Study of cache performance in distributed environment for data processing

Dzmitry Makatun [1 3] Jerome Lauret[2] Michal Sumbera [1]

[1]Nuclear Physics Institute, Academy of Sciences, Czech Republic

[2]Brookhaven National Laboratory, USA

[3]Czech Technical University in Prague, Czech Republic

May 18, 2013

Outline

1. Motivation
2. Simulation setup
3. Access patterns
4. Results of simulation
5. Conclusions

Motivation

The focus of this study is an evaluation of caching algorithms and selection of the most appropriate one for data transfer in HEP/NP computations.

RIFT: **R**easoner for **I**ntelligent **F**ile **T**ransfer

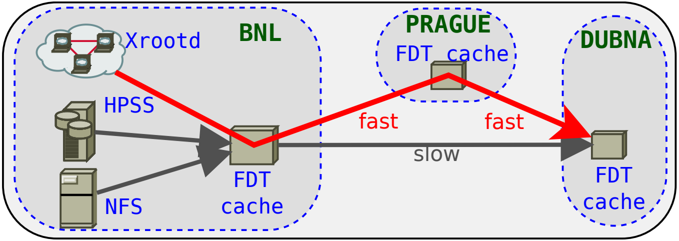
is a software being developed for efficient and controlled movement of replicated datasets within computational Grid to satisfy multiple requests in the shortest time.[a]

[a] Michal Zerola et al "One click dataset transfer: toward efficient coupling of distributed storage resources and CPUs", 2012 J. Phys.: Conf. Ser. 368 012022 doi:10.1088/1742-6596/368/1/012022

How does the RIFT work?

1. Users submit requests for LFNs.
2. RIFT generates an optimal transfer plan.
3. RIFT executes the plan with the help of available data transfer tools.

Case 1: caching at RIFT.



- In RIFT, after transfer copies of files remain at each node on the path. These copies can be used as cache.
- In case of RIFT, the size of cache is small comparing to the size of dataset ($\sim 1\%$)

Case 2: Xrootd

- At present time, all the data of STAR experiment is stored in Xrootd SE.
- It may happen, that the amount of data will exceed the capacity of Xrootd SE.

Possible solution

- Restore data from MSS (HPSS) and put in Xrootd SE upon request.
- Make space when needed by deleting files according to a cache cleanup algorithm.
- In this case the cache size is comparable to the size of the entire dataset.

Problem definition

Two cases:

- Caching for RIFT: small cache (several % of dataset).
- Xrootd as a cache: large cache (up to entire dataset)

Two aspects of caching:

- Reduce makespan of data transfer. (maximize the number of files taken from cache)
- Reduce network load. (maximize the amount of data taken from cache)

For successful cache implementation we need to know

- What is the data access pattern in HEP/NP computations?
- How does the cache performance depend on cache size?
- What caching algorithm is the most efficient?
- How can we measure an importance of a particular file (`file size, time of last access, time of creation, number of access`)?

How to measure cache performance?

Requests

N_{req} - number of requests, S_{req} - data transferred (bytes), S_j - size of file (bytes), $b_j \in \{0, 1\}$ - was file in cache or not.

Storage

N_{set} - number of unique filenames, S_{set} - storage size (bytes), S_i - size of file (bytes), R_i - requests for the file.

Cache hits (minimize overhead due to transfer startup)

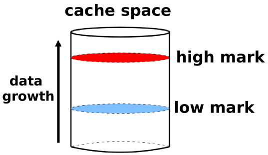
$$H = \frac{\sum_{j=1}^{N_{req}} b_j}{\sum_{i=1}^{N_{set}} (R_i - 1)} = \frac{N_{cache}}{N_{req} - N_{set}} \quad (1)$$

Cache data hits (minimize network traffic)

$$H_d = \frac{\sum_{j=1}^{N_{req}} b_j \times S_j}{\sum_{i=1}^{N_{set}} (R_i - 1) \times S_i} = \frac{S_{cache}}{S_{req} - S_{set}} \quad (2)$$

What system is simulated?

Problem formulation



Parameters

Cache size, low mark, high mark, algorithm (utility function).

Input

Access pattern: log file of user access: `[time, unique filename, size]`

Output

Cache hits, cache data hits.

What is an access pattern?

- User access pattern is data on accessed filenames and access time.
- Defines the use case: it makes sense to evaluate a particular cache algorithm for a particular access pattern.
- Input for simulation `[time, unique filename, size]`.

Random

If the access pattern is completely random, the expected cache hit and cache data hits would be equal to *cache size*/*storage size*.

Access patterns used for simulation

STAR1: RCF@BNL, **Tier-0** for STAR experiment, Xrootd log, user analysis, 3 months period (June-August 2012).

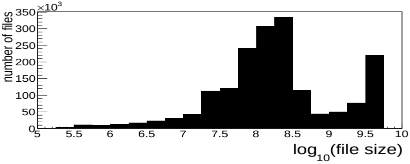
STAR2: RCF@BNL, **Tier-0** for STAR experiment, Xrootd log, user analysis, 7 months period (August 2012 - February 2013).

GOLIAS: FZU Prague, part of **Tier-2** of ATLAS. ATLAS and AUGER experiments, DPM log, user analysis + production, 3 months period (November 2012 - February 2013). AUGER makes less than 1% of total requests.

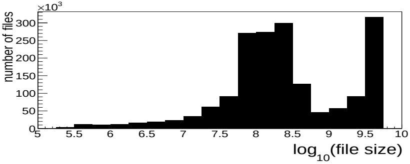
Access patterns summary

		STAR1	STAR2	GOLIAS
Time period	months	3	7	3
Number of requests	$\times 10^6$	33	52	21
Data transferred	*PB*	50	80	10
Maximal number of requests for one file	–	192	203	94260
Average number of requests per file	–	19	15	5
Number of unique files	$\times 10^6$	1.8	1.7	3.8
Total size of dataset	*PB*	1.45	2	1
Maximal file size	*GB*	5.3	5.3	18
Average file size	*GB*	0.8	1	0.3

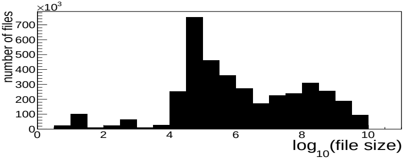
Distribution of files by size



(a) STAR1



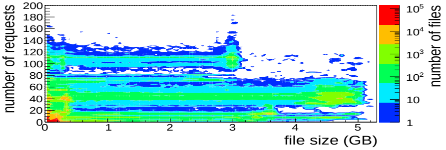
(b) STAR2



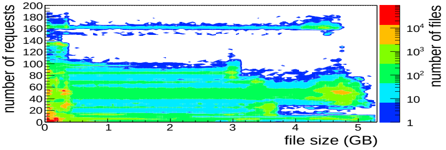
(c) GOLIAS

- At STAR filesize is limited, at GOLIAS it is not.

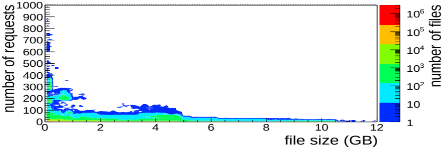
Access patterns as contour plots



(a) STAR1



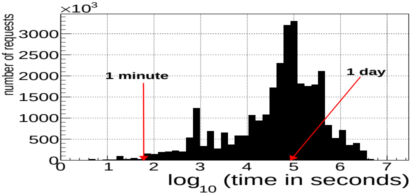
(b) STAR2



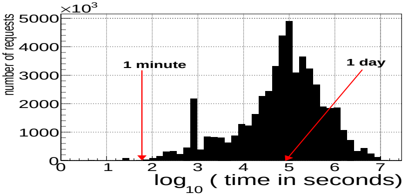
(c) GOLIAS

- Most of the files are small ones accessed several times. • GOLIAS has 2 tails: small files accessed $\sim$ 100 times; large files accessed $\sim$ 10 times. • Looping access patterns are visible.

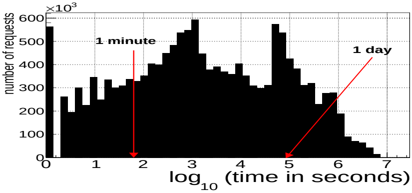
Distribution of time between two requests for the same file



(a) STAR1



(b) STAR2



(c) GOLIAS

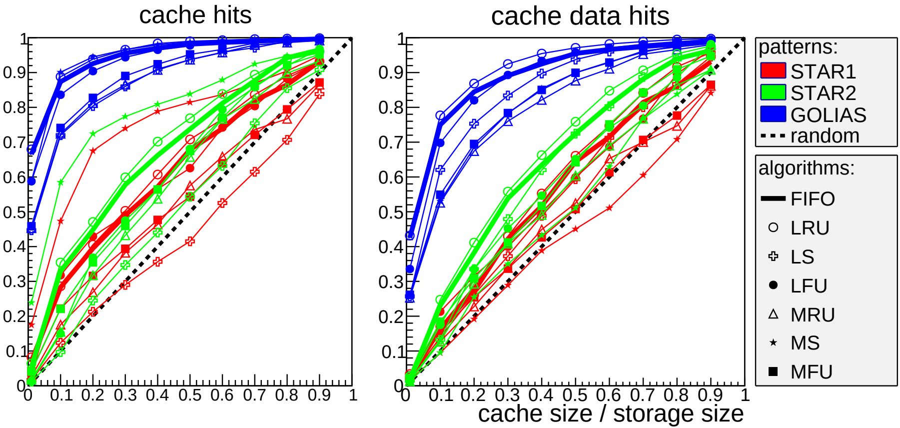
- At STAR the average period is 1 day.
- At GOLIAS the period distribution is less uniform.

What are the canonical caching algorithms?

- – **First-In-First-Out (FIFO)**: evicts files in the same order they entered the cache.
- ○ **Least-Recently-Used (LRU)**: evicts the set of files which were not used for the longest period of time.
- △ **Most-Recently-Used (MRU)**: evicts the set of files which were used most recently.
- • **Least-Frequently-Used (LFU)**: evicts the set of files which were requested less times since they entered the cache.
- ■ **Most-Frequently-Used (MFU)**: evicts the set of files which were requested most times since they entered the cache.
- ★ **Most Size (MS)**: evicts the set of files which have the largest size.
- ✢ **Least Size (LS)**: evicts the set of files which have the smallest size.

Caching algorithms performance: large cache

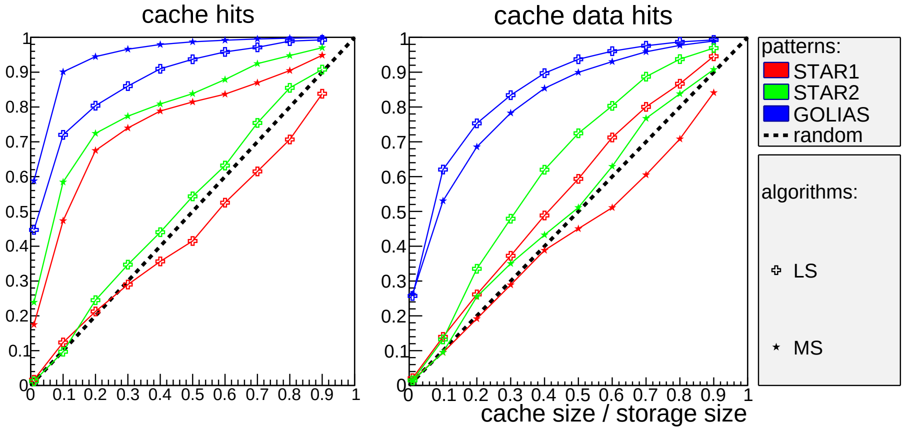
low mark = 0.75 ,high mark = 0.95



- Access patter difference between Tier-2 and Tier-0 leads to distinct cache performance.
- Majority of the algorithms lay above the line of random access pattern estimation.
- The behavior of algorithms is similar within each dataset.

Caching algorithms performance: Most Size vs Least Size

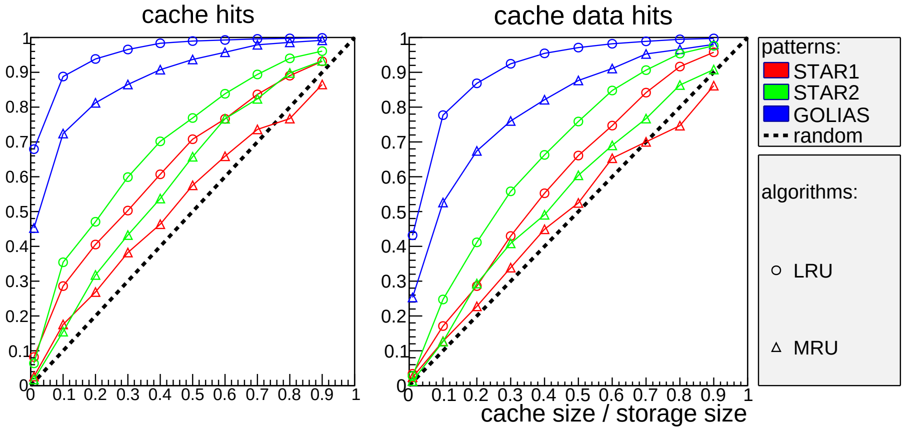
low mark = 0.75 ,high mark = 0.95



•Keeping the smallest files in cache increases cache hits but reduces the cache data hits.

Caching algorithms performance: LRU vs MRU

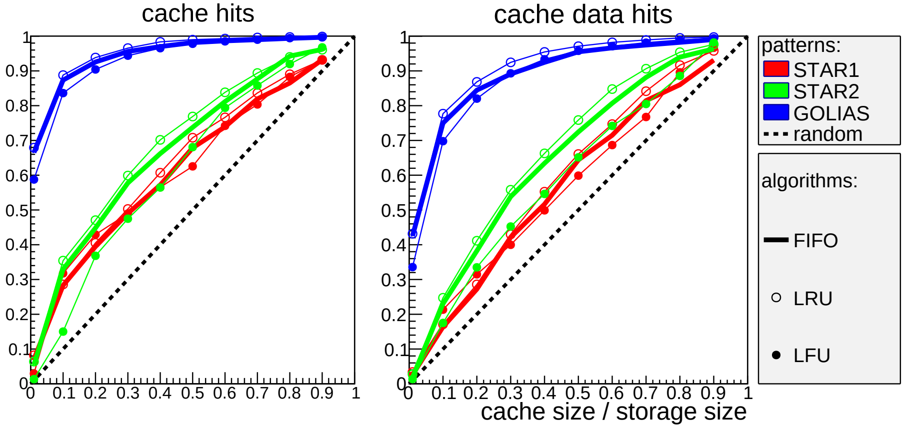
low mark = 0.75 ,high mark = 0.95



•Keeping the most recently accessed files increases both cache hits and cache data hits.

Caching algorithms performance: LFU vs LRU

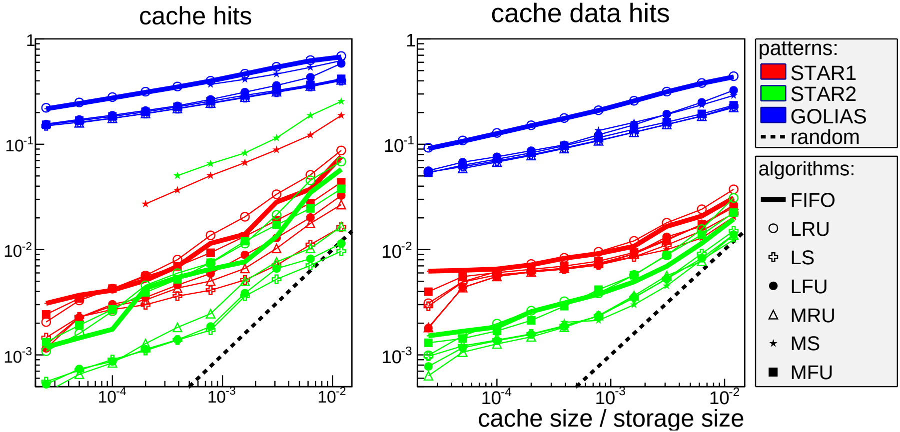
low mark = 0.75 ,high mark = 0.95



- LRU outperforms LFU as well as majority of the canonical algorithms.
- LFU has unstable performance.

Caching algorithms performance: small cache

low mark = 0.75 ,high mark = 0.85



- Tendencies are similar to those for large cache.
- Access patter difference between Tier-2 and Tier-0 leads to distinct cache performance.
- Most of the algorithms lay under FIFO.
- MS - highest cache hits for STAR patterns, but not for GOLIAS.

Are there better algorithms?

Improvements over LRU

2Q, MQ, LIRS, LRU-K, LRFU ...

General idea:

- Split cached files into several lists and treat them separately.
- Use $F(access\ count, times\ of\ last\ k\ requests)$ instead of time of last request.

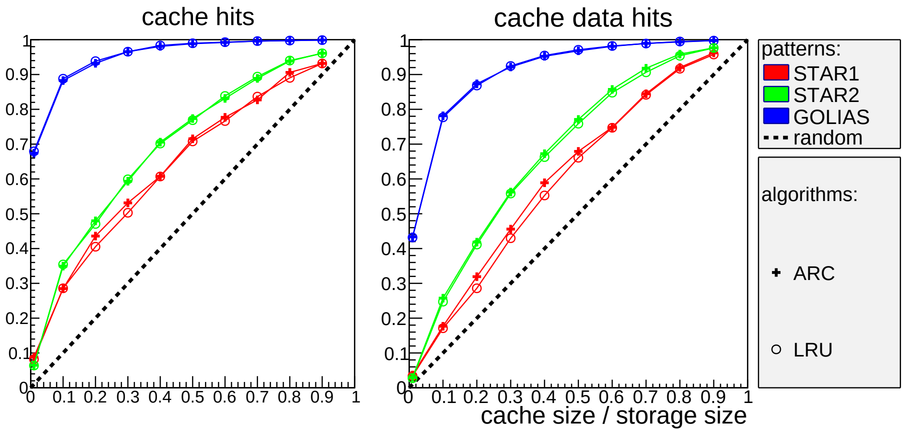
Adaptive **R**eplacement **C**ache (ARC)[a]:

- 2 lists: L1 - files with $access\ count = 1$, and L2 - files with $access\ count > 1$
- LRU is applied to both list.
- Self adjustable parameter $p = cache\ hits\ in\ L1/cache\ hits\ in\ L2$.
- The algorithm defines the number of cached files in each list depending on p.

[a] Megiddo, Nimrod; Modha, D.S., "Outperforming LRU with an Adaptive Replacement Cache algorithm," Computer , vol.37, no.4, pp.58,65, April 2004 doi: 10.1109/MC.2004.1297303

Caching algorithms performance: ARC vs LRU

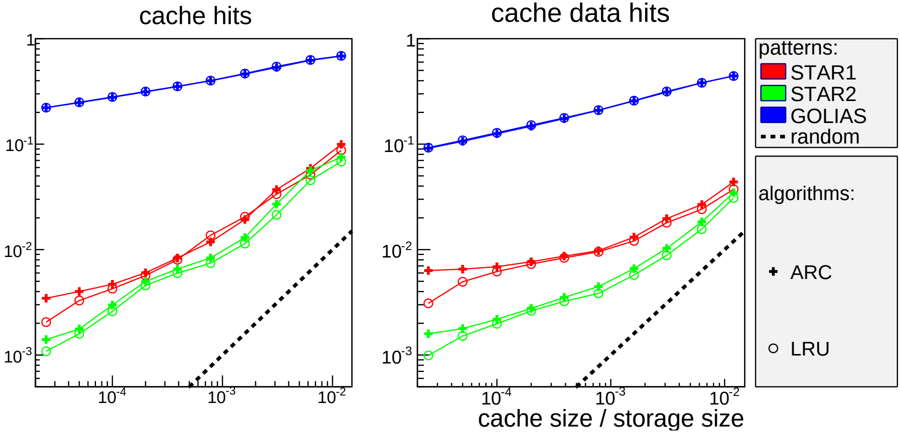
low mark = 0.75 ,high mark = 0.95



- The average improvement with ARC algorithm over LRU is ∼5% for cache hits and ∼7% for cache data hits.

Caching algorithms performance: ARC vs LRU

low mark = 0.75 ,high mark = 0.95



- The average improvement with ARC algorithm over LRU is ∼5% for cache hits and ∼7% for cache data hits.

What other algorithms are known?

algorithms using caching time (CT)

∗Least Value based on Caching Time (LVCT):

Deletes files according to the value of the Utility Function.

$$UtilityFunction = \frac{1}{CachingTime \times FileSize} \quad (3)$$

where **Caching Time** of a file F is the sum of size of all files accessed after the last request for the file F. [a]

[a] Song Jiang, Xiaodong Zhang, "Efficient distributed disk caching in data grid management", 2003. Proceedings. IEEE International Conference on Cluster Computing, 0-7695-2066-9

▽Improved-Least Value based on Caching Time (ILVCT):

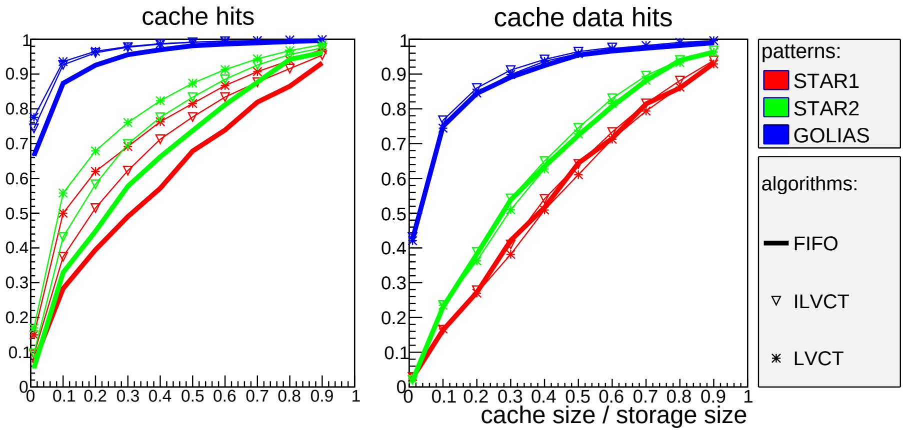
$$UtilityFunction = \frac{1}{NumberOfAccessedFiles \times CachingTime \times FileSize} \quad (4)$$

where **Number Of Accessed Files** is a count of files been requested after the last request for selected file. [a]

[a] J. P. Achara et al,"An improvement in LVCT cache replacement policy for data grid", PoS ACAT **2010**, 044 (2010).POSCI,ACAT2010,044;

Caching algorithms performance: LVCT vs ILVCT (large cache)

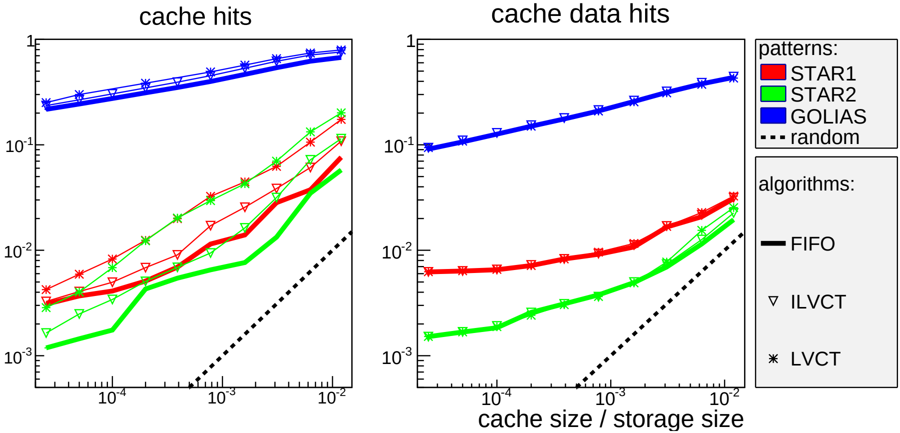
low mark = 0.75 ,high mark = 0.95



- LVCT outperforms ILVCT for studied access patterns.

Caching algorithms performance: LVCT vs ILVCT (small cache)

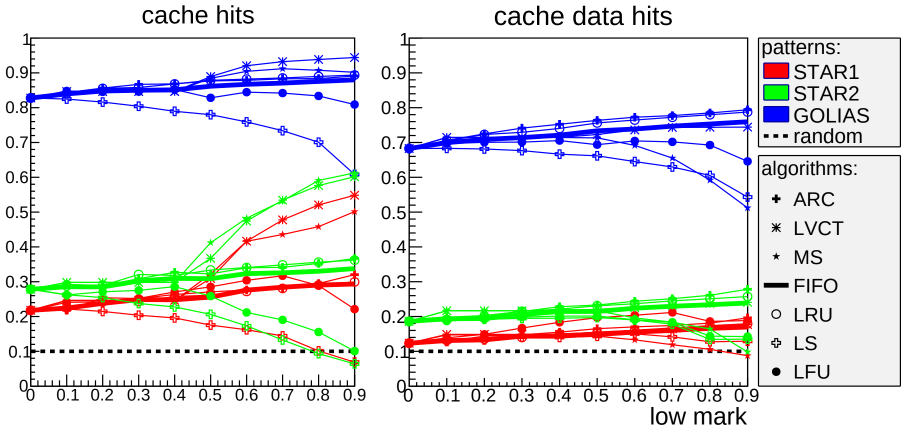
low mark = 0.75 ,high mark = 0.85



- LVCT outperforms ILVCT for studied access patterns.

Dependence of cache performance on low mark

cache size / storage size = 0.1 ,high mark = 0.95



- With higher low mark the number of clean-ups increases.
- Performance of efficient algorithms (FIFO, LRU, ARC and LVCT) increases steadily with the the low mark. For inefficient algorithms (LS, LFU, etc.) decrease is observed.

What caching algorithm is the best?

Average improvement over FIFO

Algorithm	cache hits	cache data hits
MS	116 %	-20 %
LRU	8 %	5 %
ARC	13%	11%
LVCT	86 %	2 %

For studied access patterns

- MS has the best cache hits performance but the worst cache data hits
- ARC has the highest cache data hits
- LVCT balances between cache hits and cache data hits

Conclusions

- Performance of cache algorithms implemented with watermarking concept was simulated for a wide scope of cache size and low marks. 3 access patterns from Tier-0 and Tier-2 sites of 2 different experiments were used as input for simulations.
- Regardless of the cache size, Tier-level and specificity of experiment the LVCT and ARC appear to be the most efficient caching algorithms.
- If the goal is to minimize makespan due to a transfer startup overhead the LVCT algorithm should be selected.
- If the goal is to minimize the network load the ARC algorithm is an option.

thank you for your attention.

backup

How the algorithms were compared:

$$Average\ improvement = \frac{\sum_{i=1}^{n} \frac{value2_i - value1_i}{value1_i}}{n} \quad (5)$$

where:

n - total amount of shared points (with equal parameters) for both algorithms,

i - number of point,

value1 - cache hits or cache data hits of reference algorithm,

value2 - cache hits or cache data hits of compared algorithm.