
# Lustre File Heat

A quantitative measure of data access frequency

**DataDirect Networks Japan, Inc.**

Li Xi

2015/9/23

# What is Lustre file heat?

- It is a relative attribute of files/objects
- It reflects the access frequency of the files/objects
- Its absolute value does not have precise meaning
- It is used for comparison of files/objects
- It grows as file is accessed
- It dissipates as time moves on

Lower heat
Colder
Less access frequency

Higher heat
Hotter
More access frequency

# Why we need Lustre file heat?

- **Locality characteristics of data access**
  - The main principle to improve performance in hierarchical storage systems
  - In order to improve cache efficiency, many cache algorithms including LRU are designed and used
  - In order to reduce performance degradation, HSM policies needs to avoid archiving hot files
- **Existing methods are not enough for Lustre**
  - Local cache algorithms such as LRU are not able to sort distributed objects
  - Cache algorithms such as LRU are designed for RAM/CPU
  - HSM policies based on timestamps are not accurate enough

# Design of file heat (1)

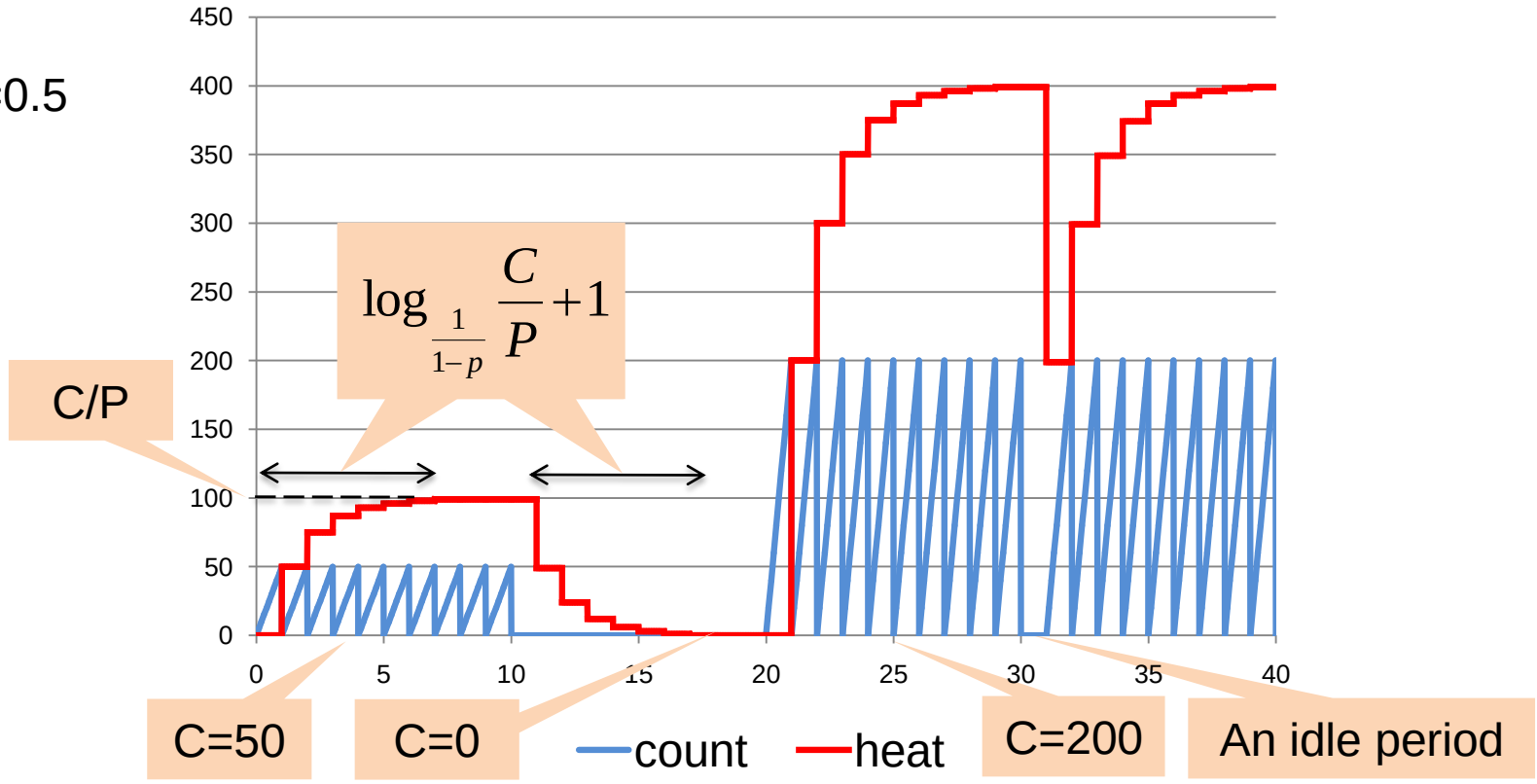
- **Time is divided into periods**
  - 1T, 2T, …, i * T, …
  - The access number during each period is counted
  - The file heat is only recalculated at the end of a time period
  - At the end of each time period, a percentage of the former file heat is lost
- **System-level constants: Period, Loss percentage**
  - It is meaningful to compare heats of files in the whole file system
  - Relative order of heats will be stable for ever if there is no file access
  - It is also meaningful to compare heats of files between file systems if the two arguments are identical

# Design of file heat (2)

- **Recursion formula: H[i+1]=(1-P)H[i]+C[i]**
  - H[i]: Heat value in the period between time points i * T and (i + 1) * T
  - P: Percentage lost every period
  - C[i] : Access count value in the period between time points i * T and (i + 1) * T
- **If C[i] is constant, for example C, then**
  - General formula: $H[i] = (H[0] - \frac{C}{P})(1-P)^i + \frac{C}{P}$
  - H[i] will converge to C/P eventually
  - If H(0) = 0, then i should be at lest $\log_{\frac{1}{1-p}} \frac{C}{P} + 1$ to make sure C/P – H[i] < 1
  - Also, the periods for heat to drop from C/P to 1 is at least $\log_{\frac{1}{1-p}} \frac{C}{P} + 1$

# What does file heat look like?

P=0.5



# How do we tune heat arguments?

- **Period**
  - Larger values for HSM and smaller values for cache
  - Even period of one second won't cause performance issue
  - Reason 1: Extended attribute of heat won't be updated each period
  - Reason 2: The file heat is not actually updated every period, but when the file is accessed or its heat is queried.
- **Loss percentage**
  - If loss percentage is 0, the heat is the access count since the beginning
  - If loss percentage is 1, the heat is the access count during the last period

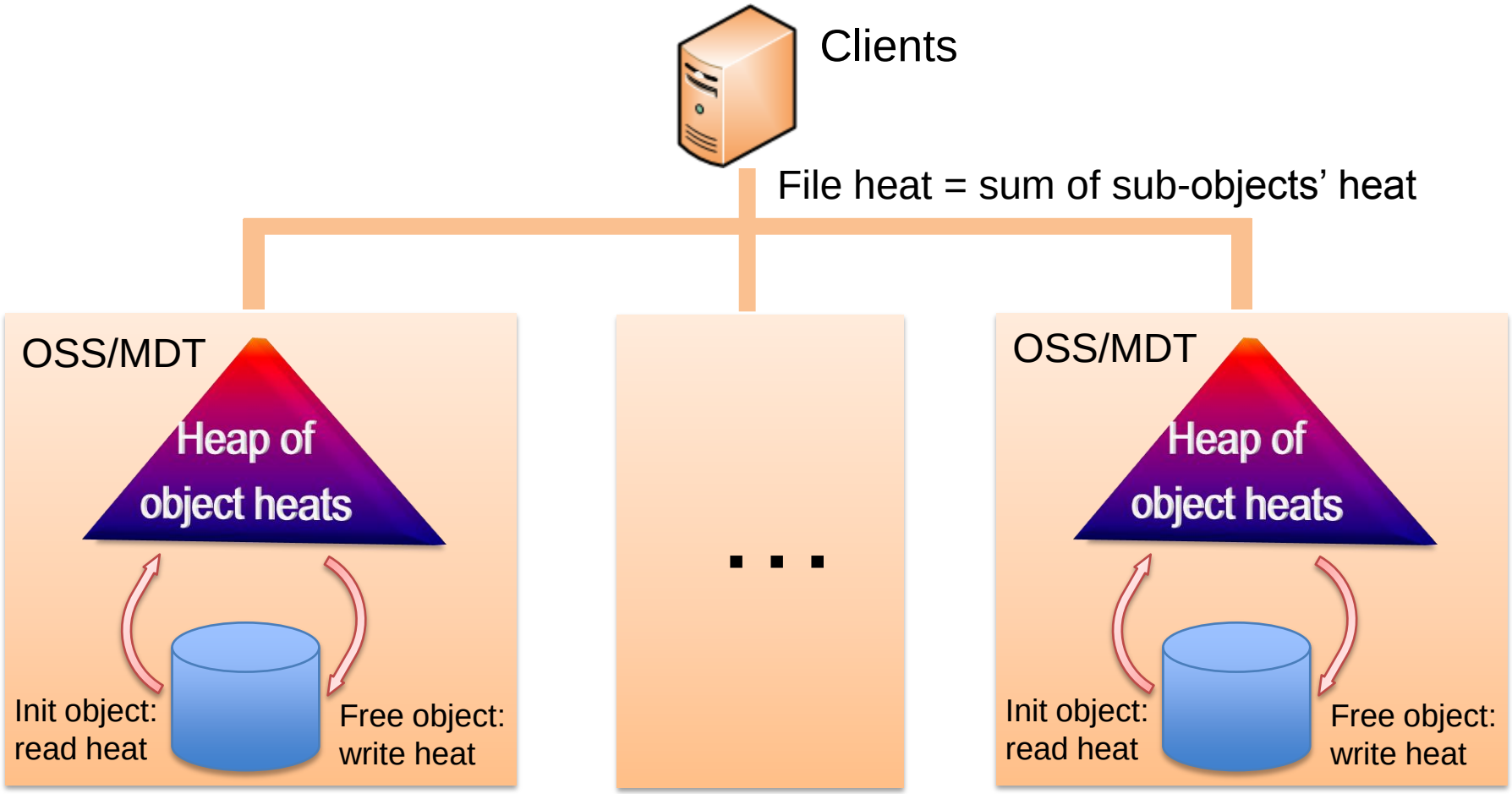
**High loss percentage**
**Smaller heat value**
**More sensitive to change**

**Smaller loss percentage**
**Larger heat value**
**Less sensitive to change**

# Implementation of file heat (1)

- **Lustre file heat = Sum of sub-objects' heat**
  - Sub-objects of a Lustre file includes the metadata object as well as all of the data objects of the file
- **OSTs/MDTs track the heats of sub-objects**
  - Origin object heat is read from storage when object initializing
  - Access counts of objects are increased whenever access happens
  - In the end of every time period, access counts are cleared, and object heat are updated
  - When an object is freed from memory, its heat is written as the xattr of the object
- **The objects can be sorted in heaps of heats**

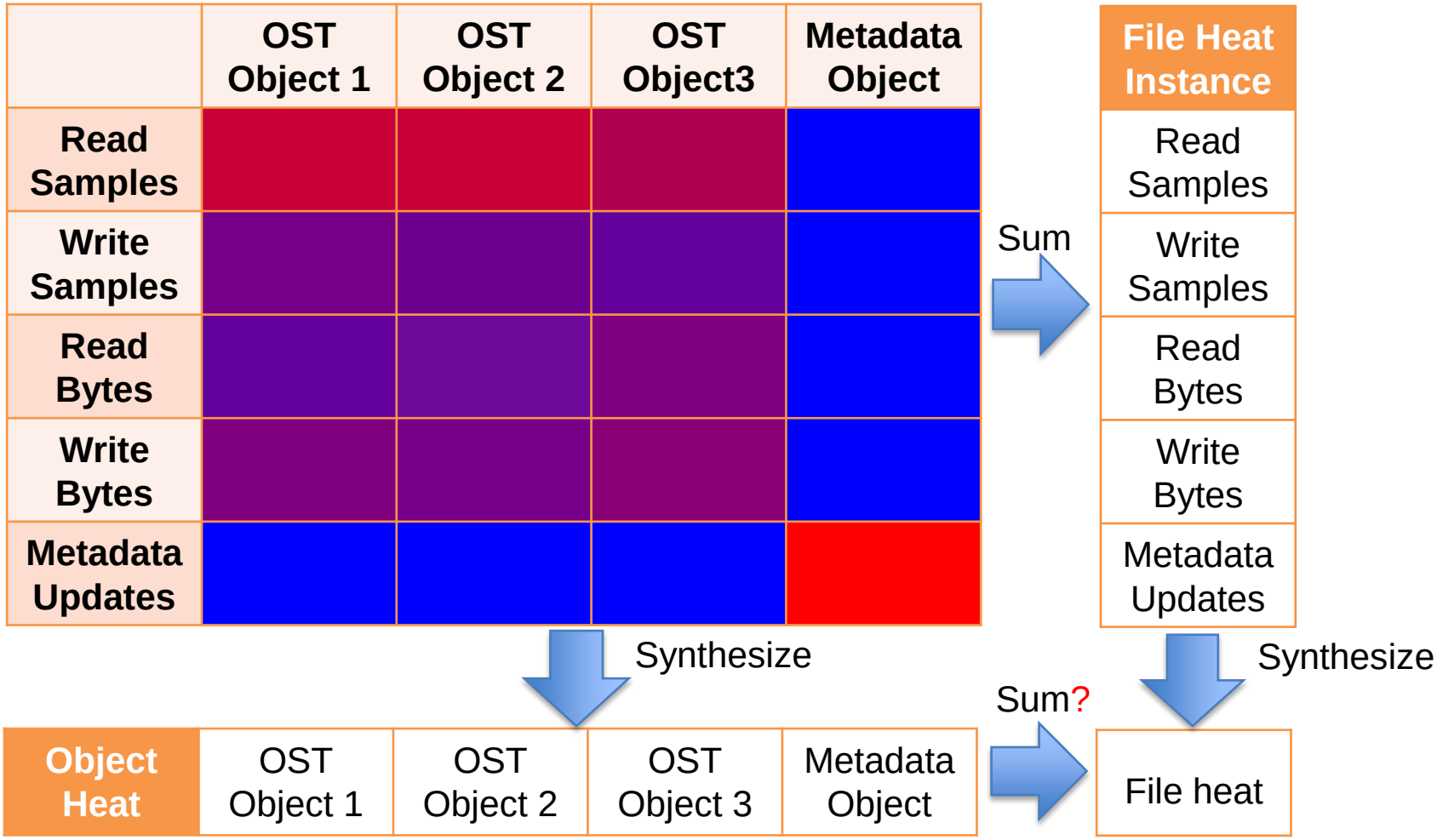
# Implementation of file heat (2)



# Implementation of file heat (3)

- **Multiple heat instances are supported**
  - read_samples, write_samples, read_bytes, write_bytes, metad ata_updates, etc.
  - It does not make sense to sum all numbers that have different dimensions and meanings
- **Heat instances can be synthesized to an aggregative indicator**
  - High read heat + Low write heat = Good to prefetch in read-only cache
  - High read heat + High write heat = Good to prefetch in read&write cache
  - Low read heat + Low write heat = Good to archive to backup system

# Implementation of file heat (4)

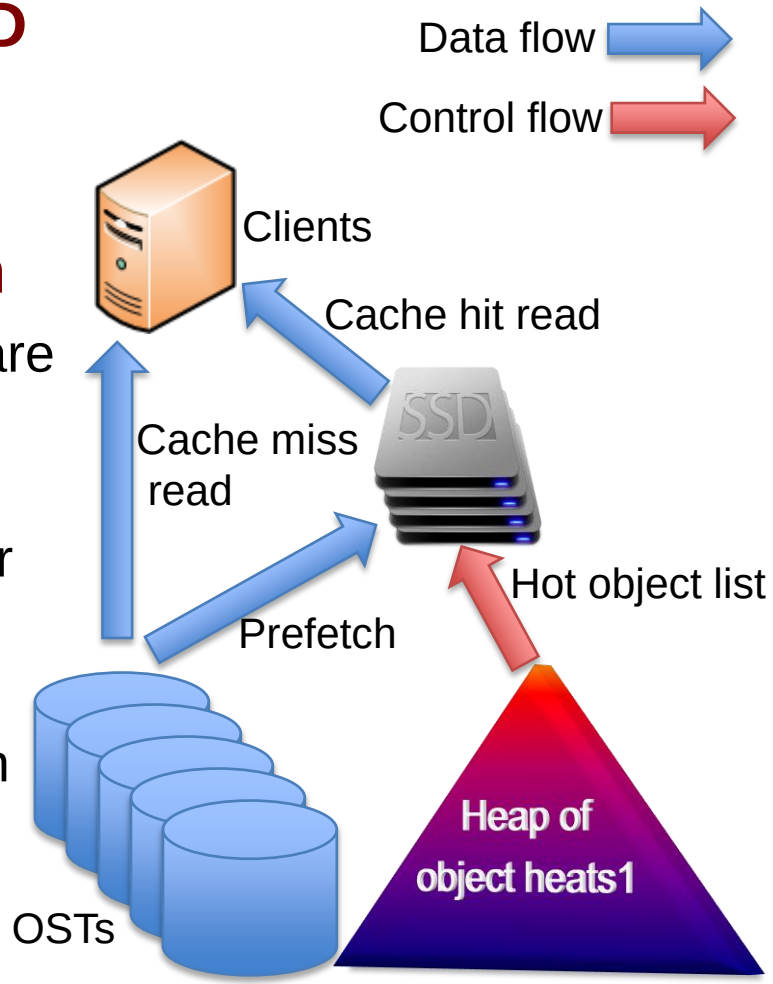


# Use cases of file heat: HSM

- **HSM policies based on file heat**
  - Objects that have lower heats will be on the top of the heaps
  - Coldest objects will be archived preferentially
- **Advantages**
  - More accurate than timestamp based policies
- **Issues:**
  - File heats are not suitable to be replicated in database outside of Lustre, thus Robinhood policies can’t query heats frequently
  - A daemon should be running on MDT to watch the change of the heat heap
  - Total file heat of sub-objects is not easy to track. Making decision based on individual object heat might be enough.

# Use cases of file heat: Cache prefetch

- **An read cache based on SSD is implemented for Lustre**
- **File heat heap is used to decide what data to prefetch**
  - Only objects with high read heats are prefetched
  - Objects with high write heat are avoided to be prefetched even their read heats are high
  - LRU is not suitable since SSD prefetch has much higher cost than memory cache



# Conclusion

- We proposed a new method to reflect the access frequency of Lustre files: file heat
- We analyzed the behavior of file heat so that the values are intelligible and predictable
- We added file heat support for Lustre which introduces negligible performance impact to the system
- We demonstrated that file heat is useful for different use cases including HSM and SSD cache

# Thank you!