Multi-Level Caching

Praneeth Yerramothu

Computer Science

Texas Tech University

Lubbock, USA

Praneeth.yerramothu@ttu.edu

***Abstract*— Multi-level caching has been initially implemented during this decade. Multilevel cache is using more than one level of cache implementation in order to make the speed of cache access almost equal to the speed of the CPU and to hold a large number of cache objects. We mainly deal with two techniques called PROMOTE and DEMOTE. These two technique help to sort and analyze the data items that needed to be placed in the cache level. Since there are multiple cache levels in the entire cache system, it is not impossible to have multiple data items placed in multiple cache levels**.

# Introduction

Multilevel cache uses more than one level of cache implementation in order to make the speed of cache access almost equal to the speed of the CPU and to hold a large number of cache objects. Due to evolution of memory storage devises from Hard disks to SSD, the need for cache to be implemented in this new type of storage devise as well, has become very vital for efficiency and to maintain the performance of the system high. In order to achieve that, Multi-level caching has been introduced. Though, this is proven to be very effective, it has failures of its own. In a processor with multiple cores, each core has its own L1 cache. This allows the core to read and write from and to the cache without worrying about interfering with other cores. The cores need shared storage, though, so that they can exchange certain information easily. The L2 cache is shared by all cores, so it's used as a sort of communal storage space where information is available for all threads.

The difference between the L2 and L3 caches is the compromise part. Caches are made of static RAM, or SRAM. This is different from the Dynamic RAM (DRAM) that makes up your main memory. Dynamic RAM needs to be "refreshed" periodically, that is, over time DRAM cells lose their value unless they are read and then re-written. Your memory controller does this automatically, but every time the memory controller has to do this (thousands of times a second) it is unable to read/write values for the processor until it is done. This slows down DRAM. SRAM does not have this limitation; SRAM will retain its value indefinitely as long as it has

operating power, making it much faster. So, your caches (both L2 and L3) are made of SRAM. The trouble is that SRAM is very expensive; In an embodiment, the data storage device utilizes a first level (L1) and a second level (L2) of cache memory to temporarily store the received data prior to commission to the specified storage location(s). In this embodiment, the data storage device first sends the data to the L1 cache memory, and subsequently thereafter, the data storage device transfers the data from the L1 cache memory to the L2 cache memory. Eventually, the data storage device transfers the data from the L2 cache memory to the specified storage location(s). The times at Which data transfer is initiated between the L1 cache memory and the L2 cache memory and/or between the L2 cache memory and the specified storage location(s) may be dependent upon occurrence of a predetermined threshold condition. For instance, in accordance with one embodiment, either one of these data transfer events may be initiated after operation of the data storage device goes idle, meaning that the device is not performing any data storage or retrieval operations. In another embodiment, the predetermined threshold condition may relate to the percentage of used or available capacity in the cache (L1 or L2) from Which the transfer is initiated. In this embodiment, the applied percentage factor (either availability or use) is analyzed against a predetermined percent capacity to determine Whether such a transfer should be initiated. For example, data may be transferred from the L1 cache memory to the L2 cache memory if the percentage of capacity currently available in the L1 cache falls below the predetermined percent capacity, which in this analysis relates to available capacity. Alternatively, data may be transferred from the L1 cache memory to the L2 cache memory if the percentage of capacity currently being used in the L1 cache exceeds the predetermined percent capacity, which in this analysis relates to used capacity.

It is contemplated that the data storage device may be a disc drive in Which data is stored on one or more recordable disc media. The recordable disc media is divided into tracks, and the tracks are divided further into sectors, which serve as the primary storage locations specified for packet storage by sending interfaces. In this embodiment, the L2 cache memory may be located on a reserved group of tracks located on any portion of the recordable disc media. For example, the L2 cache

memory may reside on the outer periphery of the disc media. In accordance with another disc drive embodiment of the present

invention, data sent from the sending interface to the drive for storage on the media may be in the form of packets that are sized smaller than data region of the sectors on the recordable media. For instance, the packets may be 512 bytes in length, Whereas the data region of these sectors maybe 1024 bytes in length, 1536 bytes in length, 2048 bytes in length, etc. In this embodiment, multiple packets stored in the L1 cache may be directly transferred to a single sector if these multiple packets align in a manner that the packets Will be stored to the entire allocated data capacity of the sector. As such, these multiple packets skip being temporarily stored in the L2 cache memory. Packets that do not align in this manner are transferred to the L2 cache memory as described above and Written to the specified disc sector(s) using a process Wherein data currently stored on the sector(s) is first read into cache, then modified with the current data, and finally Written back to the sector(s). Embodiments of the invention may be implemented as an article of manufacture such as a solid state, non-volatile memory device or a computer-readable program storage device. The computer-readable program storage device maybe a computer storage media readable by a computer processor and encoding a computer program of instructions for executing a computer process.

Multilevel cache uses more than one level of cache implementation in order to make the speed of cache access almost equal to the speed of the CPU and to hold a large number of cache objects. Due to evolution of memory storage devises from Hard disks to SSD, the need for cache to be implemented in this new type of storage devise as well, has become very vital for efficiency and to maintain the performance of the system high. In order to achieve that, Multi-level caching has been introduced. Though, this is proven to be very effective, it has failures of its own.

## Problem with Mutual Inclusion

Multilevel Inclusion is a technique which contains data that are common to both the levels, i.e. that some data is present in both the most recent data cache level 1 and recently used data at level 0. Normally level 0 cache size is made smaller when compared to that of the level 1 cache.

The main disadvantage of this implementation is that it suggests and uses the two different location for the same data which results in waste of memory space. This concept of multilevel caching already exists and used by AMD in their microprocessor chips. The main problem with this type of caching is that for the same data we need two different levels of cache and also it is a waste of memory. This will also result in removal of any cache results in updating of both levels which is a tedious job to perform.

## Overcoming Inclusion problem:

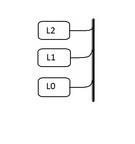
Multilevel Exclusion is a technique where only the data which is not in the level 1 are present in the smaller cache of level 0. This has a greater advantage of having less associativity between the two level cache. Such a concept is successfully used in all recent Pentium processors. It has an advantage of using smaller memory consumption and easy cache updating policy.

*Bounds for optimal boundaries*

Bounds for Optimal Performance: In the study of caching algorithms, it is invaluable to know the offline optimal performance that a multi-level cache can deliver. While Belady’s MIN [4] (an offline algorithm) is considered optimal for single-level caches, it cannot be applied to multi-level cache scenarios, where, apart from the hit ratio, the location of hits is also extremely important. Our first contribution provides insight into the optimal offline performance of multi-level caches. We provide policies called OPT-UB and OPT-LB that provably serve as upper and lower bounds for the optimal offline performance for multi-level caches along both average response time and inter-cache bandwidth usage metrics. Through a series of experiments on a wide gamut of traces, cache sizes and configurations, we demonstrate that OPT-UB and OPT-LB are very close bounds on the optimal average response time, running on an average, within 2.18% and 2.83% of each other for all the tested two-cache and three-cache hierarchies, respectively. Even for more complex hierarchies, the bounds remain close at about 10% of each other. This novel result enables us to estimate for the first time, the performance gap between the current state-of-theart algorithms and the offline optimal for multi-level caches.

## Problem Statement

The use of multi-level cache system makes each cache level have no association with the other levels. There is no connection between the cache levels. Because of no communication between the cache levels, the incoming data does not know or have a specific target destination (L0, L1, L2 in case of 3-level cache system) to occupy.



*Figure 1: Multi-Level(3) Cache System*

The possibility of a data item from the memory unit, occupying any cache level, is inevitable. But the question lies in the fact as to which cache level it would occupy. Say L1 cache is non-empty/empty and not fully occupied. But L2 cache in fully occupied and has no more space to spare for a new data item. When the data item is forced to occupy space in L2 cache, it is forcing L2 cache to evict already existing data item(s). This phenomenon is totally not necessary as there is room to spare for a new data item in L1. Had the data item been placed in L1 instead of L2, the data item(s) in L2 would not be evicted, thereby maintaining efficiency and performance. In addition to above existing problem, there is a principle called **Mutual inclusion** principle that only makes the above scenario even worse. Multilevel Inclusion is a technique which contains data that are common to both the levels, i.e that some data is present in both the most recent data cache level 1 and recently used data at level 0. Normally level 0 cache size is made smaller when compared to that of the level 1 cache. One of the key problems that the computer scientists are facing is how to make use of all the cache levels effectively without having to replace recently used data bits which thereby reduces efficiency and performance.

## Motivation

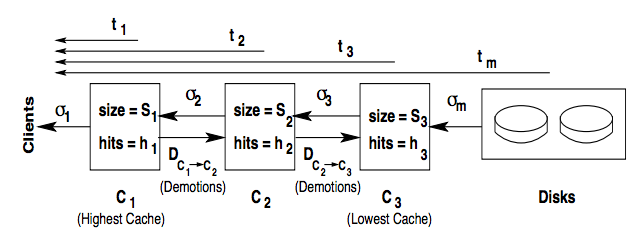
Ever since the digital world started replacing HDD (Hard disk Drive) with SSD (Solid State drive), there have been many other things that took a change in nature. Among those things, **Cache** system is one of the prominent techniques to be included in, with SSD. The compelling demand for redefinition and remodeling of cache architecture in the recently developed SSD and its craving for perfect combination with SSD made me consider this concept as my course project. In this semester, I am also taking Computer Organization Architecture course, as well. In that course, I have learnt the techniques surrounding cache and fully understood its sole purpose. These are few of the many reasons as to why I chose this topic for the course project.

## Related Work

Research was made in to finding out more information about the techniques that involved the sorting and placing of data objects into proper cache levels. To find out more about the PROMOTE and DEMOTE techniques, I read the IEEE paper on ‘Multi-level Exclusive Caching: Ofﬂine Optimality and Why promotions are better than demotions’. That paper clearly explains why PROMOTE is always better than DEMOTE. It also explains the disadvantages the DEMOTE holds as it is not a proper way to put a data item in a cache layer. It affects the efficiency and performance of the computer because it requires extra network process to compute few calculations to place the data item in an appropriate location.

The DEMOTE technique, or equivalently the Global technique, can be applied to many general purpose single-level caching policies (like, LRU, MQ, ARC, etc) to create multilevel versions that achieve exclusivity of cache contents. As with any exclusive caching scheme, DEMOTE should only be used in scenarios that benefit from exclusive caching. While the DEMOTE technique strives to improve the aggregate hit ratio over the non-exclusive variant, the overall performance might in fact suffer because of the cost of the DEMOTE operation, including: (i) network traffic to send evicted pages to lower caches, and (ii) processor cycles consumed to prepare, send and receive demoted pages. This has thwarted the practical deployment of DEMOTE in real systems.

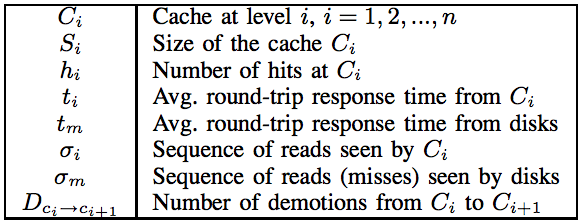
Figure 2: A multi-level single-path cache hierarchy



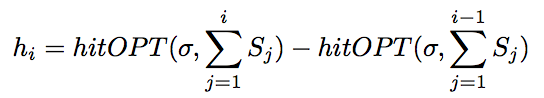
These issues are overcome by the use of technique called PROMOTE. The technique deals with the block and the way it is inserted into a particular cache layer. The search starts from the top most layer to check if a particular block is a hit or a miss. If it is a hit, the data item is inserted in the higher layer in cache provided that the computed probabilistic value is greater than that of a particular fixed value and this allows the process to insert the data item in the right place.

Not only do we show that PROMOTE is applicable to a wider range of algorithms and cache hierarchies, it is on an average, 2x more efficient than DEMOTE requiring only half the inter cache bandwidth between the various cache levels. In a wide variety of experiments, while both techniques achieved the same aggregate hit ratio, PROMOTE provided 13.0% and 37.5% more hits in the highest cache than DEMOTE when the techniques were applied to LRU and ARC algorithms, respectively, leading to better average response times even when we allow DEMOTE unlimited inter-cache bandwidth and free demotions. In limited bandwidth scenarios, PROMOTE convincingly outperforms DEMOTE. For example, in a trace from a real-life scenario, PROMOTE provided an average response time of 3.21ms as compared to 5.43ms for DEMOTE

on a two-level hierarchy of ARC caches, and 5.61ms as compared to 8.04ms on a three-level cache hierarchy.



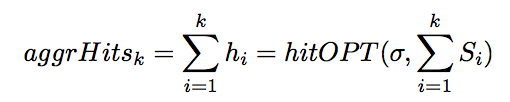
If the above variables are defined by the corresponding entities, then the performance evaluation is done by using the following conceptual computations,



Note that we intend to compute a theoretical upper bound which is necessarily achievable.

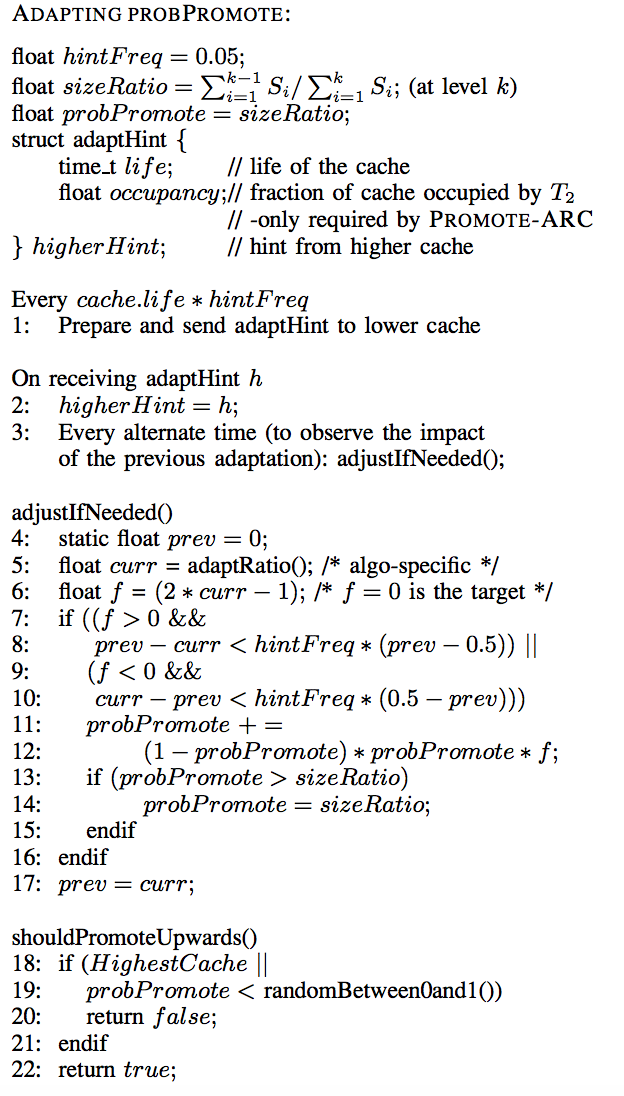
Where,

Number of hits and the optimal size of cache layer j, where the hit takes place is seen here.



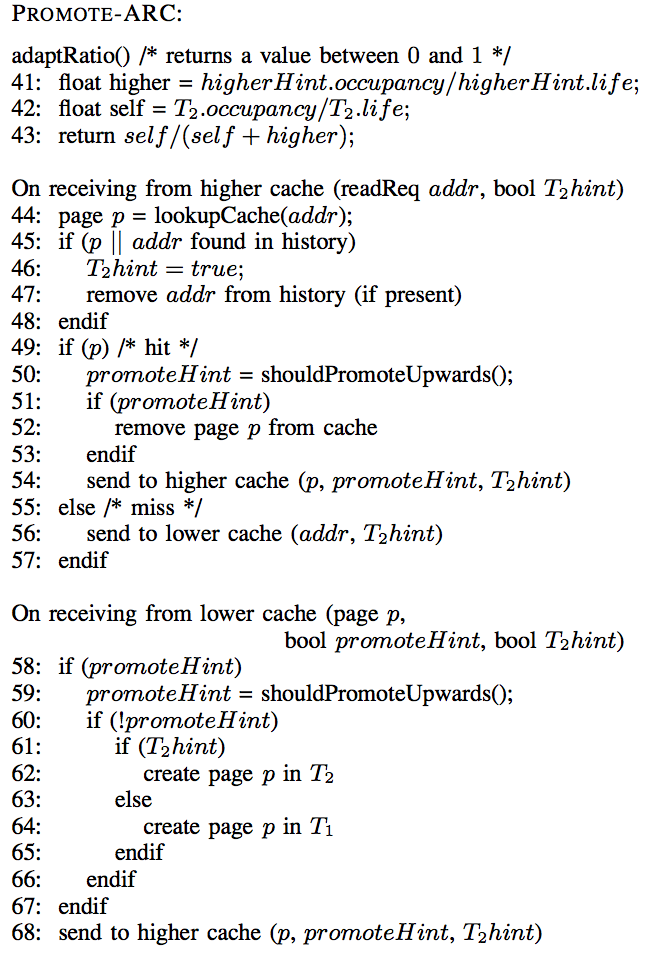
By the definition of hitOP T , this is the same as that obtained by Belady’s OPT on a cache of the aggregate size. Since Belady’s OPT is known to deliver the maximum possible hit ratio, aggrHitsk is maximized for all k ≤ n.

*Algorithm to perform PROMOTE*



## 

Figure 3: The PROMOTE-LRU method



## Technical Approach

I have implemented PROMOTE and DEMOTE using C++ and gcc compiler. Following are the code snippets that show the functionality of either of the techniques.

int i,j=0;

int \*h;

h=(int\*)malloc(sizeof(\_gConfiguration.totalLevels));

for(i=0;i<\_gConfiguration.totalLevels;i++)

{

if(j>=1 && j<\_gConfiguration.totalLevels)

{

if(\_gTestCache[i].isFull())

{

h[i]=\_gTestCache[i].pop\_back();

\_gTestCache[i].insert(\_gTestCache[i].end(),h[i-1]);

}

else

{

\_gTestCache[i].insert(\_gTestCache[i].end(),h[i-1]);

break;

}

}

else if(j<1)

{

if(\_gTestCache[i].isFull()) // if lowest level cache is full

{

h[i]=\_gTestCache[i].pop\_back();h

\_gTestCache[i].insert(\_gTestCache[i].end(),key);

}

else

{

\_gTestCache[i].insert(\_gTestCache[i].end(),key);

break;

}

j++;

}

}

Figure 4: DEMOTE Technique

while(!all\_evicted\_page\_empty())

{

for(int i = 0 ; i < \_gConfiguration.totalLevels ; i++) {

list<uint64\_t> evict\_list(\_gTestCache[i]->get\_evict\_entries());

while(!evict\_list.empty()){

uint64\_t key=evict\_list.back();

evict\_list.pop\_back();

\_gTestCache[i]->insert(\_gTestCache[i].end(),newCacheItem)

}

}

## The popped item from the cache layer is tried against the higher layer. If that layer is full, evict or remove the existing item from the cache layer and insert the new data item.

for(i=hitLayer-1; i >=0 ;i--)

{

if( newFlags)

{

\_gTestCache[++i]->remove(newReq.fsblkno,newCacheAtom);

r=randFunc();

if(r>0.5)

{

\_gTestCache[--i]->insert(newReq.fsblkno,newCacheAtom);

}

else

break;

}

else if(!noHitAtom)

{

\_gTestCache[\_gConfiguration.totalLevels]->insert(newReq.fsblkno,newCacheAtom);

while(i>0)

{

r=randFunc();

if(r>0.5)

{

\_gTestCache[--i]->insert(newReq.fsblkno,newCacheAtom);

}

else

break;

}

}

}

Figure 4: The PROMOTION TECHNIQUE

float randFunc()

{

float randval=static\_cast <float> (rand())/ static\_cast <float>(RAND\_MAX) ;

return randval;

}

Here, the code refers to the PROMOTION technique. If there is a page hit, then the block is moved from one layer to another. When this happens, the condition on the ‘if’ checks to see if the value on the page hit is greater than that of the one that’s been fixed for every level already. This process goes on until the condition fails or the process reaches the highest possible layer in the cache system.

## Research Findings

While implementing DEMOTE, it is noted that the network traffic is used in excess to send evicted pages to lower caches. When accessing the details of each page, the process checks to see if that cache level is full. If it is full, the cache would not let the process use that layer to insert an element into it. This would mean use excess network bandwidth to store and in turn, it would need more cycles to run. Thereby, it affects its performance and efficiency.

Also, all the read requests have to wait until the least recently pages are evicted. Only after all the pages have been arranged, the DEMOTE technique will be ready for a new read request. This means even performance at user-level is greatly affected. The user has to wait until the processed request is finished working. So the next request has to wait for its execution to finish. This causes overhead and thereby affects performance overall.

##### Conclusion

In this project, I have successfully implemented the techniques that involve moving around a data item and placing it in a cache

layer that is present in the cache. I have also researched on concepts such as the Replacement algorithms and Data transfer methods. In the end, We have concluded that , in our simulation, according to the data that we have acquired, PROMOTION is always better than DEMOTION technique. We have proved this in this report along with the way the cache layers are being filled with data items in their appropriate cache layers.

*Future Work*

* Multi-level Exclusive Caching: Ofﬂine Optimality and Why promotions are better than demotions, by Binny S. Gill, IBM Almaden Research Center.
* Use of Average response time and number of hit points to determine the efficiency of the PROMOTE Technique.
* To tabulate and represent, graphically, the data gathered after running performance tests on both DEMOTE and PROMOTE techniques.

##### Acknowledgment

My sincere gratitude to Dr. Yong Chen, Ph.D., Asst. Professor, for helping me in choosing this project and I thank Mr. Wei Xei, Advisor for guiding me all the way through the semester to conduct and perform research in this topic.

##### References

1. 1. Z. Fan, A. Haghdoost, D. H. C. Du and D. Voigt, "I/O-Cache: A Non-volatile Memory Based Buffer Cache Policy to Improve Storage Performance," 2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, Atlanta, GA, 2015, pp. 102-111.
2. 2. Z. Fan, D. H. Du, and D. Voigt, “H-ARC: A non-volatile memory based cache policy for solid state drives,” in *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014, pp. 1–11.
3. 3. Y. Lee, J.-S. Kim, S.-W. Lee, and S. Maeng, “Zombie chasing: Efficient flash management considering dirty data in the buffer cache,” *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 569–581, 2015.
4. 4. S. Mittal and J. S. Vetter, “A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, May 2016.
5. 5. S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, “Improving flash-based disk cache with lazy adaptive replacement,” *ACM Transactions on Storage (TOS)*, vol. 12, no. 2, p. 8, 2016.
6. 6. C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, “Nitro: a capacity-optimized SSD cache for primary storage,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 501–512.
7. Multi-level Exclusive Caching: Ofﬂine Optimality and Why promotions are better than demotions, Proceeding, FAST’08 Proceedings of the 6th USENIX Conference on File and Storage Technologies, Article no.4.