Data Caching as a Cloud Service

Gregory Chockler
IBM Research Haifa Labs
Haifa, Israel
chockler@il.ibm.com

Guy Laden
IBM Research Haifa Labs
Haifa, Israel
laden@il.ibm.com

Ymir Vigfusson
IBM Research Haifa Labs
Haifa, Israel
ymirv@il.ibm.com

Categories and Subject Descriptors

C.2.4 [**Computer Communication**]: Distributed Systems; B.3.2 [**Memory Structures**]: Cache Memories

Keywords

cache, QoS, partitioning

1. INTRODUCTION

We discuss the challenges of devising a useful shared *data cache* service as a part of the cloud platform. Outside the cloud, such a service appeals to developers for two main reasons. Most importantly, data caches reduce the response latency experienced by users. For example, rendering a content page with various personalized boxes as part of a user web session often involves numerous database lookups. Therefore, if the content generation involves cheap memory accesses to a data cache instead of actual database queries, the user experiences will improve. Moreover, data caches are simple to use: they normally expose a simple get/set interface akin to key/value stores and a rudimentary mechanism to expire values [2], thus allowing result-based caching to be seamlessly integrated with existing database-driven code.

Historically, data caches have been an integral part of database services, including IBM DB2, MySQL and Oracle [12, 1]. More recently, they have become first-class citizens that live on a separate cluster. For instance, sites like Facebook, LiveJournal, Youtube and Wikipedia which experience extremely high rates of web requests use distributed memory object caching services such as *memcached* to alleviate their database load and improve response time [2]. In 2008, Facebook reported to use over 800 memcached servers with a total of 28TB of memory to serve over 200,000 requests per second [16].

For a data cache to be of the greatest convenience inside the cloud, it should be specifically considered as a cloud platform service shared by multiple tenants. The first observation is that without minimum Quality-of-Service (QoS) guarantees for tenants, the service levels may be unacceptable for two reasons. On the one hand, very heavy contention can cause each tenant to receive an insufficient share of the cache to store its data, resulting in fewer cache hits and a poorer experience for the tenant's users. On the other hand, a shared data cache with tenant-oblivious policies, like evicting the *least-frequently-used* (LFU) or *least-recently-used* (LRU) entries, will favor giving space to the tenants with higher request rates, degrading the service quality in terms of cache hits for lower-rate customers who also pay per request. In the extreme, the large online book retail store may simply dwarf the modest yet cacheable workload of the small bookstore on the corner. Enforcing minimum QoS-guarantees for tenants solves these problems and makes the cloud more transparent.

A second observation is that adding tenant-awareness to a data cache can improve the overall hit rate of the cache. This is because more memory can be assigned to those tenants whose workloads have been more *cacheable* (higher locality of reference) instead of those who just produce a higher rate of requests. For instance, a tenant who issues a large range query or scans through a table will contaminate an LRU cache by evicting useful data of other tenants in favor of entries with limited reuse potential.

In this paper, we discuss Blaze: a simple multi-tenant data cache scheme based on the Clock replacement algorithm [6] that uses utility-based cache partitioning between tenants. We designed Blaze with the following goals in mind.

- *Minimum QoS-Guarantees:* Blaze guarantees a minimum cache memory share for each tenant irrespective of the workload of other tenants.
- *Maximize Overall Hit Rate:* Blaze dynamically allocates the remaining cache resources with preference for the tenants that benefit more from extra space in terms of hit rate.
- *Maximize Concurrency:* Blaze strives to minimize lock contention on a cache hit and thus improve concurrency.

We plan for Blaze to become the core component of a shared data cache within a cloud service. Our current focus is on reaching these goals on a single data cache server – in the future, we plan to scale out to multiple nodes by partitioning the key space across servers [2] or by similar techniques. In the following sections, we report on the design of Blaze and our initial experiences while developing the system, which remains a work in progress.

LADIS '10 Zürich, Switzerland

2. PAGE REPLACEMENT ALGORITHM

The caching problem is endowed with an extensive literature; after all caches are ubiquitous, speeding up our CPU computations, disk accesses and web content requests every day. Typically, a cache fetch (or get) involves searching for a key within a hash table and returning the value if one exists. On a cache miss, the actual value corresponding to the key is computed (such as by fetching data from the disk or sending a query to a database) and then inserted back into the cache. At steady state, the insertion will trigger another entry in the cache to be evicted, and the victim is chosen by the page replacement algorithm mentioned earlier. Since caches ultimately have finite memory, the job of the replacement policy in a cache is to place bets on what values are most likely to be requested again in the future. The optimal page replacement algorithm is indeed to evict the entry that will be used at the farthest time in the future, but without the gift of foresight one must settle for an approximate solution.

Variants of the LRU (least-recently-used) algorithm are popular in practice but generally have problems with concurrency. Their appeal stems partly from the simplicity of the LRU approach and the efficient handling of workloads with clustered reference locality, which are commonplace in CPU and storage caches. The typical implementation of LRU maintains a linked list of keys; on a hit the entry is moved to the head, and on a miss the element at the tail is evicted. These operations can be done efficiently for hardware caches, but within the application-layer the complexity of a cache hit is high relative to other replacement schemes. An LRU cache that is updated by multiple threads concurrently will have a bottleneck at the head of the linked list due to lock contention.

CLOCK is a "1-bit" approximation to the LRU algorithm, and works as follows. Each entry in the cache contains a *recently-used* bit denoting whether the entry was recently accessed. On page miss, a *hand* moves through cache entries in a clockwise fashion, switching off the recently-used bit on entries it encounters, and evicts the next entry whose recently-used bit is unset. Although it may seem like a rough approximation to LRU – entries that have not been accessed for a full clock rotation are considered to be equivalent candidates for eviction – CLOCK performs comparably to LRU in practice [17]. Moreover, CLOCK is highly concurrent, doing negligible work on a hit and avoiding bottlenecks on a miss, whereas LRU suffers from contention near the head of the linked list. Many time-critical cache services favor CLOCK or similar variants over pure LRU schemes (see references in [3]). We chose to use CLOCK as the basic page replacement algorithm for BLAZE.

3. QUALITY OF SERVICE

The quality of service should correspond to what the tenant would achieve *in isolation* from other tenants [13, 8, 15]. We remarked earlier that even if the tenant has a highly cacheable workload, it could still observe high miss rates unless the cache is being managed in a tenant-aware fashion. For a cloud tenant who's considering to use a data cache, what constitutes a meaningful quality of service guarantee?

As a first attempt, suppose that the cache memory is divided equally among all tenants. The issue is that some tenants may make inefficient use of their share (or part thereof) by generating fewer hits compared to what other "hungrier" tenants would have had they instead been given the share. Nevertheless, defining QoS in terms of allocated space is consistent with the capacity metrics that cloud providers use for metering.

A related notion is to dedicate a tenant-specific *reserve* space for each tenant. For example, a cloud provider could offer a tenant to lease a dedicated share of 1 MB of space in the data cache for a fixed fee without regard for the utilization of this chunk. The reserve space might be extended to be dynamic over time, such as to accommodate diurnal shifts or marketing campaigns.

Finally, suppose we would promise a certain hit ratio for the tenant as a function of the hit rate it would achieve in isolation [5]. While the property may be intuitively appealing, the mechanism depends on profiling accurate statistics about the cacheability of each tenant's workload.

For BLAZE, we opted to define the minimum QoS-guarantee in terms of reserve space because of the transparency of the guarantee. Each tenant t is given $\text{ReserveSize}(t)$ entries of memory. It is up to the system administrator to ensure that

$$\sum_{t \in T} \text{ReserveSize}(t) \leq \text{CacheSize}$$

where CacheSize is the total memory of the data cache.

To entice customers to use the data cache service, it is in the interest of cloud providers to maximize the overall hit rate while still providing the QoS-guarantees. These goals are in tension: the overall hit rate may decrease by giving resources to a tenant, but the tenant's service quality may improve. Our system assumes that at most an α fraction of the total cache memory can be reserved to satisfy QoS-guarantees, where α is a knob controlled by the cloud provider. The remaining memory, at least $1 - \alpha$ fraction, is then assigned to tenants in a manner that maximizes the hit rate of the data cache. We describe this mechanism in the following section.

4. CACHE PARTITIONING

Dividing a cache among multiple threads or applications to improve the overall hit rate has been previously researched in contexts such as the CPU L2-cache [19], storage caches [8, 15] and database buffer management [5, 7].

Utility-based cache partitioning can be expressed as an integer program [19]. Letting T denote the set of tenant, we will find a partition $(s_t)_{t \in T}$ to

$$\begin{aligned} \text{Maximize} \quad & \sum_{t \in T} H_t(s_t) \\ \text{s.t.} \quad & s_t \geq \text{ReserveSize}(t) \geq \text{MinSize}(t) \quad \forall t \in T \qquad \text{(QoS)} \\ \text{and} \quad & \sum_{t \in T} s_t = \text{CacheSize}. \qquad (1) \end{aligned}$$

Here, H_t denotes the number of hits received by tenant t in given period as a function of the size assigned to the tenant. The QoS objective corresponds to the reserve size for each tenant as discussed in the previous section. Note that in order to allow H_t to be estimated as a non-zero function, each tenant must be given a small minimum cache share

$$\text{MinSize}(t) \geq 1$$

on which they can generate hits.

A simple greedy cache-partitioning algorithm is to iteratively traverse the memory blocks and allocate each to the tenant who expects the greatest gain from the additional memory in terms of hit rate. Assuming the hit rate functions are concave, this greedy algorithm produces an optimal partition [19, 18]. Conveniently, a number of papers have shown that cache workloads have mostly concave hit rate curves: CPU hit rate curves are concave (Belady [5] argues that they follow $1 - \frac{a}{x^b}$ for some a, b); the same goes for typical database management systems [9], and web caches workloads have been said to have logarithmic hit rates as a function of size [4].

The difficulty lies in estimating the dynamically changing H_t functions using only a few samples. Various variants of curve-fitting have been tried, ranging from linear interpolation between sample points [19] to more methods using control theory to dampen oscillations owing to sample error [11]. The right approach depends heavily on the dynamics of the workload at hand. For Blaze, we plan to maintain a handful of recent sample points for each tenant t, and periodically perform a least-squares to fit of the points to a concave function such as

$$H_t(x) = a + b \log x$$

by varying a, b. We furthermore plan to discount fitting error to older sample points so that the function is biased towards fitting recent observations more closely.

5. MULTI-TENANT CLOCK

After computing a cache memory partition for tenants, how should a tenant-aware page replacement policy dynamically move towards the target memory partition? While the answer depends on the algorithm used, the crux of the problem is to decide from what tenant to evict an entry on a cache miss.

We have explored two ways to make Clock multi-tenant. The first approach is based on the generalized clock [17] in which entries have *counters* rather than just a single bit denoting recent use. The clock hand reduces the counter of each entry it passes by 1; on a hit, the entry is reset to the *weight* corresponding to the tenant who owns the entry. By assigning more weight to the tenants whose cache size should not shrink, the mechanism allows the entries of high-weight tenants to stay in the cache without being accessed for longer than in the tenant-oblivious Clock. There is a trade-off between the time to scan for an entry to evict and the accuracy of the counters. Ideally, the relative weights of tenants should be chosen so that the Clock converges to the target partition, but unfortunately the relationship between weights and cache size is convoluted. Nicola et al. [14], for instance, present a Markov chain model for generalized clock and give an approximate expression for average tenant cache size in terms of weights and four other variables assuming that cache requests are independent from one another. Although this approach is minimally invasive to Clock, it proved to be too difficult to control.

The second approach is to enforce tenant sizes directly on top of regular Clock. On a miss caused by tenant t, we consider two cases. If the space occupied by t is less than the target space, then we will evict a non-recent entry from any tenant $t' \neq t$ who is occupying more space than its target allows. Otherwise, we evict a non-recent entry from t itself. As in the regular Clock, if the entries in the cache receive hits during the search for an entry to evict, the search may take arbitrarily long. To ensure cache evictions are timely, Blaze will evict a recently used entry from candidate tenant if it had already had to scan through at least 20% of the entire cache. Since

$$\text{MinSize}(t) \geq 1 \text{ for all } t \in T,$$

this approach eventually converges to the target partition since the target spaces add up to the cache size by equation (1). We are currently evaluating this multi-tenant version of Clock.

6. FUTURE DIRECTIONS

Cache partitioning based on tenant performance is an abstraction that should be separate from the page replacement policy of the cache itself. Blaze, as well as most prior work, has an intimate dependence between the choice of page replacement algorithm and the partitioning mechanism. Recently proposed algorithms, such as ClockPro [10] and CAR [3], have shown improvements over Clock stemming from the use of *ghost lists*, data-less cache entries that track statistics about the hits to recently evicted entries. We plan to evaluate such approaches in the context of a data cache and cache partitioning.

In future work, we plan to both scale Blaze *out* as a fault-tolerant distributed architecture, as well as to scale it *up* by supporting storage layers such as SSDs which have different properties than DRAM. Both dimensions come with intriguing challenges, such as deciding what cache entries to replicate on what nodes while maintaining consistency, and devising ways to minimize wear due to block erase operations on the SSD.

7. CONCLUSION

If data caching is to become a useful cloud service, then QoS and multi-tenancy issues need to be addressed. In this paper, we have discussed design decisions and lessons learned while developing Blaze, a multi-tenant Clock-based data cache which aims to maximize performance while simultaneously offering QoS-guarantees to each tenant. We hope that cloud providers adopt a similar approach and work towards improving data caches for their clients.

8. REFERENCES

[1] CSQL Main Memory Database Cache - Project Homepage. `http://databasecache.com/` (accessed in May 2010).

[2] memcached - Project Homepage. `http://memcached.org/` (accessed in May 2010).

[3] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *In INFOCOM*, pages 126–134, 1999.

[5] K. P. Brown, M. J. Carey, and M. Livny. Goal-oriented buffer management revisited. *SIGMOD Rec.*, 25(2):353–364, 1996.

[6] F. J. Corbato. *Festschrift: In Honor of P. M. Morse*, chapter A Paging Experiment with the Multics System, pages 217–228. MIT Press, 1969.

[7] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.

[8] P. Goyal, D. Jadav, D. S. Modha, and R. Tewari. CacheCow: QoS for Storage System Caches. In *Eleventh International Workshop on Quality of Service (IWQoS 03)*, 2003.

[9] W. W. Hsu, A. J. Smith, and H. C. Young. Characteristics of production database workloads and the TPC benchmarks. *IBM Syst. J.*, 40(3):781–802, 2001.

[10] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 35–35, Berkeley, CA, USA, 2005. USENIX Association.

[11] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An adaptive control framework for QoS guarantees and its application to differentiated caching services. In *International Workshop on Quality of Service (IWQoS)*, pages 23–32, May 2002.

[12] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611, New York, NY, USA, 2002. ACM.

[13] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for QoS-aware clouds. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 237–250, New York, NY, USA, 2010. ACM.

[14] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. *SIGMETRICS Perform. Eval. Rev.*, 20(1):35–46, 1992.

[15] C. Patrick, R. Garg, S. Son, and M. Kandemir. Improving I/O performance using soft-QoS-based dynamic storage cache partitioning. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1 –10, aug. 2009.

[16] P. Saab. Scaling memcached at Facebook. `http://www.facebook.com/note.php?note_id=39391378919` (accessed in May 2010), December 2008.

[17] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3):223–247, 1978.

[18] H. S. Stone, J. Turek, and J. L. Wolf. Optimal Partitioning of Cache Memory. *IEEE Trans. Comput.*, 41(9):1054–1068, 1992.

[19] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Cache Partitioning for Simultaneous Multithreading Systems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 116–127, 2001.