Multi-Size Optional Offline Caching Algorithms

Andrew Choliy[1], Max Whitmore[2], Gruia Calinescu[3] (Advisor)
[1]Rutgers University, [2]Brandeis University, [3]Illinois Institute of Technology

Abstract

The optional offline caching (paging) problem, where all future file requests are known, is a variant of the heavily studied online caching problem. This offline problem has applications in web caching and distributed storage systems. Given a set of unique files with varying sizes, a series of requests for these files, fast cache memory of limited size, and slow main memory, an efficient replacement policy is necessary to decide when it is best to evict some file(s) from the cache in favor of another. It is known that this problem is NP-complete, and few papers have proposed approximation algorithms. We propose three new heuristic algorithms with effective replacement policies, as well as a 4-approximation algorithm. We then evaluate each algorithm by the metrics of runtime complexity and proximity to the optimal solutions of many synthetic data sets.

Formal Definition

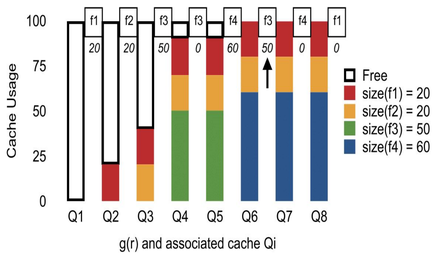
Input
- Finite set *M* of *m* unique files
- For each file *i*, a size *s(i)* ∈ Z+ and a cost *c(i)* = *s(i)*
- Finite list *R* of *n* requests to the unique files
- For each request *r*, *g(r)* denotes the unique file requested
- Initial cache Q_1 of fixed capacity C ∈ Z+. Q_1 may or may not be empty

Output
- A schedule of caches {Q_2, Q_3, … Q_n} where $\sum_{i \in Q_r} s(i) \leq C$.

Cost
- $\sum_{r=1}^{n} \sum_{i \in (g(r) \cup Q_{r+1}) \setminus Q_r} c(i)$

Optimal Replacement Policy
- Load and evict files at each new request to minimize overall fault cost

Feasible Solution: Cache Content and Associated Cost



- *C* = 100
- *cost(g(r))* is shown below each file request
- Each file not present in the cache is loaded except for *f3* at the index denoted by the arrow. It is read from main memory to save cost and later read *f4* for free

Related Work

Forced model with unit-size files and cost
- Belady [3] gives an optimal algorithm

Optional model with multi-size files and cost
- Chrobak et al. [4] show the problem is NP-complete
- Irani [5] gives the first solution as an O(log k)-approximation ratio, where k is the ratio of *C* to the size of the smallest file

Forced model with multi-size files and cost
- Albers et al. [1] provide an O(log(M + C))-approximation algorithm where M and C denote the cache size and the largest file fault cost, respectively. They also provide a reduction to the Loss Minimization problem
- Bar-noy et al. [2] provide a 4-approximation algorithm by reduction to a Loss Minimization problem

Methodology and Results

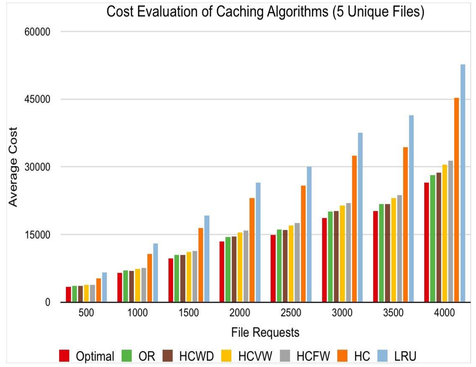
Heuristic Caching from HyCache+
In their algorithm, Zhao et. al [6] define a cost/gain value for each file equal to its size multiplied by the number of remaining requests for that file. The algorithm iterates through the list of file requests. At each step, see if the file is already in the cache. If so, continue. Otherwise, add this file to the cache if there is sufficient free space. If there is not enough space in the cache, the algorithm considers the cost of the files that would need to be evicted and the gain of adding this current file. It does so by first sorting the files in the cache by order of increasing cost. It then considers as many of the lowest cost files as necessary until the amount of free space in the cache from the potential eviction of these files is at least the size of the current file. If the sum of the costs of removing this file(s) exceeds the gain of adding the new file, the cache remains unchanged. Otherwise, those files will be evicted and the new file will be added to the cache.

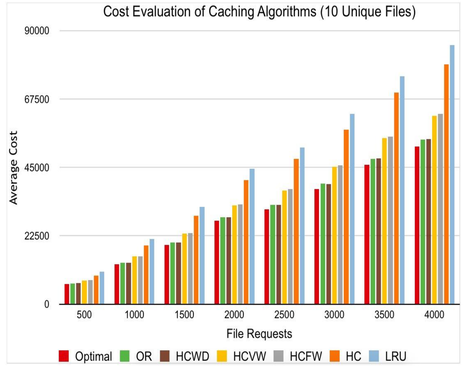
Proposed Algorithms
(1) Heuristic Caching Fixed Window (HCFW)
(2) Heuristic Caching Variable Window (HCVW)
(3) Heuristic Caching Weighted by Distance (HCWD)
(4) Overload Reduction (OR)

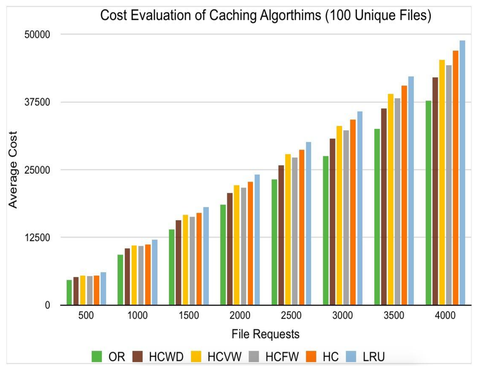
(1-3) are variants of the heuristic caching (HC) algorithm as given by Zhao et al. [6]
(4) is an adaption of Bar-noy et al.'s [2] Loss Minimization framework from the forced model to the optional; guaranteed 4-approximation

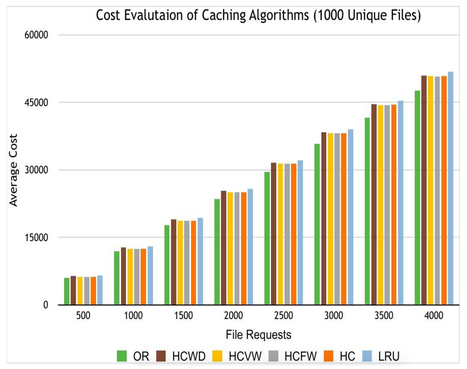
Benchmark
- Well-known LRU algorithm
- Optimal solution (when feasible to calculate)

Testing
- *C* = 100
- Between 500 and 4,000 file requests at increments of 500
- 5, 10, 100, or 1000 unique files
- File sizes range from 21 to 40 (*m* = 5, 10) or 1 to 20 (*m* = 100, 1000)
- 30 random trials for each combination of number of unique files and number of file requests with results averaged









Evaluation

Average Percent of Optimal Cost for 5 Unique Files
- HC: 168.8%
- HCFW: 117.4%
- HCVW: 114.3%
- HCWD: 107.6%
- OR: 107.3%
- LRU: 199.6%

Average Cost Comparison vs HC for 100 Unique Files
- OR: 18.2% improvement
- HCWD: 8.6%
- HCFW: 4.3%
- HCVW: 2.5%

Performance Evaluation:
- HC, HCFW, HCVW, and HCWD all have a runtime complexity of O(*nm* log (*m*)). OR has a runtime complexity of O(n^2).
- For *m* < 1000, HCFW, HCVW, and HCWD significantly reduce average cost while maintaining HC's fast runtime.
- OR usually has the best average cost, but with significantly longer runtime.
- HCWD algorithm strikes a good balance of low cost and fast runtime.
- When unique files is close to number of file requests (*m* = 1000), each algorithm performs similarly, even LRU. This makes sense because it becomes rare that files are repeated often, minimizing the usefulness of caches in the first place.

Conclusion and Future Work

We propose three heuristic algorithms with effective replacement policies, as well as a 4-approximation algorithm.

HCFW, HCVW, and HCWD show significant improvement over the HC algorithm. Although it is much slower than the other algorithms, OR returns the smallest cost on average and is always a 4-approximation even if the cost for reading a file is arbitrary (not related to the file size).

Additional evaluation using real-world workload data would provide greater insight to the usefulness of our algorithms in applications. Future work could examine the performance of our algorithms using multiple caches, distributed at several compute nodes.

Acknowledgements

This work was supported by the National Science Foundation under award NSF - 1461260 (BigDataX REU).

References

[1] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. 1999. Page replacement for general caching problems. In Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms (SODA '99). Society for Industrial and Applied Mathematics, Baltimore Maryland, 31–40.
[2] Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, and Joseph (Seffi) Naor. 2001. A Unified Approach to Approximating Resource Allocation and Scheduling. Association for Computing Machinery (ACM) 48, 5 (2001), 1069–1090.
[3] L. A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-storage Computer. IBM Systems Journal 5, 2 (1966), 78–101.
[4] Marek Chrobak, Gerhard J. Woeginger, Kazuhisa Makino, and Haifeng Xu. 2012. Caching is hard - even in the fault model. Algorithmica 63, 4 (2012), 781–794.
[5] Sandy Irani. 1999. Page Replacement with Multi-Size Pages and Applications to Web Caching. In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing (STOC '97), Vol. 3. ACM, El Paso Texas, 701–710.
[6] Dongfang Zhao, Kan Qiao, and Ioan Raicu. 2015. HyCache+: Towards Scalable High-Performance Caching Middleware for Parallel File Systems. International Journal of Big Data Intelligence (2015).