*I.J. Intelligent Systems and Applications,* 2012, 4, 26-32
Published Online April 2012 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijisa.2012.04.04

# Modern CPU's Memory Architecture – A Programmer's Outlook

Eisha Akanksha[1]
[1]PG Scholar, Department of ECE, M.V.J College of Engineering, Bangalore, India.
Email:eisha.b@gmail.com

I.Hameem Shanavas[2], Vadamalai Nallusamy[3]
[2]Assistant Professor, [3]Associate Professor, Department of ECE, M.V.J College of Engineering, Bangalore, India
Email:hameemshan@gmail.com;nallu1910@gmail.com

***Abstract*—This paper provides the overview of modern computer's memory usage and how the programmers can exploit the architecture to best utilize for enhanced design of software programs. The paper talks about the main memory, cache memory and memory interfaces with CPU's. We also talk about the basic problems of cache in multicore CPU's. Cache Synchronization algorithms are also touches upon. The optimization techniques using cache coloring avoidance is too talked about. The paper does not talk about the NUMA (non uniform memory architecture that is beyond that requires different programming skills. For general understanding we assume a computer processor has MMU enabled. The techniques for smaller computer AKA embedded processor without virtual memory and TLB will differ in optimization approach. The paper is inspired by "What every computer scientist should know about floating point numbers" and "What every programmer should know about memory".**

***Index Terms*—TLB, Cache, DRAM, SRAM, Cache coherency, Set associative cache.**

## I. INTRODUCTION

MOST CPU's today uses the memory on multiple level. Generally the memory at the proximity of CPU is costly and less, whereas the memory at the distance (wire distance) is bigger, slower and cheap[1]. Today getting the computer in market with 8GB DRAM is cheap, but L1/L2 cache of such computer is very small in terms of 10's of KB's and few MB's respectively. The access time of L1 (that is generally SRAM) is few cycles whereas L2 is few 10's cycles and accessing main memory is considered a bad programming if accessed too frequently. The access time is huge and in terms of 100's of cycles. So optimizing the code to run and access L1 Instruction and Data cache is the simplest way to optimizing the code. But today's computers are used as general purpose and typical programs such as firefox and internet explore consumes memory about 50 MB. While typing the document at MAC OS X 10.6, MS word is taking 51M main memory. Unix top[2] command gives the memory usage of the currently running programs. If the developer of the such a software assumes and memory is free, then the computer would be so slow as the mechanical calculator of past[4]. Similar work has been done from floating point perspective. Most of the style of work is inspired by the work of [5].

The first technique is to access the frequently accessed data so that CPU can find the data from the L1 cache, in the worst case from L2 case. There should be least requirement to go to the main memory to fetch the data. But given the fact, "This is not as easy as it thought". I present the some concept where such an access can be made as fast as possible. Using cache never came free. There are penalties associated this this. Such as, even if there are lots of L1 cache is free, but a poorly written program cans his associativity miss in the cache. Such a problem is known as "Cache color" [6].

With modern CPU's there lies the problems with DMA. For DMA to happen and CPU to use the latest information of DMA'd data, the cache has to coherent. Else CPU will keep using the old data that leads to very subtle bugs in software. Design of coherent memory is very difficult from software and hardware point of view. Most modern and efficient OS such as Linux provides the way to sync cache and main memory. But that flushes the caches and is very costly operation in terms of time complexity. Unlike storage subsystems, removing the main memory as a bottleneck has proven much more difficult and almost all solutions require changes to the hardware. Today these changes mainly come in the following forms:

- RAM hardware design (speed and parallelism).
- Memory controller designs.
- CPU caches.
- Direct memory access (DMA) for devices.

For the most part, this document will deal with CPU caches and some effects of memory controller design. In the process of exploring these topics, we will explore DMA and bring it into the larger picture. However, we will start with overview of the design for today's commodity hardware. This is a prerequisite to understanding the problems and the limitations of efficiently using memory subsystems. We will also learn about, in some detail, the different types of RAM and illustrate why these differences still exist.

Section II describes the definition of Cache memory, Set associative cache, Cache coloring, dynamic Random access memory. Section III discusses the Hardware model present today. Section IV details the different approaches based on programmers' perspective. Section V gives the Memory performance tool .

## II. Definitions

### A. Cache Memory

*Cache Memory* is a cache used by the central processing unit of a computer to reduce the average time to access memory. Cache memory is access via memory bus that has limited bandwidth. The width of bus is CACHELINE.

### B. Set-associative cache

Set-associative cache is fully associative cache, and direct mapped cache. It's considered a reasonable compromise between the complex hardware needed for fully associative caches (which requires parallel searches of all slots), and the simplistic direct-mapped scheme, which may cause collisions of addresses to the same slot (similar to collisions in a hash table). [7]

### C. Cache coloring

Cache coloring (also known as page coloring) is the process of attempting to allocate free pages that are contiguous from the CPU cache's point of view, in order to maximize the total number of pages cached by the processor.

### D. Dynamic random-access memory

This is a type of random-access memory that stores each bit of data in a separate capacitor within an integrated circuit. The capacitor can be either charged or discharged; these two states are taken to represent the two values of a bit, conventionally called 0 and 1. Since capacitors leak charge, the information eventually fades unless the capacitor charge is refreshed periodically. Because of this refresh requirement, it is a *dynamic* memory as opposed to SRAM and other *static* memory.

### E. Magnetic RAM( MRAM)

The Magnetic Random Access Memory (MRAM) is considered one of the potential candidates that will replace current on-chip memories (RAM, EEPROM, and flash memory) in the future[8]. The MRAM has a high speed and also it does not need high supply voltage for Read & Write operations, so it has the advantages of RAM and flash memory, making it a potentially good choice for SOC. The testing of MRAM, however, has not been fully investigated.

## III. Hardware of Today

Scaling these days is most often achieved horizontally instead of vertically, meaning today it is more cost-effective to use many smaller, connected commodity computers instead of a few really large and exceptionally fast (and expensive) systems.X86 family of computer[5] (including Pentium, Sandy bridge) uses Northbridge and Southbridge to interface various peripherals. Today the PCI, PCI Express, SATA, and USB buses are of most importance, but PATA, IEEE 1394, serial, and parallel ports are also supported by the Southbridge. Older systems had AGP slots which were attached to the Northbridge. This was done for performance reasons related to insufficiently fast connections between the Northbridge and Southbridge. However, today the PCI-E slots are all connected to the Southbridge.

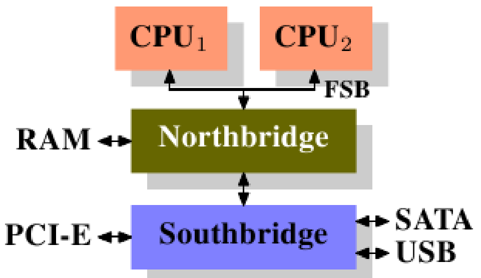

Figure 1: Northbridge and south bridge of inter computer. FSB is front side bus to interface CPU. All the buses have limited bandwidth to transfer data. All communication with RAM must pass through the Northbridge. Communication between a CPU and a device attached to the Southbridge is routed through the Northbridge.

To overcome memory bus bandwidth, following scheme is deployed.

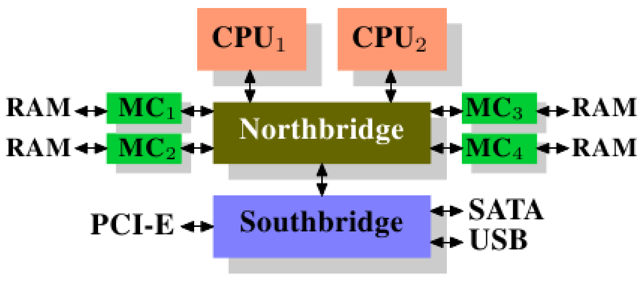

Figure 2: Scheme to overcome single bus in X86 family of CPU's.

Using multiple external memory controllers is not the only way to increase memory bandwidth. One other increasingly popular way is to integrate memory controllers into the CPUs and attach memory to each CPU. This architecture is made popular by SMP systems based on AMD's Opteron processor.

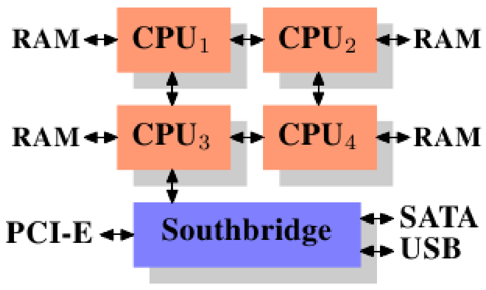

Figure 3: AMD opteron processor scheme to reduce the RAM access time.

The advantage of this architecture is that more than one memory bus exists and therefore total bandwidth increases. This design also supports more memory[9,10]. Concurrent memory access patterns reduce delays by simultaneously accessing different memory banks.

## IV. Programmer's Approach

*Technique 1: Cache Access*

The most important improvements a programmer can make with respect to caches are those, which affect the level 1 cache. We will discuss it first before including the other levels. Obviously, all the optimizations for the level 1 cache also affect the other caches. The theme for all memory access is the same: improve locality (spatial and temporal) and align the code and data.

If the code spans the data in multiple cachelines, the results are similar to following figure.

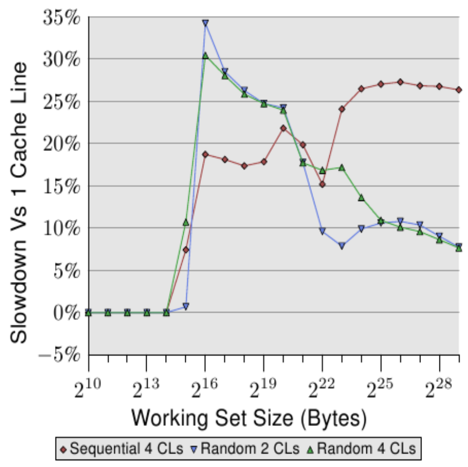

Figure 4: A CPU has 8K of cache memory. We see the slow down after this set. If the working memory increases the results starts worsening.

Other point a programmer can see, is to avoid the access of unaligned address. The graph shows the slowdown the program incurs because of the unaligned accesses. The effects are more dramatic for the sequential access case than for the random case because, in the latter case, the costs of unaligned accesses are partially hidden by the generally higher costs of the memory access. In the sequential case, for working set sizes which do fit into the L2 cache, the slowdown is about 300%.For x86 binaries gcc has support for relaxed stack alignment requirements:

```
-mpreferred-stack-boundary=2
```

If this option is given a value of N, the stack alignment requirement will be set to $2^N$ bytes.

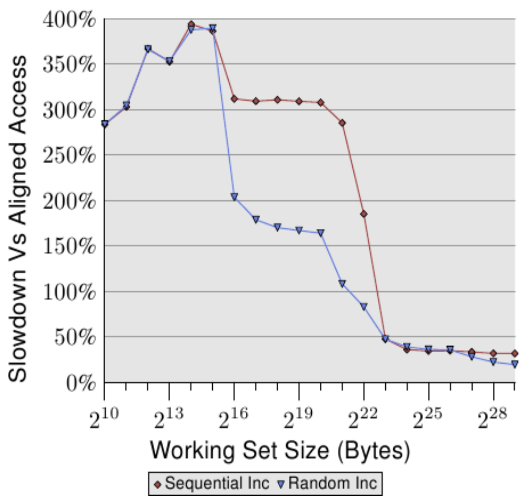

Figure 5: Slow down and aligned access is seen as size of working set memory. Working set memory is in main memory. Here we can figure out working set size in L1 size does not bring much degradation.

*Technique 2: Understanding and using cache associativity*

For programmer's associativity is something worth paying attention to. A bad data structure lead to cache color problem that leads to associativity miss of cache. Laying out data at boundaries that are powers of two happens often enough in the real world, but this is exactly the situation that can easily lead to the above effects and degraded performance. Unaligned accesses can increase the probability of conflict misses since each access might require an additional cache line.

If this optimization is performed, another related optimization is possible, too. AMD's processors [1][12], at least, implement the L1d as several individual banks. The L1d can receive two data words per cycle but only if both words are stored in different banks or in a bank with the same index. The bank address is encoded in the low bits of the virtual address as shown in Figure 6.6. If variables that are used together are also stored together the likelihood that they are in different banks or the same bank with the same index is high.

Compilers have options to enable levels of optimization; specific optimizations can also be individually enabled. Many of the optimizations enabled at high optimization levels (-O2 and -O3 for gcc) deal with loop optimizations and function inlining. In general, these are good optimizations.

*Technique 3:Using GCC (compilers standard primitives)*

GCC provides the primitive to optimizing the branch prediction. Most operating systems provide macros "LIKELY, UNLIKELY". Based on logic design of the program, a likely or unlikely can be placed to reduce the branch overhead. Intel profiler and operating systems promitives provides the way to measure the branch mispredicts. pmcstat can be used on FreeBSD OS, vTunes on Windows. Such optimization of memory can lead to drastic improvement in program performance.

GCC provides built in expect that is lower level expansion of LIKELY/ UNLIKELY macro. Please refer to your operating system macros and primitives.

*Technique 4: Optimizing Level 2 and Higher Cache Access*

- cache misses are always very expensive. While L1 misses (hopefully) frequently hit L2 and higher cache, thus limiting the penalties, there is obviously no fallback for the last level cache.
- L2 caches and higher are often shared by multiple cores and/or hyper-threads. The effective cache size available to each execution unit is therefore usually less than the total cache size.

To avoid the high costs of cache misses, the working set size should be matched to the cache size. If data is only needed once this obviously is not necessary since the cache would be ineffective anyway. We are talking about workloads where the data set is needed more than once. In such a case the use of a working set which is too large to fit into the cache will create large amounts of cache misses which, even with prefetching being performed successfully, will slow down the program.

*Technique 4: Optimizing TLB Usage*

There are two kinds of optimization of TLB usage. The first optimization is to reduce the number of pages a program has to use. This automatically results in fewer TLB misses. The second optimization is to make the TLB lookup cheaper by reducing the number higher-level directory tables that must be allocated. Fewer tables' means less memory usage that can result is higher cache hit rates for the directory lookup.

The first optimization is closely related to the minimization of page faults. While page faults usually are a one-time cost, TLB misses are a perpetual penalty given that the TLB cache is usually small and it is flushed frequently. Page faults are orders of magnitude more expensive than TLB misses but, if a program is running long enough and certain parts of the program are executed frequently enough, TLB misses can outweigh even page fault costs. It is therefore important to regard page optimization not only from the perspective of page faults but also from the TLB miss perspective. The difference is that, while page fault optimizations only require page-wide grouping of the code and data, TLB optimization requires that, at any point in time, as few TLB entries are in use as possible.

*Technique 5: Hardware Prefetching*

The trigger for hardware prefetching is usually a sequence of two or more cache misses in a certain pattern. These cache misses can be to succeeding or preceding cache lines. In old implementations only cache misses to adjacent cache lines are recognized. With contemporary hardware, strides are recognized as well; meaning that skipping a fixed number of cache lines is recognized as a pattern and handled appropriately.

It would be bad for performance if every single cache miss triggered a hardware prefetch. Random memory accesses, for instance to global variables, are quite common and the resulting prefetches would mostly waste FSB bandwidth. This is why, to kick start prefetching, at least two cache misses are needed. Processors today all expect there to be more than one stream of memory accesses. The processor tries to automatically assign each cache miss to such a stream and, if the threshold is reached, start hardware prefetching. CPUs today can keep track of eight to sixteen separate streams for the higher level caches [14] The prefetching gives the very little improvement. It is recommended to use this technique as the final improvement.

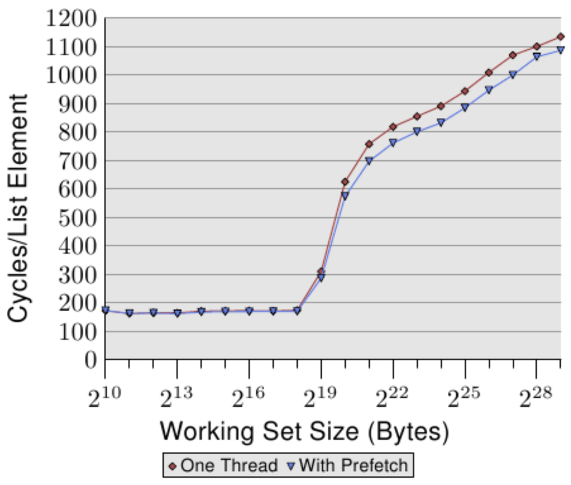

Figure.5. Hardware prefetching

gcc today is able to emit prefetch instructions automatically in one situation. If a loop is iterating over an array the following option can be used:

```
-fprefetch-loop-arrays
```

## V. Memory performance tools

As discussed previously FreeBSD operating system comes with pmcstat [1] to figure out the memory usages. It used hwpmc kernel object to do profiling. Similar ways, Linux provides profile utility [16]. Pfmon is another Linux tool providing performance indictor.

Especially due to the multi-core paradigm, the memory and network components of a machine are becoming critical resources. Several solutions have been proposed to decrease latency such as memory hierarchies and multi-threading. Larger or multiple interconnect fabrics and different network topologies are continuously being proposed and tested to increase bandwidth utilization. However, due to the ever increasing number of cores on a chip, available bandwidth has to be rationed very carefully. Moreover, there are the architectural limitations of the memory and network themselves like memory physical boundaries, router's buffer length, etc. Due to these factors, possible architectural bottlenecks might appear. Complicating the issue, certain algorithmic bottlenecks might not become apparent until certain resource limits are reached. A bottleneck, or hot-spot, occurs when a resource is oversubscribed by an application or job. In the case of the network, an architectural bottleneck might represent a routing node in which its buffers are full. In the case of the memory, this might be contention on the same memory node, DIMM, rank or page. This memory behavior may not be apparent to processor centric tools and it might not even appear until a large data set is used (e.g. when bank segments, or other memory structural barriers, are crossed). Among the several parallel designs, the Cray XMT is a very special type of shared memory machine. It consists of massive multi-threaded processors with no data caches, fine grained synchronization in memory and a large global shared memory, all arranged in a 3D torus network. As a highly parallel multi-threaded machine with shared memory, it provides a great opportunity to study future massive multi-core shared memory machines. These systems will have an ever increasing number of cores and a pool of shared memory. Moreover, the Cray XMT can be used to experiment with communication and shared memory to a scale that is not possible in current shared memory machines. Due to these characteristics the Cray XMT can be used as a prototype for future shared memory machines. Moreover, the XMT uses a very light UNIX derivate OS. This OS is optimized for parallel execution environment and introduces very little noise. This means low OS noise and jitter. These features make the XMT a very attractive machine for this tool.

## VI. Conclusion

We discussed about the various bottleneck and how to measure those. Performance improvement is rather iterative process. A good understanding of operating system, kernel internals, and virtual memory architecture is needed to optimize the programs holistically. More to add, these technique cannot compete the algorithm optimization. A programmer needs to understand the asymptotic complexity of program before even going for this level of optimization.

## References

[1] Techyon Semiconductor "DRAM Pricing – A White Paper" , September 3,2002.

[2] Manual pages of Linux, FreeBSD.

[3] Prentice Hall of India, Cormen, Leiserson, Rivest "Introduction to Algorithms" 3rd edition. ISBN-978-81-203-4007-7

[4] Ulrich Drepper, "What Every Programmer Should Know About Memory", Novemeber 2007.RedHat,Inc.

[5] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Computing Surveys, 23(1):5–48, March 1991. URL http://citeseer.ist.psu.edu/goldberg91what.html. 1

[6] Kris Caspersky "Code Optimization: Effective Memory Usage" ,First edition, A-List Publishing, September , 2003.ISBN-10: 1931769249

[7] Paul E Mckenney, "Transactional Memory Everywhere: 2012 Update for HTM"

[8] I.Hameem Shanavas &R.K.Gnanamurthy,"A study of Magnetic memory defects and analysis methods", International Journal of Information Technology and Engineering Vol. 1, No. 1, January-June 2010, pp. 19-22.

[9] Ulrich Drepper, A open source Geek, "What every programmer should know about memory", September 2007.RedHat,Inc.

[10] gnu.org/gcc for GCC manual.

[11] Joe Gebis and David Patterson. Embracing and Extending 20th-Century Instruction Set Architectures. Computer, 40(4):68–75, April 2007. 8.4

[12] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In Proceedings of 20th International Symposium on Computer Architecture, 1993. URL http://citeseer.ist.psu.edu/herlihy93transactional.html. 8.2, 8.2.2, 8.2.3, 8.2.4

[13] http://wiki.freebsd.org/PmcTools/CallchainCaptureAndAnalysis, "Developer tools of Freebsd, Callchain capture and analysis with pmctools."

[14] Ram Huggahalli, Ravi Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O, 2005. URL http://www.stanford.edu/group/comparch/papers/huggahalli05.pdf. 6.3

[15] http://gcc.gnu.org/ "GCC, the GNU Compiler Collections"

[16] John Engel (engel@us.ibm.com), Linux Technical Consultant, IBM, Software Group, "Identify Performance Bottlenecks with OProfile for Linux on Power", May 2005.

**Eisha Akanksha** has completed Bachelor Degree in Electronics and Communication from Vaish College Of Engineering Rohtak (Haryana) and She is currently pursuing Masters in Digital Electronics and Communication Engineering from M.V.J College of Engineering, Bangalore.Her research area includes Memory mangament, Wirless Communication. Email:eisha.b@gmail.com

**Hameem Shanavas .I** is the Doctoral Research Scholar of Anna University, Coimbatore, India. He is currently working Assistant Professor, Department of ECE, M.V.J. College of Engineering, Bangalore, India. He has completed his Bachelor Degree in Electronics and Communication (2006), Masters in VLSI Design (2008) and also he completed Masters in Business Administration (2009). He worked for various institutions in electronics and communication department around many states in India .He had more than 30 publications in international level. He is in editorial committee of many International Journals like IJESET, WASET and reviewer for many Journals like IEEE Transactions, Science Direct, VLSICS, SIPICS, IJANS etc. He is the member of Professional bodies like ISECE , IACSIT, IAEng. His research areas are VLSI Physical Design and Testing. Email:hameemshan@gmail.com.

**V Nallusamy** completed his diploma in Electronics and Communication Engineering from Government Polytechnic, Aranthangi, Tamilnadu in 2001 and subsequently received Bachelor in Engineering from Pavendar Bharathidasn College of Engineering under Bharathidasan University, Trichirappalli, Tamilnadu in 2004. He also obtained a Master of Engineering in Computers and Communication from Pavendar Bharathidasn College of Engineering under Anna University, Tamilnadu in 2010. He is currently leading the Department of ECE, M.V.J. College of Engineering, Bangalore, India. He has published many journals and attended many Conferences in National and International Level. His research areas are Embedded Systems, Robotics and CAD Algorithms. Email: nallu1910@gmail.com