Motivation

- Applications require storage systems with
  - Consistent/predictable performance
  - High throughput
  - Low latency
- Example application: Storage for VMs
Hardware options

- Memory (DRAM)
  - Expensive (for primary storage)
  - Often unnecessarily fast (given its cost)
- Hard-drives
  - Too slow & unpredictable (random workloads)
- Solid-state drives (or Flash in general)
  - Balance cost & performance
  - Not quite predictable but fast
Outline

• Performance (un)predictability in SSDs
• Making Flash storage predictable with Rails
• Erasure Coding and Rails
• Conclusion
SSD Performance Unpredictability
Reads are perfect
Writes are unpredictable
Blocking events

For 600ms out of 1sec the device is blocked
Read/write performance

<table>
<thead>
<tr>
<th>Sequential Read</th>
<th>Random Read</th>
<th>Sequential Write</th>
<th>Random Write</th>
</tr>
</thead>
</table>

Only 25% of the reads
Physical read/write separation with Rails
Basic 2 drive design

Drives stay in sync (periodically)
Large storage arrays and distributed storage already employ redundancy (replication or erasure coding)
An object is spread among nodes depending on the redundancy.

Reading nodes accumulate writes in RAM.
Flushing writes

Write nodes flush data to drives

\[ D_{i-4,i-3,i-2} \] flushed during frame \( i \)

Time frame \( i \)

\[ D_{i-4...i}^{(1)} \]
\[ D_{i-3...i}^{(2)} \]
\[ D_{i-2...i}^{(3)} \]
\[ D_{i-1...i}^{(4)} \]
\[ D_{i}^{(5)} \]
\[ D_{i}^{(6)} \]

Time frame \( k = i + 1 \)

\[ D_{i-1...k}^{(1)} \]
\[ D_{i...k}^{(2)} \]
\[ D_{i-2...k}^{(3)} \]
\[ D_{i-1...k}^{(4)} \]
\[ D_{i...k}^{(5)} \]
\[ D_{k}^{(6)} \]

Readers window
Read/Write performance (without Rails)

Only 25% of the reads
Reads mostly unaffected (with Rails)
Blocking events when running traces

Without Rails

With Rails
Rails & erasure coding

• Avoid storage space overhead of replication

• Perform reads through reconstruction (decoding)
  • Utilizes current set of drives dedicated to reading
  • Pay in computational cost

• Scale by constructing redundancy groups
  • Computational cost scalable
  • Maintain read/write separation
Erasure coding

- Write object of size 100MB
- Obfuscate (encode) to 120MB
- Split into 12 chunks of 10MB each
- Distribute across 12 drives
- Any 10 drives/chunks may be used to read the original object
Decoding throughput

[k = m, k chunks unavailable]
Read throughput

Throughput (MB/s)

Read throughput without/with decoding [k=m, k drives unavailable]

Without decoding
With decoding
Redundancy groups with R/W separation

Hypergraph with four overlapping hyperedges (redundancy groups), each containing three vertices (drives)
Generating redundancy groups for R/W separation

Physical drives

$P_i = \text{drives of color } C_i$

Physical drives
Write throughput bound

Total write throughput (as a number of drives)

Number of drives writing
Performance of eRails (erasure coding, 6 drives)

Without eRails

With eRails
Performance of eRails (erasure coding, 10 drives)

Without eR Rails

With eR Rails
Conclusion

• Solid-state drives
  • Fast but lack consistent performance under R/W workloads

• Rails
  • Uses redundancy to physically separate reads from writes
  • Provides read-only throughput and latency (for reads)
  • Performance is at least as good for writes as before

• Rails & Erasure Coding
  • Space-efficient redundancy method for Rails

• Scaling
  • Generate overlapping redundancy groups
  • Proportional increase of computational cost under EC
Throughput in #threads

Decoding throughput [ k = m, k chunks unavailable ]

# Threads

Throughput (MB/s)