Enhancing System Performance Utilizing Cache Architecture

[1]Harminder Singh, [2]Dr Sudesh Kumar, [3]Harpreet Kaur
[1,3]SGHS Khalsa College, Panjokhra Sahib, Ambala
[2]BRCMCET,Bahal

***Abstract*-Memory System is a crucial part of both the general purpose systems and the embedded systems. The use of embedded systems is increasing day by day in almost every field of life. Embedded systems are goal specific systems that are modeled for one or a few target applications. This requires the full customization of the system architecture so as to meet some power, performance and cost requirements. As the speed of the processor is almost ten times faster than that of the memory system, the memory system limits the processing power of the processor making the whole system inefficient. To overcome this speed mismatch between the processor and the main memory a smaller and faster memory component is fabricated in between these components. This smaller and faster memory component is referred to as the cache. A cache System is intended to speed up applications by providing means to manage cached data and instructions of various dynamic natures. Caching is a popular technique employed in a wide range of applications throughout computer systems in an attempt to hide the full cost of accessing some relatively slow device or connection by seeking a different capacity/speed tradeoff. This paper focuses on the techniques that can efficiently utilize the cache memory so as to improve the performance of the whole system.**

Keywords-Cache memory, optimization, cache utilization, memory system

I. INTRODUCTION

While designing the general purpose computer systems we have no prior knowledge about how the system would be employed in the real situation. So these systems are drafted for good average performance for a wide set of typical applications. However the embedded systems are the application oriented systems that are intended to solve some specific problems, so the architecture of these systems can be customized to meet the needs of a given application. Earlier embedded systems uses a single core processor to process the data. As an optimization attempt, to increase the processing power of the embedded systems multi core processors are fabricated, in which multiple CPU cores can process the data simultaneously. The memory subsystem of any embedded system also plays a crucial role in optimizing the working efficiency of the system. Latest CPU's uses various levels of memory hierarchy that they can access directly. Generally the memories fabricated closest to the CPU are the fastest memories, but they are costly and have very low storing capacity. Whereas the memories at a larger distance from the CPU are cheaper, but they are slower and can store comparatively larger data. The access time of the main memory that is DRAM(Dynamic Random Access Memory) is much larger than the CPU processing time. Thus the main memory limits the processing power of the CPU. To overcome this speed mismatch between the main memory and the cpu several small but fast memories are employed between them. The CPU uses cache memory to store the frequently used instructions by the program thus improving the overall speed of the system. These fast memories are called the cache memories. Memory hierarchy implemented with cache structures has received considerable attention from researchers and system designers. Cache Memories built into the CPU (also called the on chip memory) itself is referred to as Level 1 (L1) cache. Some cache memories are fabricated on a separate chip next to the CPU, these cache memories are called Level 2 (L2) cache. Some CPU's have both Level-1 and Level-2 cache built in and uses a separate chip as Level 3 (L3) cache. The L1 and L2 caches are comparatively faster than the L3 cache. The figure shows architecture of a typical multi core processor that contains on chip L1 and L2 cache for each core and the off chip L3 cache is shared among all the processor cores.

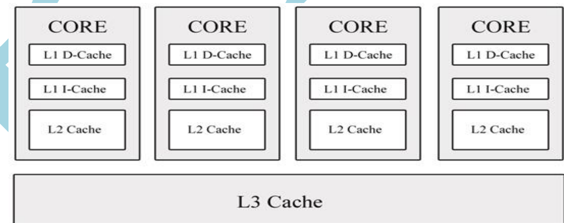

Fig 1.1- Organization of Cache memory in multi core processor systems

The main limitation of the cache memory is that it's capacity is much smaller than the main memory that it is backing up. So all the program's instructions or data can't be kept inside the cache memory .Some special techniques must be employed that selects the data needed frequently in the program and stores that data into the cache .The capacity of various cache memories used in Intel I7 processor is shown in the table below.

Table 1.1: Cache Configuration of Intel I7 Quad Core Processor

Name of Memory	Size	Associativity
L1 Data Cache	4 × 32 KBytes	8 way
L1 Instruction Cache	4 × 32 KBytes	8 way
L2 Cache	4 × 256 KBytes	8 way
L3 Cache	6MB	12 way

Thus optimizing the cache by choosing its various features like cache size, levels, cache line etc plays very important role in optimizing the overall embedded system.

II. CACHE OPTIMIZATION TECHNIQUES

Cache Memory Hierarchies

Size of the cache memories have a large impact on *both* off-chip latency and on-chip latency. In order to reduce the cache misses, cache capacity is partitioned into several levels. It is a well-known fact that larger caches take longer time to access where as smaller caches can be accessed in few CPU cycles. A small cache usually called a level-1 cache is placed very close to the CPU (on chip) . It ensures higher efficiency and lower latency. This on chip cache can be further divided into two different part. One part focuses on caching the data and the second part is used to cache the program instruction thus optimizing the performance of the system. These on chip memories can be accessed commonly in one or two CPU cycles. The size of these L1 cache memories is very small . For intel i7 processer the size of the L1 cache memory is 64 Kb which are divided into two parts of 32 kb each for the program data and program instructions as shown in the table 1.1. If the size of L1 cache is increased further the on chip latency will also increase which is not desirable to solve this problem a second level of cache memories Level-2 caches are used. Intel I7 processor contains approximately 256 kb of level-2 cache memory. In multi core processer environment each core has its own private L1 and L2 cache memories. To further optimize the efficiency of the system a third level of cache memory is Level -3 is used which is shared among all the processor cores. Intel I7 processor contains approximately 6 MB of L3 cache which is shared by all the four processor cores

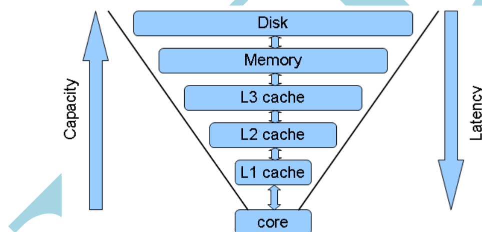

Fig 2.1- Capacity and Latency of memory hierarchy

Working of Locality of Reference

Due to a very low capacity, the cache memories can cache limited amount of data. When new data are loaded into cache the old data needs to be replace. The caches can be efficiently used if the cached data is reused in the near future before the data gets replaced by new data. This is defined by the principle of locality. Locality can be classified as temporal locality and spatial locality. Temporal locality enhances system performance. It conveys us whether memory locations recently accessed in a program are likely to be used again in the near future or not. A method that is called repeatedly in a short period of time is assumed to have a high temporal locality. The program can be slightly altered in order to use the benefits provided by the cache memories as shown below.

Table 2.1- Algorithms with low and high temporal locality

Algorithm-1: Low Temporal Locality	Algorithm-2 High Temporal Locality
Read data file *Generate Output* *Write Output file* *Read data file* *Generate Output* *Write Output file* *Read data file* *Generate Output* *Write Output file* ………. ………. ………. …… … *1000 times*	*Read data file* *Read data file* *Read data file* ………. ………. *1000 times* *Generate Output* *Generate Output* *Generate Output* ………. ………. *1000 times* *Write Output file* *Write Output file* *Write Output file* ………. ………. *1000 times*

Here two algorithm of the same program are given, one with low temporal locality and the other with the higher temporal locality. Algorithm-1 does not use the cache architecture efficiently because every time new instruction replaces the old cached instruction which results in a lot of cache misses. By slightly changing the algorithm we first read all the files which uses the same read instruction cached in the cache memory then read instruction is replaced with the generate output instruction and then with the write output instruction. This variant of Algorithm results in a very small number of cache misses. Hence cache memory is efficiently used.

III. CACHE MAPPING ASPECTS

Inside the cache memory the data is stored in the cache lines. Cache lines are used to store contiguous blocks of main memory. If the data requested by the CPU is present in any cache line then it is called a cache hit. In case of the cache hit, the data is fetched from the cache rather than main memory which saves several CPU cycles hence speed up the program. In case if the requested data or instruction is not found in the cache memory, the data needs to be searched in higher levels of memory hierarchy this process is known as cache miss. In case of cache miss the requested data is fetched from the main memory and rather than just processing the data it is also copied to one of the available cache line in cache memory. If no cache line is available to hold the data the old data in a cache line needs to be replaced by the new requested data. Depending upon how the data blocks are stored in the cache line there are three ways data can be mapped in the cache system. These are direct mapping, full associative mapping and set associative mapping. In full associative mapping, when a request for a particular data that needs to be fetched is made to the cache, the address of that particular data is then compared with each and every entry in the directory. If the requested data's address is found (*a cache hit*), the data is fetched from the corresponding location in the cache and returned to the processor for processing, On the other hand, if a cache *miss* occurs the address is compared in higher levels of the memory hierarchy. In a cache architecture with direct mapping technique, lower order line address bits points to the directory. Since more than one line addresses map into

the same location in the directory, the higher order line address bits (these are called tag bits) needs to be compared with the directory address to ensure a cache hit. If the comparison is invalid, then a cache miss occurs. The request address that is given to the cache system by the processor is actually partitioned into several parts, each of this part has a different aspect in accessing data. The working of the set associative cache techniques is somewhat similar to the direct-mapped cache. Some of the bits from the line address helps to reach a particular cache directory. However, after reaching any directory we have multiple choices: 2, 4, or more complete line addresses are enclosed in the directory. Each of these line addresses corresponds to a unique location in a sub-cache. By collecting all these sub-caches we get the total cache array[7]. In a set associative cache, all of these sub-arrays can be accessed in parallel, along with the cache directory. If any of the entries in the cache directory matches the address requested by the processor, then there is a cache hit, In this case the particular sub-cache array is fetched and contents are transferred back to the processor.

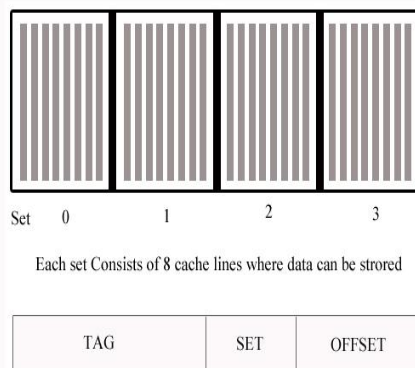

Fig 3.1 - An overview of 8-way set associative cache

IV. IMPROVED TECHNIQUES FOR DATA ACCESS

To improve the temporal locality of the data and effectively utilizing the cache we can change the order in which the nested loops in the program are executed. In case of loop the successive iterations of the loop often uses the same data word or the adjacent data words in the memory.

Table 4.1- Algorithms demonstrating loop interchange

Algorithm 4.1	Algorithm 4.2
1. Int arrA[n] ; 2. for j=1 to n do 3. for i=1 to n do 4. arrA[i,j] = - arrA[i,j]; 5. end for 6. end for	1. Int arrA[n] ; 2. for i=1 to n do 3. for j=1 to n do 4. arrA[i,j] = - arrA[i,j]; 5. end for 6. end for

So by slight modification in the loop the cache hit rates can be significantly increased. Consider two algorithms shown below. Both of these algorithms does the exactly same calculation but the order in which they access the array elements are different. These different access ordering increases the hit rate of cache because very less amount of replacements will be needed in the second algorithm.

Loop Interchange

Loop interchange is the data access optimization techniques that can optimize the nested loops in the program by altering the ordering in which the loops gets processed.

If the arrays are stored in the row- major order in the main memory, then the Algorithm 2.1 will result in lot of cache misses because this algorithm is processing the arrays in a column- major order as shown in the figure 4.1. In this case the cached data shown by the shaded area is not used by the program hence these loops are not optimized.

Fig 4.1 Processing array in Colomn Order

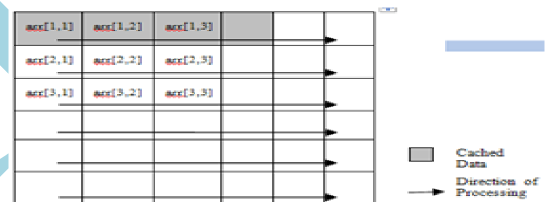

Fig 4.1 Processing array in Row Major Order

By interchanging the loop variables i and j as shown in Fig 4.2 the processing of the array can be performed in the row-major order which will then take the benefit of cached data thus decreasing the hit rate.

Blocking

A very important change in the algorithms is the involvement of blocking data structures that can fit in the cache. By organizing the data memory accesses, we can populate the systems cache memory with a small part of the larger data structure. After loading this small block in the memory the idea is to maximize the use of this loaded block. By reusing this data block available in the cache memory we can reduce frequency of memory access. Blocking of data is an optimization technique which can help in avoiding memory bandwidth bottlenecks in a number of applications that use large data structures [5]. The main idea behind blocking is to first examine the inherent data in the application that how it can be reused by ensuring that data remains in cache memory during multiple requests. Blocking technique can be applied one dimensional, two dimensional or three dimensional spatial data structures. Some applications that uses the iterative code can further take benefit from blocking during

multiple iterations (this is commonly known as temporal blocking) to diminish bandwidth bottlenecks. Blocking can be implemented by changing the program slightly. It involves a combination of loop interchange and loop splitting. In the source code of most of the applications, blocking techniques is effectively accomplished by the programmer by constructing the right source changes with some slight parameterization of the block-factors that can speed up the application. Consider a case of matrix multiplication, let two matrices A and B are multiplied and the results are stored in the third matrix. To calculate one element of matrix c requires accessing an entire row of matrix *a* and an entire column of matrix *b*. To calculate an entire column of matrix *c* we need to access all the rows of matrix *a*, so matrix *a* is accessed *n* times, once for each column of *c*. Likewise, every column of matrix *b* must be read again for each element of matrix *c*, so matrix *b* is read *m* times.

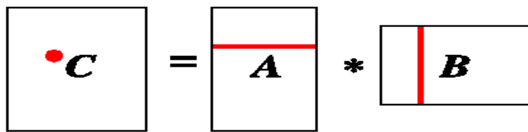

Fig 4.3 - Simple matrix multiplication

If matrix *a* and matrix *b* does not fit in the cache memory, the already cached rows and columns are more likely to be replaced by the new ones, so there may be a little reuse of cache memory. As a result, this process will require fetching of data in from main memory, *n* times for matrix *a* and *m* times for matrix *b*.

Because the operations on matrices in Fig 4.3 are unordered (not order dependent), this problem with numerous number of cache misses can be fixed by considering the matrices in smaller sub-blocks as shown in Fig 4.4. In blocking, a block of matrix *c* can be calculated efficiently by taking the dot-product of a row block of matrix *a* with a column-block of matrix *b*. The dot-product of these matrices consists of multiplication of series of sub-matrices. When three blocks of the matrices, one from each matrix, all are cached in cache memory simultaneously, the elements of those blocks need not to be read from memory again and again.

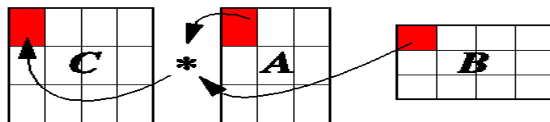

Fig 4.4- Matrix multiplication using cache blocking technique

Loop Fusion- Merging Adjacent Loops

In order to improve run-time performance of the system and reduce loop overhead, some adjacent loops can be combined into one loop. This technique is called loop fusion. Loop fusion or loop jamming is a technique that transforms some loops in the program. This technique replaces multiple loops with a single loop. This techniques is possible only when two or more loops iterate over the same range of data and does not reference each other's data. Loop fusion technique works by increasing the number of code statements and accessed arrays within a loop. Loop fusion is based upon temporal locality[6]. It optimize the program by efficiently implementing the data access by reducing the time interval between the requesting of the same data, hence it increases the chances of the data being retained in the cache and increasing hit ratio frequency.

Table 4.2- Algorithms demonstrating loop fusion

Algorithm 4.3	*Algorithm 4.4*
1) int A[n], B[n] ; *2) for i=1 to n do* *3) A[i] = A[i] + 3;* *4) end for* *5) for j=1 to n do* *6) B[j] = B[j]+4;* *7) end for*	*1) int A[n], B[n] ;* *2) for i=1 to n do* *3) A[i] = A[i] + 3;* *4) B[j] = B[j]+4;* *5) end for*

Although loop fusion helps in lowering the loop overhead, it does not always ensure the improved run-time performance. On some of the system architectures, two different loops may actually perform better than one single loop. In these cases, a single loop may be changed into two different loops, which is called Loop fission.

V. CONCLUSION

There is a large speed mismatch between the CPU and the main memory of the computer system. Thus the processing speed of the CPU is limited by the speed of main memory. To overcome this speed disparity cache memories are incorporated in computer system architectures to speed up the system. Caches are small but fast memories that are fabricated very close to the processor. Caches speed up the program execution by storing the recently used data from the main memory. Various levels of caches are used to get even better results. Some caches are shared among various cores of the processor whereas some caches are private to each processor core. Having cache fabricated in the system it is still not necessary that the cache is properly utilized. Therefore some techniques are discussed that must be used while programming in order to speed up the execution of the program by the CPU. In some cases by slightly changing the loop variables in a nested loop cached data can be more efficiently used and cache hit ratio can be improved. Further by using blocking complex calculations like matrix multiplication can be effectively performed by caching the smaller block of the matrix in the memory and then reusing that block for several iteration. At last loop fusion can reduce the loop overhead by performing the calculations of two adjacent loops within a single loop.

REFERENCES

[1] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler transformation for high-performance computing. ACM Computing Surveys, 26:345-420, December 1994.

[2] D.F. Bacon, et al. A Compiler framework for restricting data declaration to enhance cache and TLB effectiveness. In CASCON 194, pages 270-282, Toronto, Canada,1 994

[3] M.J Wolfe. Optimizing super compliers for super computers. The MIT Press, Cambridge MA, 1989.

[4] R. Allen and K. Kennedy. Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, San Francisco, California, USA, 2001

[5] Amandas (Intel), Compiler methodology for Intel MIC Architecture, November 7,2013, https://software.intel.com/en-us/articles/cache-blocking-techniques

[6] Compiler optimizations, http://www.compileroptimizations.com/category/loop_fusion.htm

[7] Computer Organization and Architecture, http://gyan.fragnel.ac.in/~surve/COA/Memory/Memory_Cache.html

[8] N. Ahmed, N. Mateev, and K. Pingali. Tiling ImperfectlyNested Loop Nests. In Proc. of the ACM/IEEE Supercomputing Conference, Dallas, Texas, USA, 2000

IJRRA