MEMORY SENSITIVE CACHING IN JAVA

Iliyan Nenov, Panayot Dobrikov

Abstract: *This paper describes the architecture of memory sensitive Java cache, which benefits from both the "on demand" soft reference objects de-allocation provided by the JVM and the good hit-rate behaviour of the Last Recently Used (LRU) eviction policy. The experiments show that such cache implementation outperforms the fixed size LRU cache implementation and has the same stability under high load, heavy memory allocation and garbage collecting. The described technique can efficiently improve the performance of a fixed size memory cache implementation.*

Key Words: *Memory Sensitive Caching, Java, LRU*

INTRODUCTION

Memory cache could be defined as part of the memory, used as continually updated buffer storage, which purpose is to optimize the data transfers between system elements with different characteristics. Caching data in memory is an efficient way to improve the performance of any software. One of the most trivial examples of memory cache implementation is an array in which each element represents data (cache entry) mapped to an index (key).

Usually cache implementations have fixed size and the main design decisions standing upon such cache implementation are what eviction policy to be used. In contrast to fixed size caches, memory sensitive caches can be classified as ones with varying size. Designing a memory cache, able to operate on the Java platform could be more complicated because of the lack of efficient and reliable mechanisms for memory management. When designing memory sensitive cache in Java it is convenient to benefit from the advantages of the only mechanisms for memory management on this platform: the garbage collector and the behaviour of soft/weak references. On the one hand the garbage collector removes the complexity of the explicit memory management; on the other hand it limits the possibility for implementing dynamic resizing of the cache.

By using soft/weak references and the garbage collector functionality one can design and implement *memory sensitive cache* in Java as combination of two parts: the first part acts as an ordinary fixed size cache (*stable part*) and the second one (*memory sensitive* or *dynamic part*) enrols the evicted data from the first part. The dynamic part of the cache is designed as canonicalizing mappings [8] (implementation example could be a hash table in which the value part of its entries is soft/weak reference holding the cached data) thus giving the memory cache the opportunity to change dynamically its size according to the current memory needs.

The experiments show that such memory sensitive cache outperforms the fixed size cache and has good performance under various request distributions; it has as well the same stability under high load, heavy memory allocation and garbage collecting. Consequently memory sensitive technique can successfully be applied for improving the performance of a fixed sized caching strategy.

RELATED WORK

The general requirements for a caching utility have been circumstantially discussed, which resulted in a specification candidate published in Java Community Process (JCP), under JSR 107 - The Java Temporary Cache API (JCache) [5]. Most of the currently existing cache utilities are implementing the specification developed under JSR 107. Each of these use some of the most popular eviction policies: LRU (Least/Last Recently Used), MRU (Most Recently Used), MFU (Most Frequently Used), FIFO (First-In First-Out), TTL (Time-To-Live, after certain amount of time the object automatically leaves the cache) and other derivatives. Some implementations are able to work with plugged in custom implemented eviction policy too. Despite the abundant variety of eviction policies, it is a common opinion that universal eviction policy, capable to perform well in any situation, does not exist.

Common way to measure the *performance* of a cache implementation, in a particular use case situation, is to compute the *hit ratio* of the cache. The hit ratio μ of the cache is a value varying in the interval [0...1]. We define hit ratio as follows: $\mu = \frac{\varsigma}{\Sigma}$ where ς is the number of the successful requests to the cache (the number of the hits) and Σ is the total number of requests to the cache. It is easy to see that, when the value of μ inclines to 1 then all requests to the cache are successful and hence we do not access the data from its source anymore. Respectively, when the value of μ inclines to 0 there is no benefit in using the cache.

Based on its behaviour LRU is one of the most commonly used eviction policies for cache implementations. Under the competitive online algorithms model, it is proven that FIFO and LRU have one and the same competitiveness and in case of locality of references LRU incurs less times on faults [4], [6]; under particular relation among the accessed data it was also proven that LRU is better than FIFO [2]. Parallel with the advantages many publications exhibit overtly the weaknesses of different eviction policies [2], [10].

Young [11] proposed an on-line caching strategy which does not strongly depends on the cache size and empirically showed that if there is particular request distribution then appropriate eviction policy could guarantee optimal performance.

Our experiments show that memory sensitive technique applied to fixed size cache can partly compensate the disadvantages of any eviction policy. Such result is not obvious, as there is no technique in languages with automated memory management (like Java) that can be reliably used to implement common caching strategy in a way that the cache size is properly resized “on demand”. The experiments also show that a cache implementation based on memory sensitive technique can achieve good performance under heavy load, heavy memory allocation, garbage collecting and various request distributions.

THE DESIGN

The memory sensitive cache consists of two parts: a stable one and a dynamic one. Each of the parts of the memory sensitive cache can follow different eviction policies and the dynamic part changes its size by enrolling the data evicted from the stable part. The general rule for the dynamic part is: if there is enough free memory, the cache grows in size by enrolling the evicted objects from the stable part and in case of memory scare the dynamic part will shrink by evicting some of its objects. The dynamic changes in the size of the cache could be realized by relying on the recommendations (specified in the JVM specification) to the garbage collector specification and the management of soft/weak references [8]. As a result, the data which is not often accessed has greater probability to be evicted from the cache.

In general the fixed size cache conforms to the following rule; if a cached object is not frequently used (according to the applied eviction policy) it has to be evicted from the cache to make space for other objects. In contrast, if there is enough free memory, the memory sensitive cache increases its size and enrols the new objects without evicting any potential cache leavers; each saved object could be turned later into successful hit in the cache. Due to this policy, a memory sensitive cache is able to outperform the fixed size cache in hit ratio.

To implement the dynamic part of the cache one has to use data structures that are able to change their size dynamically. Hash tables are preferred because of the O(1) amortized complexity of lookup and modification operations although the resizing of the structure in most common implementations can be costly in practice under load. Other dynamic data structures such as trees do not need resizing but the access time to the elements depends (logarithmically) on the number of the elements they hold. Still, the usage of hash table gives the most satisfying practical results.

The proposed design guarantees the following characteristics of the memory sensitive cache: (1) the performance of the cache will be at least the same as the performance of its stable part, since the memory sensitive cache contains the functionality of the fixed size cache. (2) The cache will be able dynamically to change its size therefore it will not constantly occupy a lot of memory. (3) The design could be implemented reliably in a language with automated memory management (like Java) with no techniques to implement common caching strategy in a way that the cache is properly resized "on demand".

The drawback of this design lies in the access time of the data stored in the dynamic part which in general will be longer than the access time to the stable part; however it will be acceptable compared to the access time of the data stored in the original source.

IMPLEMENTATION GUIDELINES

By using canonicalized mapping the cache elements will be evicted from the cache if and only if they are garbage collected. Evicting elements from the dynamic part of the cache leads to shrinking of the dynamic part of the memory sensitive cache.

It is required for the JVM implementations to clear soft references before an exception for not enough memory is thrown, otherwise it is not specified when or whether to clear them. JVM implementations are encouraged to clear soft references when the program demand for memory exceeds the supply, it is also recommended to clear older soft references before newer ones as well as to clear recently used soft references after those who haven't been used recently [8]. If there is enough memory, soft references will be 'strong' enough to keep the softly referenced data in the heap. If memory becomes insufficient it is up to the garbage collector to clear the softly referenced data, and to decide which elements to collect (evict from the cache).

For the realization of the weak part of the memory sensitive cache it is convenient to use the so called 'canonicalizing mappings' [8], in which we have mapping between a 'key' object and soft reference - 'value' object (holding a cache element which is a potential cache leaver).

One of the most trivial and commonly used implementations of canonicalizing mappings pattern is a hash table having soft/weak reference as a value object in each of its entries. The problem with this realization is when an object is garbage collected its soft reference object will continue to exist pointing to a *null* cache entry, which is not valid. Consequently this problem will lead to unnecessary resizing of the data-structure. To control the resizing of the dynamic part we need to implement a mechanism for clearing, in real time, the invalid soft references from the canonicalizing mapping data-structure (i.e. when the object is garbage collected its soft reference object must be immediately removed from the data-structure too). For implementing such mechanism we can successfully rely on the functionality offered by the *java.lang.ref.ReferenceQueue* class.

The out-performance of the memory sensitive cache compared to fixed size cache comes from its dynamic part which saves the evicted data from the stable part and keeps it in the memory. When data is requested and it is not located in the stable part it will be found in the dynamic part unless it is not garbage collected (evicted from the cache). Memory sensitive cache is expected to show better performance, if objects found in the dynamic part are moved back to the fixed size part. Without formal analysis, such approach is intuitively close to the LRU strategy. In case of memory scare it is up to the garbage collector to decide how to shrink the dynamic part of the memory sensitive cache, therefore with memory sensitive technique one could implement a cache which objects has different probability to be evicted.

THE EXPERIMENTS

It was proven experimentally that the memory sensitive technique can be used successfully to improve the performance of a fixed size cache.

In order to model the requests to the cache we used an *access graph* [1]. The vertices of the graph are objects, the edge from *p* to *q* represents that *q* can be requested immediately after *p.* We associate each vertex of the access graph with a real number in the interval [0...1] corresponding to the probability of the object *q* to be requested after the object *p*. By justifying the values associated with the edges we can simulate different kind of request distributions. The access graph can be successfully used to model locality of references and uniform distribution of requests too.

For the experiments we use a Java implementation following the above described implementation guidelines. Several series of tests with different types of distribution have been done and the results have been compared.

The first experiment aim to verify that there is no overhead associated with the introduction of dynamic part i.e. that memory sensitive cache performs as well as fixed size cache when the former is not able to benefit from its weak part. For this experiment was used an application which took all the free memory, not giving any opportunity for the memory sensitive cache to use its dynamic part.

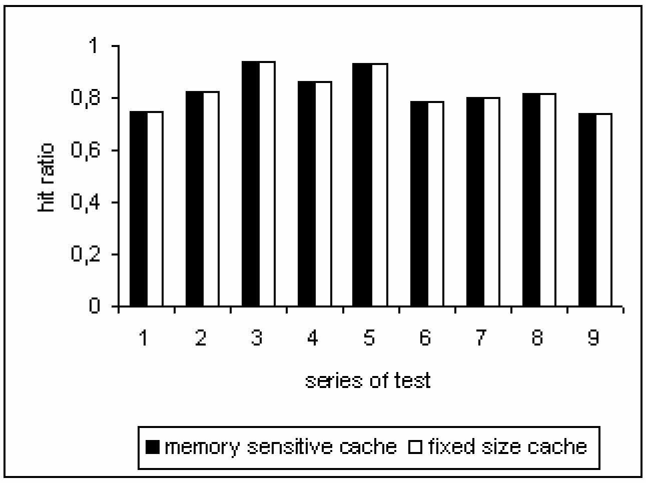

Figure 1: Hit ratio of memory sensitive LRU cache and fixed size LRU cache (requests with defined distribution, heap is almost full)

The results of the fixed size cache and memory sensitive cache are different in less than 0.4%.

The second experiment aim to verify that memory sensitive cache outperforms the fixed size cache, given particular request distribution.

The experiment is divided in two sets of tests: the first set is executed with defined distributions and the second is executed with uniform distribution of requests. In both series of tests the dynamic part of the memory sensitive cache is allowed to change dynamically its size.

The first series of tests is executed with various distributions of requests which did not allow the memory sensitive cache to use its full potential. Despite that it performs better than a fixed size cache.

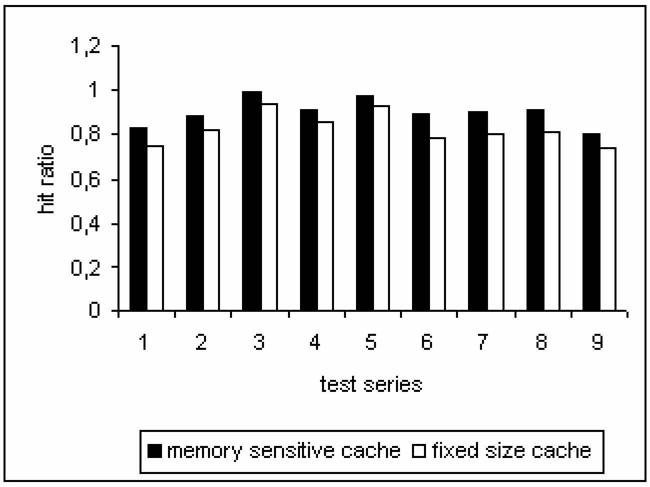

Figure 2: Hit ratio of memory sensitive LRU cache and fixed size LRU cache (requests with defined distribution, enough free memory)

The first set of tests shows that some strict distribution of requests can influence the performance of the memory sensitive cache. Despite the strict order of the requests the memory sensitive cache outperforms the fixed size cache in all the tests, managing to save some of the objects in its dynamic part in contrast to the fixed size cache which has evicted those objects.

The second series of tests aim to verify that the memory sensitive cache outperforms the fixed size cache in case of uniform distribution of requests. The dynamic part of the memory sensitive cache is able to change dynamically its size.

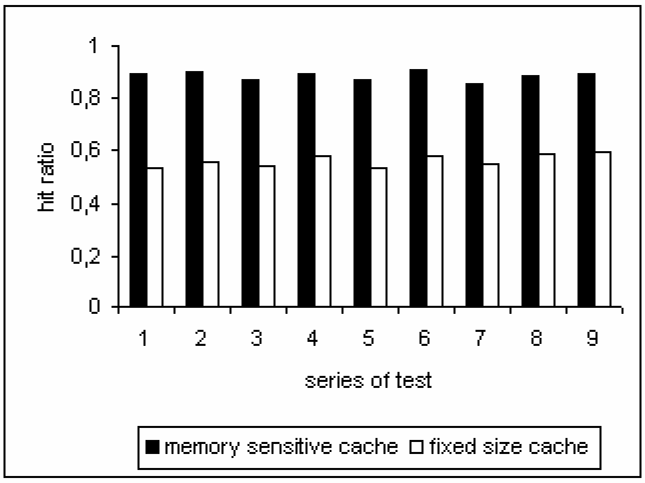

Figure 3: Hit ratio of memory sensitive LRU cache and fixed size LRU cache (requests are with uniform distribution, enough free memory)

The results confirmed that with uniform distribution of requests, the fixed size cache evicts many more valuable objects; most of these objects memory sensitive cache manages to save and retrieve later on. In contrast, the fixed size cache has to retrieve those objects from its original source therefore its response time results are much better.

CONCLUSIONS AND FUTURE WORK

Memory sensitive technique can be used for improving the performance of fixed size caches regardless of the distribution of requests. Such a result is not obvious, as there is no technique in languages, with automated memory management (like Java), that can be used to implement common caching strategy in a way that the cache size is properly and reliably resized “on demand”. In this paper we describe the design and implementation of such memory sensitive cache. The experiments show that in cases when the memory sensitive cache could not benefit from its dynamic part the performance is the same as the performance of the fixed size part of the cache. This is to be expected since memory sensitive cache contains all the functionality the fixed size cache has. The experiments also show that the memory sensitive technique could be used to increase the performance of a fixed size cache implementation without influencing the work of the other applications competing for the same memory the cache is using (i.e. the cache can change its size dynamically according to the memory needs).

As part of a future work, one can evaluate an approach of memory sensitive cache, organized on more than two layers (parts) with different policies for the objects to pass from one layer to another.

There is also potential for improvements in the Java Platform: native canonicalized mappings, in which the whole entry is removed from the data-structure in case the object is garbage collected, could have strong and positive impact on the memory sensitive cache performance.

REFERENCES

[1] Borodin A., S. Irani, P. Raghavan, B. Schieber. Competitive paging with locality of reference. Journal of Computer and System Sciences, 50(2):244-258, April 1995.

[2] Chrobak M., J. Noga. LRU better than FIFO. Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms, 78 – 81, 1999.

[3] Cognitive Science Laboratory at Princeton University - WordNet lexical database http://wordnet.princeton.edu/cgi-bin/webwn

[4] Irani S., A. Karlin, S. Phillips. Strongly competitive algorithms for paging with locality of reference. Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms, 228 – 236, 1992.

[5] Java Temporary Cache API (JSR107 – www.jcp.org)

[6] Shedler G., C. Tung. Locality in page reference strings. SIAM Journal on Computing, 1:218-241, 1972.

[7] Sleator D., R. E. Tarjan. Amortized efficiency of list update and paging rules. Comm. ACM, 28(2):202-208, February 1985.

[8] Venners B. Inside Java 2 Virtual Machine. Blacklick, OH: McGraw-Hill, 1999.

[9] Young N. On-line file caching. Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms, 82-86, 1999.

[10] Young N. The k-server dual and loose competi-tiveness for paging. Algorithmica, 11(6):525-541, June 1994.

[11] Young N. On-line caching as cache size varies. Proceedings of the second annual ACM-SIAM symposium on discrete algorithms, 241 – 250, 1991.

ABOUT THE AUTHORS

Iliyan N. Nenov, candidate for MSc degree in Computer Science, Sofia University "Sv. Kliment Ohridski", Sofia, Bulgaria; Software engineer at SAP Labs Bulgaria, Sofia, Bulgaria (http://www.sap.com/company/saplabs/bulgaria) working in the Java Server Technology Group. E-mail: ilian.nenov@sap.com Office Phone: (+359) 2 9157-451

Panayot M. Dobrikov, MSc Computer Science at Sofia University "Sv. Kliment Ohridski", Sofia, Bulgaria; Software architect at SAP AG. Walldorf, Germany (www.sap.com) working in the Java Server Technology Group.
E-mail: panayot.dobrikov@sap.com Office Phone: (+49) 6227 7-64631