Trojans Cache: A Case for Novel Replacement Algorithms Beyond LRU

Shahram Ghandeharizadeh, Jason Yap, Showvick Kalra, Jorge Gonzalez, Igor Shvager, Elham Keshavarzian

Los Angeles, California

Executive Summary

Introduced more than a quarter of a century ago, the LRU algorithm has had a profound impact on different facets of computer industry ranging from operating systems to database management systems and proxy web cache servers. With advances in processor, storage and networking technology, the traditional LRU algorithm is no longer suitable for a growing number of enterprise applications. This is because LRU lacks concepts such as admission control and cost associated with retrieving a cached data item. Using Trojans Cache configured with main memory only, this white paper demonstrates the limitations of LRU and compares it with an alternative that is faster and more efficient.

1. Introduction

Diverse enterprises ranging from on-line retailers to social networking web sites strive to enhance the experience of their users by providing fast response times. Their typical infrastructure consists of several tiers: database servers, cache servers, applications and web servers. The cache tier minimizes the number of queries issued to the database tier, speeding up those references that observe a cache hit. For example, one may store the results of a query consisting of several join predicates in the cache tier to prevent the database servers from processing the join operation repeatedly, enhancing both the response time experienced by the user and freeing the database servers to process other queries.

Figure 1 shows an example interaction between the different components of a system. In this figure, when the application issues a query to retrieve the result set (value) for a specific key attribute value (key), it first queries the cache server for the result set, Arrow 1. If the value is not found, Arrow 2, then the application server queries the database servers for the results, Arrows 3 and 4. It serializes this result as a value and stores the (key, value) pair in the cache tier for future reference, Arrow 5.

LRU [1, 4] is one algorithm that is used by cache servers. This algorithm manages the (key, value) pairs stored in a cache by their recency of references. When the cache is full and the application attempts to store a new (key, value) pair, LRU victimizes those objects that were least recently accessed in favor of storing this new (key, value) pair.

The recency property of LRU ignores several important details of Figure 1. First, it does not differentiate between the different (key, value) pairs and their associated cost. To elaborate, an application may measure the time to retrieve the results of the query for a given key. The results associated with a key that invokes an expensive query is more valuable because it is more taxing of system resources and impacts the response time experienced by the end users. Second, once the cache is filled with (key, value) pairs, LRU lacks an admission control policy to decide whether a new (key, value) pair should victimize a cache resident (key, value) pair. It deletes an existing (key, value) pair to insert the new one always. Third, LRU is not sensitive to the size of different values stored in the cache.

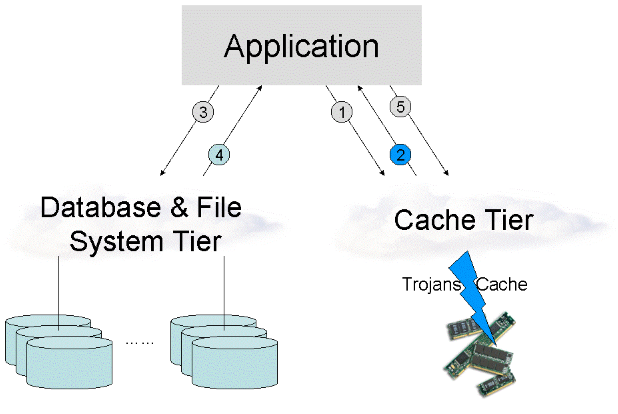

Figure 1: A deployment and a key lookup scenario that fails to find its value in the Cache Tier.

2. Is There an Alternative to LRU?

The answer is a definite "Yes." One algorithm is Interval based GreedyDual [2] (IGD) and it is implemented in Trojans Cache. Trojans Cache [3] (TCache) is designed to support a variety of cache replacement algorithms. It can be configured with DRAM, flash memory, magnetic disk, and a combination of these devices. This white paper focuses on TCache configured with DRAM only in order to compare IGD with LRU.

We compare LRU with IGD using three different metrics: 1) average service time to either insert into the cache or perform a cache lookup, 2) cache hit rate, and 3) byte hit rate. The average service time is a key quality of service metric that impacts the users of the system. Cache and byte hit rates describe the behavior of different algorithms when processing requests. Cache hit rate denotes the fraction of issued requests serviced using the cache. For example, a 90% cache hit rate means that, on the average, 9 out of every 10 requests were serviced using the cache. Byte hit rate is the fraction of total bytes retrieved that were serviced using the cache. For example, a 90% byte hit rate means that if the application retrieved a total of 1 Terabytes of data then 900 Gigabytes of that data was provided using the cache.

Table 1 shows the cache hit rate, byte hit rate, and the time to process 10 million lookups using TCache[1] configured with different amounts of memory: 512 MB, 1 GB, and 1.5 GB. These experiments are modeled after the usage scenario of Figure 1: When the client performs a lookup in TCache, if the lookup succeeds then it proceeds to the next request. Otherwise, it retrieves the (key, value) pair from the database and inserts it into TCache. We assume the database retrieval time is zero to show it is possible to implement IGD efficiently (this assumption is removed in the next few paragraphs). The size of the value ranges from 10 Bytes to 2 Kilobytes and the total size of referenced objects is 1 GB. Each experiment starts with an empty cache. Thus, the cache hit rate may not reach 100%.

Table 1. IGD outperforms LRU by considering both the size of the cached data items and their recency of references.

	Cache Hit Rate		Byte Hit Rate		Elapsed Time (Seconds)	
	LRU	**IGD**	**LRU**	**IGD**	**LRU**	**IGD**
512 MB	52%	56%	51%	50%	1202	564
1 GB	68%	74%	68%	70%	1134	726
1.5 GB	81%	86%	80%	84%	1012	743

Table 1 reveals several interesting observations. First, the cache and byte hit rates of IGD are competitive, exceeding those observed with LRU in almost all cases. This is because IGD considers the size of values when selecting victims. Second, IGD provides a much faster response time than LRU because it performs less work. While LRU inserts every new (key, value) pair into the cache, IGD does not insert a new pair unless its admission control identifies the new pair worthy of evicting the least valued (key, value) pair(s) occupying the cache. This enables IGD to be faster than LRU by more than 30% with 1.5 GB of memory to more than a factor of two with 512 MB of memory. Third, when one compares IGD with itself and different cache sizes, IGD becomes faster with less memory: 25% faster as we reduce memory from 1.5 GB to 512 MB. This is because with smaller cache sizes, a larger fraction of key lookups do not find their values in the cache. With IGD, these are references to (key, value) pairs that are neither as popular nor as expensive as (key, value) pairs occupying the cache. Hence, IGD's admission control rejects the inserts issued by these requests, Step 5 of Figure 1, preventing wasteful work that slows down the system and utilizes resources unnecessarily. LRU cannot do the same because it lacks an admission control policy. LRU becomes faster as we increase cache size from 512 MB to 1.5 GB because its rate of cache misses, cache insertions and evictions slows down with larger cache sizes.

IGD outperforms LRU by a wider margin once we remove the assumption that the time to query the database tier is zero, i.e., negligible. This is because IGD employs the concept of cost to consider the time to retrieve a cached data item from the database tier. The results presented in Table 2 assume this cost is 1, 10, 100, and 1000 milliseconds for different cached items. Assuming a single thread issues ten million requests to the system one after another, every time the referenced object observes a cache hit, its service time is the time to retrieve the object from the cache (one out of two components of last column of Table 1). If the thread observes a cache miss then its service time is the time to retrieve the object from the database tier (1, 10, 100, or 1000 millisecond) plus the time to insert the (key, value) pair into the cache (second component that constitutes the last column of Table 1). A larger cache size reduces the number of cache misses, improving the performance of both LRU and IGD. The performance offered by IGD with 512 MB of memory is almost the same as LRU with 1 to 1.5 GB of memory because IGD selects victims by considering the cost of retrieving a value from the database tier. In essence, one may reduce the amount of required memory (and servers) by using IGD instead of LRU.

[1] Using a 3.4 GHz Intel Pentium 4 PC with Microsoft Windows Server 2003, Enterprise Edition, Service Pack 2.

Table 2. IGD outperforms LRU by considering the cost of retrieving a value from the database tier.

	Total Time (Days)		Gets/Second	
	LRU	**IGD**	**LRU**	**IGD**
512 MB	15.7	7.3	7.4	15.9
1 GB	10.3	3.5	11.2	29.7
1.5 GB	6.5	3.3	17.7	34.7

3. Conclusions

Trojans Cache, TCache, supports IGD as a novel algorithm that considers recency of references to cached data items, their size, and cost of retrieval from a database/file system tier. This results in a cache tier that is several times faster than one using the traditional LRU replacement algorithm. With a cache tier consisting of a fixed number of servers using LRU, IGD holds the potential to reduce the number of server by at least one half.

4. REFERENCES

[1] Denning, P. J. *The Working Set Model for Program Behavior*. Communications of the ACM, Vol. 11, No. 5, Pages 323-333, 1968.

[2] Ghandeharizadeh, S., and Shayande, S. *Greedy Cache Management Techniques for Mobile Devices*. First International IEEE Workshop on Ambient Intelligence, Media, Sensing (AIMS), 2007.

[3] Ghandeharizadeh, S., Yap, J., Kalra, S., Gonzalez, J., Shvager, I., Keshavarzian, E., and Cariño, F. *Trojans Cache*. In preperation.

[4] O'Neil, E., O'Neil, E., and Weikum, G. *The LRU-K Page Replacement Algorithm for Database Disk Buffering*. In ACM SIGMOD, 297-306, 1993.