International Journal Of Scientific Research And Education
||Volume||1||Issue|| 6 ||Pages|| 119-125 ||2013|| ISSN (e): 2321-7545
Website: http://ijsae.in

CACHE MEMORY OPTIMIZATION

JITENDER SINGH YADAV, MOHIT YADAV* ANKIT JAIN*

Email-jitender1191m@gmail.com

mohit12yadav@yahoo.com

ankitjain199256@gmail.com

ABSRTACT

This research paper investigates the cache memory and its various Optimizing techniques. The early beginning part of the paper makes you familiar with the term cache. Further ahead, the paper covers the importance of cache memory in microprocessors. Then, a detailed cache organization is explained with the help of appropriate diagrams for easy understanding. Details of cache optimization methods implemented by the cache are also undertaken in this paper. In order to allay the impact of the growing gap between CPU speed and main memory performance, today's computer architectures implement hierarchical memory structures. The idea behind this approach is to hide both the low main memory bandwidth and the latency of main memory accesses which is slow in contrast to the floating point performance of the CPUs. Usually, there is a small and expensive high speed memory sitting on top of the hierarchy which is usually integrated within the processor chip to provide data with low latency and high bandwidth; i.e., the CPU registers. Moving further away from the CPU, the layers of memory successively become larger and slower. The memory components which are located between the processor core and main memory are called cache memories or caches. this paper focuses on optimization techniques reviews for enhancing cache performance.

Thus, going through this paper one will end up with a good understanding of cache and its Optimizing techniques.

***KEYWORDS**:-Cache. Memory. Hierarchical memory , Latency , Bandwidth.*

1. INTRODUCTION

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. Whenever data must be passed through the system bus, the data transfer speed slows to the motherboard’s capability. The CPU can process data much faster by avoiding the bottleneck created by the system bus.

Cache built into the CPU itself is referred to as *Level 1 (L1)* cache. Cache that resides on a separate chip next to the CPU is called *Level 2 (L2)* cache. Some CPUs have both L1 and L2 cache built-in and designate the separate cache chip as *Level 3 (L3)* cache. Cache that is built into the CPU is faster than separate cache, running at the speed of the microprocessor itself. However, separate cache is still roughly twice as fast as Random Access Memory (RAM). Cache is more expensive than RAM, but it is well worth getting a CPU and motherboard with built-in cache in order to maximize system performance.

so, cache plays very important role in processing of cpu to be fast and for this

the optimization of cache is also very necessary. In this various optimization

techniques are to be reviewed.

2.ARCHITECTURE AND PERFORMANCE EVALUATION OF CACHES

2.1 Organization of Cache Memories

Typically, a memory hierarchy contains a rather small number of registers on the chip which are accessible without delay. Furthermore, a small cache , usually called level one (L1) cache | is placed on the chip to ensure low latency and high bandwidth. The L1 cache is often split into two separate parts; one only keeps data, the other instructions. The latency of on{chip caches is commonly one or two cycles. The chip designers, however, already face the problem that large on{chip caches of new microprocessors running at high clock rates cannot deliver data within one cycle since the signal delays are too long. Therefore, the size of on{chip L1 caches is limited to 64 Kbyte or even less for many chip designs. However, larger cache sizes with accordingly higher access latencies start to appear.

The L1 caches are usually backed up by a level two (L2) cache. A few years ago, architectures typically implemented the L2 cache on the motherboard, using SRAM chip technology. On{chip L2 caches are usually smaller than 512 Kbyte and deliver data with a latency of approximately 5 to 10 cycles. If the L2 caches are implemented on{ chip, an o_{chip level three (L3) cache may be added to the hierarchy. O_{chip cache sizes vary from 1 Mbyte to 16 Mbyte. They provide data with a latency of about 10 to 20 CPU cycles.

2.2 LOCALITY OF REFERENCES.

Because of their limited size, caches can only hold copies of recently used data or code. Typically, when new data are loaded into the cache, other data have to be replaced. Caches improve performance only if cache blocks which have already been loaded are reused before being replaced by others. The reason why caches can substantially reduce program execution time is the principle of locality of references which states that recently used data are very likely to be reused in the near future. Locality can be subdivided into temporal locality and spatial locality. A sequence of references exhibits temporal locality if recently accessed data are likely to be accessed again in the near future. A sequence of references exposes spatial locality if data located close together in address space tend to be referenced close together in time.

2.3 ASPECTS OF CACHE ARCHITECTURES.

In this section, we brief review the basic aspects of cache architectures. Data within the cache are stored in cache lines. A cache line holds the contents of a contiguous block of main memory. If data requested by the processor are found in a cache line, it is called a cache hit. Otherwise, a cache miss occurs. The contents of the memory block containing the requested word are then fetched from a lower memory layer and copied into a cache line.

2.4 MEASURING AND SIMULATING CACHE BEHAVIOR.

In general, profiling tools are used in order to determine if a code runs efficiently, to identify performance bottlenecks, and to guide code optimization. One fundamental concept of any memory hierarchy, however, is to hide the existence of caches. This generally complicates data locality optimizations; a speedup in execution time only indicates an enhancement of locality behaviour, but it is no evidence. To allow performance profiling regardless of this fact, many microprocessor manufacturers add dedicated registers to their CPUs in order to count certain events. Typical quantities which can be measured include cache misses and cache hits for various cache levels, pipeline stalls, processor cycles, instruction issues, and branch mis-predictions. Another approach towards evaluating code performance is based on instru-mentation. Profiling tools such as GNU prof and ATOM insert calls to a monitoring library into the program to gather information for small code regions. The library routines may either include complex programs themselves (e.g., simulators) or only modify counters. Instrumentation is used, for example, to determine the fraction of the CPU time spent in a certain subroutine.

3. BASIC TECHNIQUES FOR IMPROVING CACHE EFFICIENCY.

A number of cache optimization techniques that were implemented in single core processors were successfully implemented in multi core processors. Multi-level cache with the current structure of two-level has been implemented since the very first multi core processor. In this configuration, the first-level cache is private to each core and coherence is maintained between them with MESI or MOESI protocols . The second-level cache has been implemented with different design options in various architectures. In general, the second-level cache is shared among all cores with a number of optimizations to be discussed in this section. One of the major innovations in the design of the second level cache is NUCA (Non Uniform Cache Architecture)cache . The reason for building NUCA organization is that the second-level cache is made much larger than the first-level to satisfy the design requirements of multi-level cache. The result is a slower access time with the increasing cache size. This problem is resolved by dividing the cache into banks. The context of a specific core is kept in a bank physically closer to it gaining improvement in the speed of access. A number of variants of NUCA have evolved over the last few years with many innovations implemented in current generation processors.

3.1 DATA ACCESS OPTIMIZATIONS.

Data access optimizations are code transformations which change the order in which iterations in a loop nest are executed. The goal of these transformations is mainly to improve temporal locality. Moreover, they can also expose parallelism and make loop iterations vectorizable. Note that the data access optimizations we present in this section maintain all data dependencies and do not change the results of the numerical computations1.Usually, it is difficult to decide which combination of transformations must be applied in order to achieve a maximum performance gain. Compilers typically use heuristics to determine whether a transformation will be effective or not. Loop transformation theory and algorithms found in the literature typically focus on transformations for perfectly nested loops [1]; i.e., nested loops where all assignment statements are contained in the innermost loop.

Loop Interchange. This transformation reverses the order of two adjacent loops

in a loop nest . Generally speaking, loop interchange can be applied if the order of the loop execution is unimportant. it can improve locality by reducing the stride of an array{based computation. The stride is the distance of array elements in memory accessed within consecutive loop iterations. Upon a memory reference, several words of an array are loaded into a cache line. If the array is larger than the cache, accesses with large stride only use one word per cache line. The other words which are loaded into the cache line are evicted before they can be reused. Loop interchange can also be used to enable and improve vectorization and parallelism, and to improve register reuse. The different targets may be convicting. For example, increasing parallelism requires loops with no dependencies to be moved outward, whereas vectorization requires them to be moved inward.

Loop Fusion. Loop fusion is a transformation which takes two adjacent loops that have the same iteration space traversal and combines their bodies into a single loop . Loop fusion | sometimes also called loop jamming | is the inverse transformation of loop distribution or loop fission which breaks a single loop into multiple loops with the same iteration space. Fusing two loops results in a single loop which contains more instructions

in its body and therefore offers increased instruction level parallelism.

Loop Blocking. Loop blocking (also called loop tiling) is a loop transformation

which increases the depth of a loop nest with depth n by adding additional loops

to the loop nest. The depth of the resulting loop nest will be anything from n+1

to 2n. Loop blocking is primarily used to improve data locality by enhancing

the reuse of data in cache.

3.2 DATA LAYOUT OPTIMIZATIONS.

Data access optimizations have proven to be able to improve the data locality of applications by reordering the computation, as we have shown in the previous section. However, for many applications, loop transformations alone may not be sufficient for achieving reasonable data locality. Especially for computations with a high degree of conflict misses3, loop transformations are not effective in improving performance.

Data layout optimizations modify how data structures and variables are arranged in memory. These transformations aim at avoiding effects like cache conflict misses and false sharing. They are further intended to improve the spatial locality of a code. Data layout optimizations include changing base addresses of variables, modifying array sizes, transposing array dimensions, and merging arrays. These techniques are usually applied at compile time, although some optimizations can also be applied at runtime.

Array Padding. If two arrays are accessed in an alternating manner and the data structures happen to be mapped to the same cache lines, a high number of conflict misses are introduced.

Inter array padding inserts unused variables(pads) between two arrays in order to avoid cross interference. Introducing pads modifies the offset of the second array such that both arrays are then mapped to different parts of the cache.

Intra array padding, on the other hand, inserts unused array elements between rows of a multidimensional array by increasing the leading dimension of the array; i.e., the dimension running fastest in memory is increased by a small number of extra elements. Which dimension runs fastest in memory depends on the programming language. For example, in Fortran77 the leftmost dimension is the leading dimension, whereas in C/C++ the rightmost dimension runs fastest.

Array merging is best applied if elements of different arrays are located far apart in memory but usually accessed together. Transforming the data structures will change the data layout such that the elements become contiguous in memory.

Array Transpose. This technique permutes the dimensions within multidimensional arrays and eventually reorders the array .This transformation has a similar effect as loop interchange

4. 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 behaviour. 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.