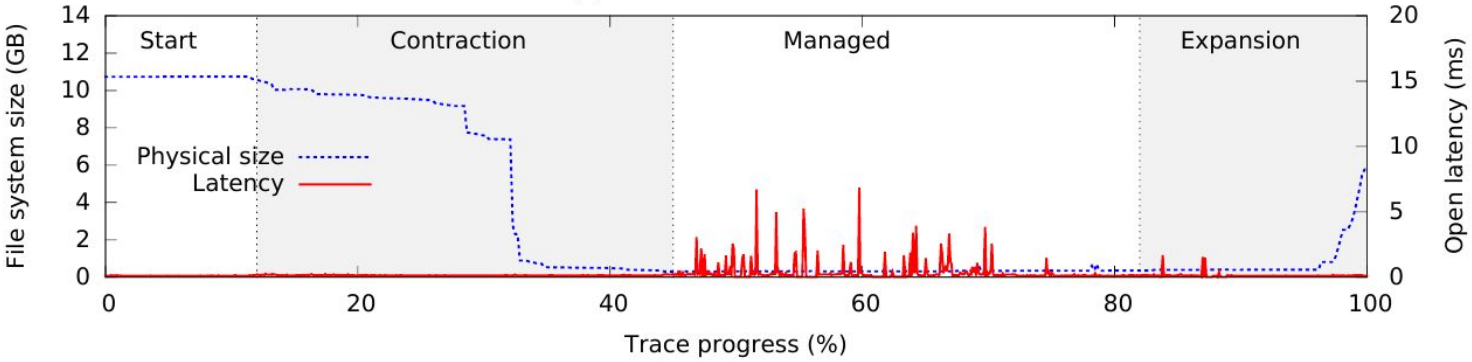
Filebench performance



CarillonFS elasticity

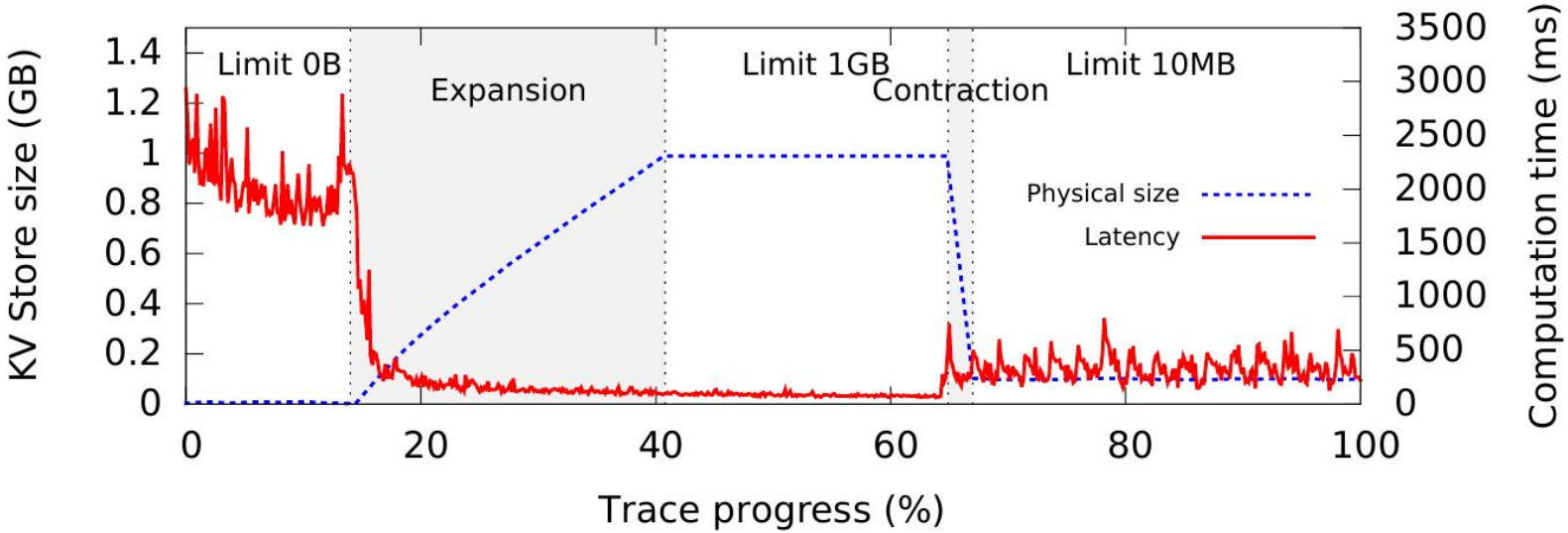


(a) *500MB available to Carillon-FS.*



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

CarillonKV elasticity

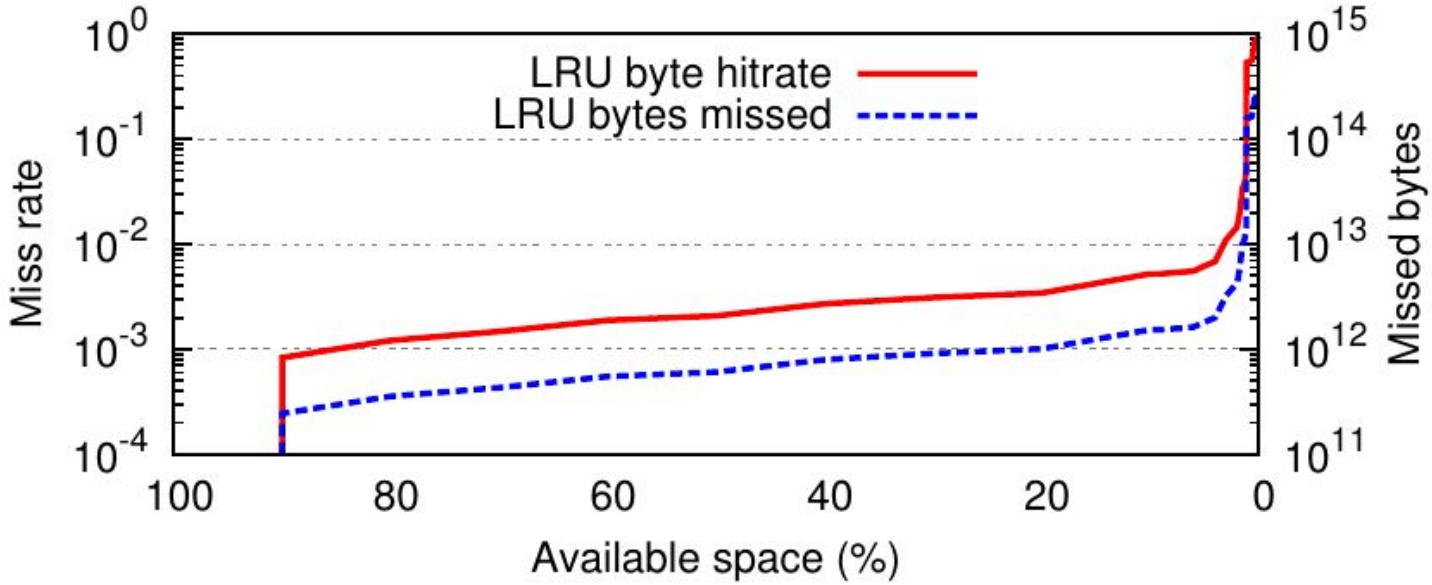


Questions?

Bonus slides!

Highly skewed trace

A vast majority of file accesses happens to a very small subset of files



Example motif

Network storage motif

Contracts a file by copying it to a remote store

Expands by copying back

Very similar to the one used in our evaluations

```
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);
```