Techniques to Improve Cache Utilization.

Abstract

Nowadays modern processors are using on chip caches of increasingly greater sizes. Caching techniques can enhance a system's overall performance by having a fast processor and a slow memory at the same level. One of the major strategies to improve cache utilization is replacement policy. Due to technological evolution in computer science, there shows a huge number of approaches and algorithms to improve cache performance and with the continuous evolution of computer science those approaches and algorithms got evolve. In this paper we will analyze various cache utilization techniques and as well as replacement algorithms.

Introduction

Cache is a high-speed memory. It is use to increase the speed and synchronizing with high-speed CPU. It is less costly than main memory. It is an extremely fast memory that uses as a buffer between RAM and the CPU. It holds frequently used data and instructions so that they can immediately called to the CPU when needed. It is use to reduce the average access time to access data from the main memory. There are number of different independent caches in a processor, which store instructions and data. A processor without cache takes plenty of time to execute the instructions. We always want to reduce the execution time of the instructions, for this reason we use cache memory in the CPU so that we can reduce the speed gap between slow memory like RAM or HDD or SSD and fast processor like modern days multicore processors with a significantly reduced cost. As we know that cache reduce the execution time but it is completely depending on its implementation structure or hardware implementation and software implementation. So, if we can utilize these two sectors then we can utilize the cache to its maximum potential capabilities. We will discuss on this matter in the following phrases.

Cache Implementation on Processor and Its work Process

In general, we categorize cache memory in three level like L1, L2, and L3 cache. L1 or level 1 cache is way faster than L2 and L3 or level 2 and level 3 caches also L1 cache is smaller than those 2 caches. We implement L1 cache next to the processor or CPU it is also known as on chip memory. We already know that L2 cache is slower than L1 cache but it does not slower than L3 cache. L2 cache is faster than L-3 cache and we implement L2 cache besides the level 1 or L1 cache. Then there comes the L3 cache which is slower than the previous caches L1 and L2. We implement this on close to RAM or main memory. As we nowadays multicore processor are being used in every where so the cache implementation makes an advancement. Previously we implement it on a single core processor now we implement the same structure in every core of a multicore processor but here every core has its own L1 and L2 caches but they share a same shared cache which is L3 cache close to the main memory. If we give attention to the implementation structure, we can point out that there present a hierarchy in the implementation based on the speed level. We call this hierarchical structure of processor. To get the visualization of hierarchical structure here we present figure.

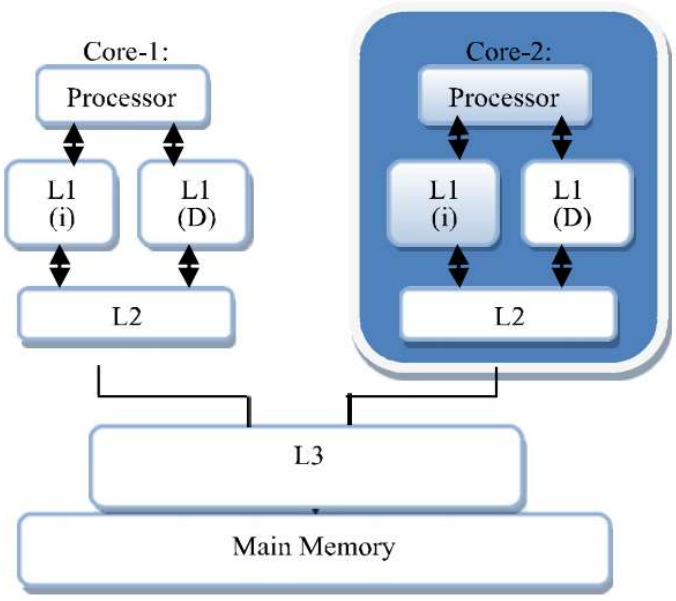

Fig: Hierarchical Structure of Cache Implementation on multicore processor.

As we already discussed on the cache implementation on modern days multicore processors now will have to know how a processor will work with having cache memory. In processor there has plenty of program counter from where its decided which instructions would be call to execute. To simply understand we assume there has a one program counter, now when any instruction needs to be called CPU first search the instruction in the cache memory if its found in the cache memory then we say hit cache but if the instruction does not fount in the cache memory it then search in the main memory for the instruction in this case we call it cache miss. In the case of cache miss the block belongs to the instruction will be copied into the cache memory so that next time for that block of instruction can be found in cache memory, that's how the whole process will continue. Now there present different mapping techniques to store the block of ram into the line of cache memory. Such as Direct mapping, Fully Associative mapping, and K-way Set Associative mapping. Now if we talk about cache miss there three types of cache misses.

I. Obvious cache miss which occurred when cache is initially empty and access at its initial phase.
II. Collision cache miss which occurred when cache is full and at the same time new instruction try to replace the existing one.
III. Capacity cache miss which occurred due to the small size of cache.

For the first type of cache miss we don't have anything in hand so utilization of cache memory does not have any effect on it that's why our concern is on type 2 and 3 of cache miss. Furthermore, due to the small capacity of cache memory it is required to replace its content according to the needs of program and precise time period. Now when we change our focus on replace content on cache memory, we find another important thing for Cache that is its replacement policy. Replacement policy helps to decide which instruction, data or content of cache memory will be replaced by the new content of main memory. It means by using replacement policies we can reduce the cache thus increase the cache hit and thus these policies will help to utilize cache to its maximum potential capabilities from the software side.

Replacement Algorithms

In the previous section we have discussed the cache replacement policy, to define these policies engineers from around the world has designed several algorithms that's known as Replacement Algorithm. In this section we will go through those algorithms that will help us to utilize the cache memory. Maybe we have several replacement algorithms but every algorithm does not have same efficiency. We can call an algorithm is more efficiency when it takes less time and cache misses is significantly low while the price of it as might as cheaper.

LRU (Least Recently Used) Algorithm):

This algorithm rejects the least recently used item from the cache to make space for the new data item. To gain this, a history of all data items that is which data item is used when it was kept. A variable known as Aging Bit is used to load this information, this algorithm shows better performance but the implementation cost is much more. Variants of LRU are the most beloved of all among all other algorithms. The key advantage of this policy is its simple execution, time, and space overhead is constant.

HLFU (History Least Recently Used) Algorithm):

Content of cache replace by LFU algorithm is performed by replacing the least frequently requested instructions while in HLFU which is an extension of LFU considers the History function in the cache replacement process. Respective LFU threshold is a linear function on the amount of currently used cache. The HLFU algorithm will replace the cached objects based on the Hist value as compared to the defined threshold in LFU.

LFU (Least Frequently Used) Algorithm:

It usually measures how often instructions have been used. The instructions had been used less, replace from the cache first. If all objects have the same frequency then this algorithm randomly replace any instruction.

GDS (Greedy Dual Size):

There comes a new term index which is measured according to the size of a file. The larger the file smaller its index. A file with the smallest index is replaced in this algorithm. Here we use Inflation value to keep track of frequently accessed files in the cache.

CAR (Clock with Adaptive Replacement) Algorithm:

The implementation of this algorithms very simple and it has a very low overhead on cache hits. It shows high performance and it also provides service of self-tuning. It is scan preventive which results in low space overheads that are less than 1 percent. CAR does not care for certain workloads.

ARC (Adaptive Replacement Cache):

It is easy to implement and running time independent from cache size. It has a low position overhead of approximately 0.75 percent of the size of the cache. It is scan resistant with self-tuning. It always balanced recency and frequency features by responding to changing access patterns. It divides cache into two queues that are handle by using clock that contains pages accessed only once, while the other contains the page which is accessed more than one time. It comes with constant complexity per request.

RR (Random Replacement) Algorithm:

It selects any of the instructions of cache randomly and replace them with the desired one. This algorithm doesn't need to keep track of the history of the data contents and it does not need any data structure. For this it consumes fewer resources and that’s why it is less costly.

LR+5LF Algorithm:

It is a combination of two popular replacement algorithm LRU and LFU. The problems that arrived in LRU and LFU are solved by this algorithm. The weighing problem of LRU AND LFU is solved here which reduces cache miss with a greater amount than LFU, LFU, and FIFO in L1 and L2 caches.

LLF (Lowest Latency First):

It maintains the average latency to a minimum by first expelling the object with the lowest download latency. Where the data is retrieved by executing a query against a relational database it gives the best output.

Conclusion

At the end of the discussion, we can conclude that the more technology advances the more speed is required to complete a task by machine or computers means computers need to take less and less time to execute millions and billions of instructions. For this generation we have caches to increase the speed of machines but having caches is not enough and we saw that. So, to gain the maximum performance from caches we need to utilize them and for utilization we have different techniques from the hardware and software aspects. In this report we mainly focused on the software side. Thus, we discussed more on the replacement algorithm which plays a greater role to activate the potential capabilities of cache memory to its maximum level.

References

[1] A. Z. M. A. M. S. Qaisar Javaid, "Cache Memory: An Analysis on Replacement," *HAL archives-ouverts,* October 4th 2017.

[2] H. Dybdahl, "Architectural Techniques to," *Doktoravhandlinger ved NTNU,* March 27, 2007.

[3] A. B. H. S. L. S. E. Z. Z. Lingda Li, "Tag-Split Cache for Efficient GPGPU Cache Utilization," *Pacific Northwest National Lab,* 2018.

[4] J. R. Srinivasan, "Improving Cache Utilisation," *University Of Cambridge,* June 2011.