Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 3, Issue. 9, September 2014, pg.838 – 840

RESEARCH ARTICLE

Cache Memory – Various Algorithm

Priyanka Yadav[1], Vishal Sharma[2], Priti Yadav[3]

yadav.priyanka2601@gmail.com [1], 02vishalsharma@gmail.com [2], yadav.preeti44@yahoo.com [3]

[1]Dronacharya College Of Engineering (CSE, Department)

[2]Dronacharya College Of Engineering (CSE, Department)

[3]Dronacharya College Of Engineering (CSE, Department)

ABSTRACT: In this paper we have proposed the new algorithm for cache memory organization. The beginning of paper explain the term cache to us followed by architecture of cache memory organization which include cache manager, memory cache, private file cache and shared file cache. Further we have investigated various different types of cache algorithms their advantages and disadvantages. Cache memory is mainly measured by hit ratio and latency of cache. In our new proposed algorithm cache has high hit ratio and less latency than others. This paper gives the overview of cache architecture and focus on new cache optimization algorithm and at the end highlight the future of cache structure. Thus, going through this paper one will end up with a good understanding of cache and its Optimizing techniques

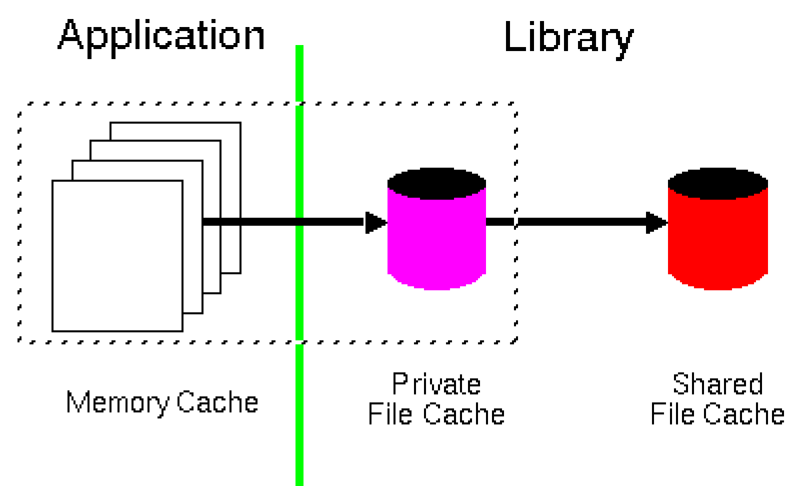
KEYWORDS: Cache memory, Cache manager, Cache algorithms, Hit ratio, Latency.

1. INTRODUCTION

A **CPU cache** is a cache used by the central processing unit (CPU) of a computer to reduce the average time to access data from the main memory. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2 etc.). Cache (pronounced cash) memory is extremely fast memory that is built into a computer's central processing unit (CPU), or located next to it on a separate chip. The CPU uses cache memory to store instructions that are repeatedly required to run programs, improving overall system speed. The advantage of cache memory is that the CPU does not have to use the motherboard's system bus for data transfer.

2. THE CACHE MANAGER

Caching is a required part of any efficient Internet access applications as it saves bandwidth and improves access performance significantly in almost all types of accesses. This sections describes the architecture behind the cache management in the Library. The cache management is intended to be used both as a proxy cache and a client cache or simply as a cache relay. It does not include the interaction between an application and a proxy server as this is regarded as an external access and hence outside the scope of the local cache. The basic structure of the cache is illustrated in the figure below.



The figure described the cache hierarchy starting from left to right; it does not describe the data flow. Any of the three cache handlers can be left out in which case a cache request will fall through to the next handler in the hierarchy and finally be passed to the protocol manager which issues a request to either the origin server, a proxy server, or a gateway. Any of the handlers can also be short circuited by using a set of cache directives which are explained in the User's Guide. In the following, each part will be described in more detail.

© 2014, IJCSMC All Rights Reserved

Memory Cache

The memory cache is completely handled by the application and is only consulted by the Library when servicing a request. It is considered private to a specific instance of an application and is not intended to be shared between instances. Handling the memory cache includes the following tasks: object storage, garbage collection, and object retrieval. The application can initiate a memory cache handler by registering a call back function that is called from within the Library on each request. The details of this registration is described in the User's Guide.

Traditionally, the memory cache is based on handling the graphic objects described by the HyperDoc object in memory as the user keeps requesting new documents. The HyperDoc object is only declared in the Library - the real definition is left to the application as it is for the application to handle graphic objects. For example, the Line Mode Browser has its own definition of the HyperDoc object called HText which describes a fully parsed HTML object with enough information to display itself to the user. However, the memory cache handler can handle other objects than HTML, for example images, audio clips etc. It is important to note that the Library does not imply any limitations on the usage of the memory cache.

The memory cache must define its own garbage collection algorithm which can be based on available memory etc. Again, the Line Mode Browser has a very simple memory management of how long objects stay around in memory. It is determined by a constant in the GridText module and is by default set to 5 documents. This approach can be much more advanced and the memory garbage collection can be determined by the size of the graphic objects, when they expire etc. but the API is the same no matter how the garbage collector is implemented.

Private File Cache

The private file cache is to be regarded as a direct extension of the memory cache as intended for intermediate term storage of data objects. As the memory cache, it is intended to be private to a single instance of an application as long as the instance is running. However, as a file cache is persistent, it can be shared between several instances of various applications as long as exactly one instance owns the private cache at any one time. The single ownership of a private cache means that the cache can be accessed via the local file system by one instance of an application only.

There are two purposes of the private file cache:

1. To maintain a persistent cache for applications that do not have a shared cache.
2. To maintain a private persistent cache for specific groups of documents that are not to be shared among other applications. Examples of such are documents with a HTTP header *Pragma: Private* which will be introduced in HTTP/1.1

Often an important difference between the memory cache and the file cache is the format of the data. As mentioned above, in the memory cache, the cached objects can be pre-parsed objects ready to be displayed to the user. In a file cache the data objects are always stored along with their metainformation so that important header information like Expires, Last-Modified, Language etc. is a part of the stored object together with any unknown metainformation that might be a part of the object.

Shared File Cache

A shared file cache which can be accessed by several independent applications requires its own cache manager in order to ensure a consistent cache and to handle garbage collection. A shared file cache can in many ways be regarded as similar to a proxy cache as a single application do not know when a cached object is either discarded or refreshed in the shared cache area.

If a shared cache manager does exist then the only remaining purpose of a private file cache is to store explicitly private objects. All other objects will be stored in the shared cache.

As for the private file cache, the data objects are always stored along with their meta information so that any meta information associated with an object can be returned to the requesting application.

3. CACHE ALGORITHMS

Belady's Algorithm

The *most* efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This optimal result is referred to as Belady's optimal algorithm or the clairvoyant algorithm. Since it is generally impossible to predict how far in the future information will be needed, this is generally not implementable in practice. The practical minimum can be calculated only after experimentation, and one can compare the effectiveness of the actually chosen cache algorithm.

Least Recently Used (LRU)

Discards the least recently used items first. This algorithm requires keeping track of what was used when, which is expensive if one wants to make sure the algorithm always discards *the* least recently used item. General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits. In such an implementation, every time a cache-line is used, the age of all other cache-lines changes. LRU is actually a family of caching algorithms with members including: 2Q by Theodore Johnson and Dennis Shasha and LRU/K by Pat O'Neil, Betty O'Neil and Gerhard Weikum.

Most Recently Used (MRU)

Discards, in contrast to LRU, the most recently used items first. In findings presented at the 11th VLDB conference, Chou and Dewitt noted that "When a file is being repeatedly scanned in a [Looping Sequential] reference pattern, MRU is the best replacement algorithm."[3] Subsequently other researchers presenting at the 22nd VLDB conference noted that for random access patterns and repeated scans over large datasets (sometimes known as cyclic access patterns) MRU cache algorithms have more hits than LRU due to their tendency to retain older data.[4] MRU algorithms are most useful in situations where the older an item is, the more likely it is to be accessed.

Pseudo-LRU (PLRU)

For CPU caches with large associativity (generally >4 ways), the implementation cost of LRU becomes prohibitive. In many CPU caches, a scheme that almost always discards one of the least recently used items is sufficient. So many CPU designers choose a PLRU algorithm which only needs one bit per cache item to work. PLRU typically has a slightly worse miss ratio, has a slightly better latency, and uses slightly less power than LRU.

Random Replacement (RR)

Randomly selects a candidate item and discards it to make space when necessary. This algorithm does not require keeping any information about the access history. For its simplicity, it has been used in ARM processors.[5] It admits efficient stochastic simulation.[6]

Segmented LRU (SLRU)

An SLRU cache is divided into two segments, a probationary segment and a protected segment. Lines in each segment are ordered from the most to the least recently accessed. Data from misses is added to the cache at the most recently accessed end of the probationary segment. Hits are removed from wherever they currently reside and added to the most recently accessed end of the protected segment. Lines in the protected segment have thus been accessed at least twice. The protected segment is finite, so migration of a line from the probationary segment to the protected segment may force the migration of the LRU line in the protected segment to the most recently used (MRU) end of the probationary segment, giving this line another chance to be accessed before being replaced. The size limit on the protected segment is an SLRU parameter that varies according to the I/O workload patterns. Whenever data must be discarded from the cache, lines are obtained from the LRU end of the probationary segment.[7]"

© 2014, IJCSMC All Rights Reserved

2-way set associative

Used for high-speed CPU caches where even PLRU is too slow. The address of a new item is used to calculate one of two possible locations in the cache where it is allowed to go. The LRU of the two is discarded. This requires one bit per pair of cache lines, to indicate which of the two was the least recently used.

Direct-mapped cache

Used for the highest-speed CPU caches where even 2-way set associative caches are too slow. The address of the new item is used to calculate the one location in the cache where it is allowed to go. Whatever was there before is discarded.

Least-Frequently Used (LFU)

Counts how often an item is needed. Those that are used least often are discarded first.

Low Inter-reference Recency Set (LIRS)

A page replacement algorithm with an improved performance over LRU and many other newer replacement algorithms. This is achieved by using reuse distance as a metric for dynamically ranking accessed pages to make a replacement decision. The algorithm was developed by Song Jiang and Xiaodong Zhang.

Adaptive Replacement Cache (ARC)

Constantly balances between LRU and LFU, to improve the combined result.[8] ARC improves on SLRU by using information about recently-evicted cache items to dynamically adjust the size of the protected segment and the probationary segment to make the best use of the available cache space.

Clock with Adaptive Replacement (CAR)

Combines Adaptive Replacement Cache (ARC) and CLOCK. CAR has performance comparable to ARC, and substantially outperforms both LRU and CLOCK. Like ARC, CAR is self-tuning and requires no user-specified magic parameters.

4. PROPOSED ALGORITHM

The average memory reference time is

$$T = m * T_m + T_h + E$$

where

T = average memory reference time

m = miss ratio = 1 - (hit ratio)

T_m = time to make a main memory access when there is a miss (or, with multi-level cache, average memory reference time for the next-lower cache)

T_h= the latency: the time to reference the cache when there is a hit

E = various secondary effects, such as queuing effects in multiprocessor systems

There are two primary figures of merit of a cache: The latency, and the hit rate. There are also a number of secondary factors affecting cache performance.[1]

The "hit ratio" of a cache describes how often a searched-for item is actually found in the cache. More efficient replacement policies keep track of more usage information in order to improve the hit rate (for a given cache size).

The "latency" of a cache describes how long after requesting a desired item the cache can return that item (when there is a hit). Faster replacement strategies typically keep track of less usage information—or, in the case of direct-mapped cache, no information—to reduce the amount of time required to update that information.

Each replacement strategy is a compromise between hit rate and latency.

Measurements of "the hit ratio" are typically performed on benchmark applications. The actual hit ratio varies widely from one application to another. In particular, video and audio streaming applications often have a hit ratio close to zero, because each bit of data in the stream is read once for the first time (a compulsory miss), used, and then never read or written again. Even worse, many cache algorithms (in particular, LRU) allow this streaming data to fill the cache, pushing out of the cache information that will be used again soon (cache pollution).

Hence Algorithm is as follow:

1. We must design cache with high speed and performance should include combination of LRU algorithm and Fully Associative mapped cache.

2. There should be criteria to Remove the least used data but not completely from the cache in spite that it must be store in backup hand of cache hence, if needed in future so it can be accessed more quickly.

Search must be performed simultaneously in directive of cache and cache itself.

5. CONCLUSION

Cache performance optimizations can yield significant execution speedups, particularly when applied to numerically intensive codes. Several of the basic optimization techniques can automatically be introduced by optimizing compilers, most of the tuning effort is left to the programmer. This is especially true, if the resulting algorithms have different numerical properties; e.g., concerning stability, robustness, or convergence behavior. In order to simplify the development of portable (cache) efficient numerical applications in science and engineering, optimized routines are often provided by machine{specific software libraries. Future computer architecture trends further motivate research efforts focusing on memory hierarchy optimizations. Forecasts predict the number of transistors on chip increasing beyond one billion. Computer architects have announced that most of the transistors will be used for larger on{chip caches and on chip memory. Most of the forecast systems will be equipped with memory structures similar to the memory hierarchies currently in use. While those future caches will be bigger and smarter, the data structures presently used in real{world scientific codes already exceed the maximum capacity of forecast cache memories by several orders of magnitude. Today's applications in scientific computing typically require several Megabytes up to hundreds of Gigabytes of memory.

REFERENCES

1. N. Ahmed, N. Mateev, and K. Pingali. Tiling Imperfectly{Nested Loop Nests. In Proc. of the ACM/IEEE Supercomputing Conference, Dallas, Texas, USA, 2000.
2. R. Allen and K. Kennedy. Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, San Francisco, California, USA, 2001.
3. M. Altieri, C. Becker, and S. Turek. On the Realistic Performance of Linear Algebra Components in Iterative Solvers. In H.-J. Bungartz, F. Durst, and C. Zenger, editors, High Performance Scientific and Engineering Computing, Proc. of the Int. FORTWIHR Conference on HPSEC, volume 8 of LNCSE, pages 3{12. Springer, 1998.
4. B.S. Andersen, J.A. Gunnels, F. Gustavson, and J. Wa_sniewski. A Recursive Formulation of the Inversion of Symmetric Positive Definite Matrices in Packed Storage Data Format. In Proc. of the 6th Int. Conference on Applied Parallel Computing, volume 2367 of LNCS, pages 287{296, Espoo, Finland, 2002. Springer.
5. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz,.
6. An Overview of Cache Optimization Techniques and Cache Aware Numerical Algorithms.
7. Wikipedia