ADAPTING THE ARC CACHE MANAGEMENT POLICY TO FILE GRANULARITY

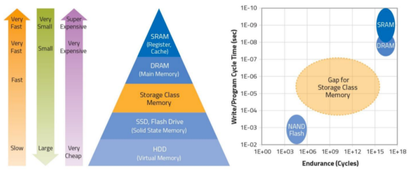
Hocine Mahni, Stéphane Rubini, Jalil Boukhobza, Sebastien Gougeaud, Philippe Deniel

ADAPTING THE ARC CACHE MANAGEMENT POLICY TO FILE GRANULARITY

Hocine MAHNI[1], Stéphane RUBINI [2], Jalil BOUKHOBZA[1]
Sebastien GOUGEAUD[3], Philippe DENIEL[3]
[1] ENSTA Bretagne, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
[2] Univ. Bretagne Occidentale, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
[3]CEA Bruyères-le-Châtel, France

1- HPC Data Placement on Heterogeneous and Multilevel Storage System

- The global data volume will reach 181 zettabytes in 2025.
- Exascale computing may widen the gap between computation, main memory and storage.
- Exploitating multi-tier and heterogeneous storage systems (see table below) is a key to reach trade-off between performance, cost, and capacity.



The new memory hierarchy with SCM[1].

Target Architecture : Heterogeneous three-tier architecture: SSD on the top tier for high performance. HDD as the middle tier, lower performance but higher capacity and lower cost; tape as the bottom tier for archival purposes.

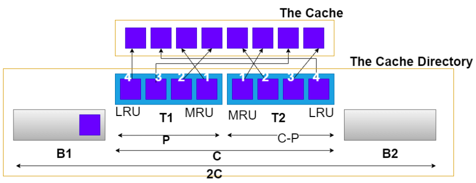
Application: File placement tasks in CEA supercomputers are performed using ***Robinhood***[2], a tool for applying and planning data placement policies. This tool works at the **file granularity**.

Device	Read latency	Write Latency	Write endurance	Cell size (F)^2	Cost
NVM(STT-RAM)	2-35ns	3-50ns	$>10^{15}$	6-50	Highest
NVM(PCM)	20-60ns	20-150ns	$10^8 - 10^9$	4-12	2-8 \$ /GB
SSD	15-35us	200-500us	$10^4 - 10^5$	4-6	0.5-2\$ /GB
HDD	3-5ms	3-5ms	$>10^{15}$	N/A	0.06-0.3\$ /GB

Problem Statement: How to place and migrate data to/from storage tiers according to application QoS.

2- Background on Adaptative Replacement Cache

- Which data to cache in top performance tier can be solved at the operating system level.
- ARC (Adaptive Replacement Cache) is a reference state-of-the-art work [3].



- T1 is an LRU list for pages accessed only once, while T2 keeps items accessed more than once
- C : cache size, P: the current target size for the list T2.
- B1 and B2 are ghost lists used to keep track of the pages evicted by T1 and T2, respectively.

Algorithm 1 Pseudo code of ARC

1: Initialize T1=B1=T1=T2=B2, x: requested page.
2: x in T1 or T2: cache hit, move x to MRU T2.
3: x in B1: cache miss,
Adapt p= min (c,p+ max(|B2|/|B1|,1))
Replace(page), move x to mru T2.
4: x in B2: cache miss,
Adapt p= max (0,p-max(|B1|/|B2|,1))
Replace(page), move x to mru T2.
5: x not in $L1 \cup L2$ *cache miss.*
case 1: |L1|=c :
if |t1|<c then delete the LRU page of B1, Replace(page)
else delete LRU page of T1.
case 2: |L1| <c and |L1| + |L2|>= c:
if |L1|+|L2| = 2c then delete the LRU page of B2.

Replace(page): If either |T1|>p or (|T1|=p and x in B2), replace the LRU page in T1
If either |T1|<p or (|T1|=p and x in B1), replace the LRU page in T2.

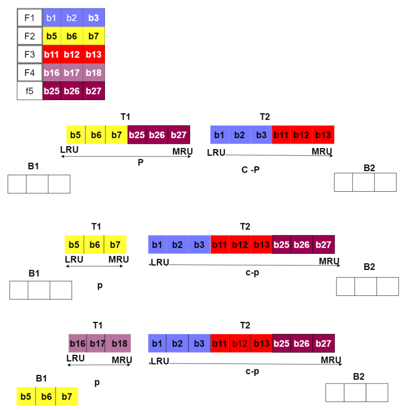
3- Adapting the ARC cache management policy to file granularity

Assumptions: files have the same size + move entire files between T1 and T2

Algorithm 2 Pseudo ARC algorithm with file-level granularity(V1)

1: Initialize T1=B1=T1=T2=B2, f: requested file such as $\forall f$ sf=$\sum_{i=1}^{m} b_i$ and all files's size is equal to sf, block b's size is designated as sb.
2: f in T1 or T2: cache hit, move f to MRU T2.
3: f in B1: cache miss,
Adapt p= min (c,p+ max(|B2|/|B1| *sf/sb, sf/sb))
Replace(file), move f to mru T2.
4: f in B2: cache miss,
Adapt p= max (0,p-max(|B1|/|B2|*sf/sb, sf/sb))
Replace(file), move f to mru T2.
5: f not in $L1 \cup L2$ *cache miss.*
case 1: |L1|=c :
if |t1|<c then delete the LRU file of B1, Replace(file)
else delete LRU file of T1.
case 2: |L1| <c and |L1| + |L2|>= c:
if |L1|+|L2| = 2c then delete the LRU file of B2.

Replace(file): is the same as that of the original ARC algorithm, the only difference being that it replaces a file instead of a page.



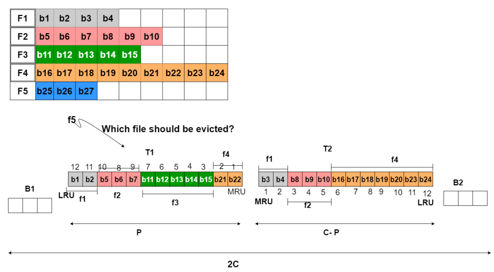
Assumptions: files have different sizes and data are moved between T1 and T2 with a page granularity while data are moved between tiers at a file granularity.

Algorithm 3 Pseudo ARC algorithm with file-level granularity(V2)

1: Initialize T1=B1=T1=T2=B2, b: requested BLOCK, each block is associated with a given file and sf: files's size, block b's size is designated as sb.
2: b in T1 or T2: cache hit, move b to MRU T2.
3: b in B1: cache miss,
Adapt p= min (c,p+ max((|B2|/|B1|)sf/sb, sf/sb))
Replace(file), move b to mru T2.
4: b in B2: cache miss,
Adapt p= max (0,p-max((|B1|/|B2|)sf/sb ,sf/sb))
Replace(file), move b to mru T2.
5: b not in $L1 \cup L2$ *cache miss.*
case 1: |L1|=c :
if |t1|<c then delete the LRU blocks of B1, Replace(file)
else delete file f with highest score.
case 2: |L1| <c and |L1| + |L2|>= c:
if |L1|+|L2| = 2c then delete the LRU page of B2.

Replace(file): To decide which file to delete and replace, we calculate a score that favors evicting files with a high proportion in the LRU portion of T1 and T2 while protecting files that have more blocks in the MRU portion of T1 and T2. Such as:

$$S = \frac{\sum_{bi \in t1 \cap f}(index(bi) + \alpha \sum_{bi \in t2 \cap f}(index(bi))}{cardf}.$$



4- Related work

20 years after its introduction, ARC remains a reference strategy [4][5][6][7][8].
Several studies were based on the principle of using recency and frequency of access to manage caches, such as Lecar[7] and its enhanced version, Cacheus[6]. These approaches maintain two lists, LRU (Least Recently Used) and LFU (Least Frequently Used), and prioritize recency or frequency based on a regret ratio while using machine learning algorithms to select the best strategy.

6- Conclusion and Future Work

We have proposed a version of the ARC algorithm for managing a two-tier (HDD-SSD) storage architecture at the file level. Our strategy is based on striking a balance between the recency and frequency of access to keep recently and frequently used files in the top tier, while preserving the logic of ARC.
For future work: Evaluation of both versions in a multi-tier simulator, including additional parameters to consider for score calculation, such as file lifespan.

References

[1] Mark LaPedus. Next-gen memory ramping up.

[2] Thomas Leibovici. Taking back control of hpc file systems with robinhood policy engine. *arXiv preprint arXiv:1505.01448*, 2015.

[3] Nimrod et. al. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, 2003.

[4] Nimrod et. al. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[5] Santana et. al. To arc or not to arc. In *HotStorage*, pages 14–14, 2015.

[6] Liana et. al. Learning cache replacement with cacheus. In *FAST*, pages 341–354, 2021.

[7] Giuseppe et. al. Driving cache replacement with ml-based lecar. In *HotStorage*, pages 928–936, 2018.

[8] Singh et. al. Adaptive replacement cache policy in named data networking. In *IEEE CONIT*, pages 1–5. IEEE, 2021.