A Survey of Cache Optimization and Memory Management Techniques

Ameya Kathapurkar (@01485008) and Mayur Bagul (@01455429)

The department of Electrical and Computer Engineering,
University of Texas at San Antonio
One UTSA Circle
San Antonio, Texas, 78249, USA

***Abstract*— The survey will validate cache importance, cache optimization techniques and memory management. Cache is the fastest memory available today. If the Cache is full and the processor does not find the required data in the cache, then it would take more time to access that data from main memory, also memory bandwidth would be under-utilized, unless all needed data were already in the Cache. Cache design plays an important role in fast data interchangeability. In achieving this goal there are software and hardware techniques that optimize the cache. Details of cache optimization methods implemented in cache is undertaken in this survey. The survey will explicate the factors like hit rate, miss rate and miss penalty which are very important. Also factors such as cache bandwidth, latency of main memory and power consumption plays an important role in cache optimization. Components like small and multi-level cache, pipelined cache, trace cache and non-blocking cache cause the cache optimization. Also memory management issues such as memory replacement policies and memory optimization techniques are surveyed. Thus, this survey will get detail understanding of cache, its optimizing techniques and memory management.**

Keywords— **Cache optimization, main memory, memory management, cache bandwidth, pipelined cache, trace cache, non-blocking cache, latency, power consumption, memory management replacement policies.**

I. INTRODUCTION

The linkage of the modern day computer systems with the memory is the vibrant topic in research domain that imposes its implications on fast data access and data transfer. Just like human brain the memory has data and instructions.

The memory system is categorized into three main categories. First the Main memory which consists of ROM and RAM that store the data and instructions of processing code. Second the Cache which is a part of main memory used nowadays to speedup the data access. And third the secondary memory which is the hard drive that stores all types of data. The concentration here is the main memory and cache. Manipulation of data access and data transfer are the main aspects associated with main memory and cache. From late 20th century the speed of memory access has drop down from 210 nanoseconds to 0.25 nanoseconds and still decreasing.

The factors such as hit rate, miss rate, miss penalty, cache bandwidth latency of main memory and power consumption for cache optimization will be researched and surveyed. Considering memory management strategies, memory replacement strategy is well researched for paging in virtual memory systems and thus survey will include various replacement strategies. Virtual memory management, segmentation and paging will be probed in detail. Regarding memory optimization use of hardware (optimizing DRAM requirement) and software techniques (data organization and compiler design) will be discussed.

2. MEMORY ARCHITECTURE DESIGN

This section elucidates the memory connection with the cache that optimizes the modern computer systems.

Memory hierarchy was not a concern in the last two decades until the ideas of cache and virtual memory spurred for trade-offs of cost and performance. This enabled the multi-processing and parallel processing to form the modern computing systems. Such high demanding computation was not possible in the old memory systems that were of single level. The concept of memory hierarchy takes advantage of the principle of locality, which states that accessed memory words will be referenced again quickly precisely, temporal locality and that memory words adjacent to an accessed word will be accessed soon after the access in question is spatial locality. Loops, functions, procedures and variables used for counting and totaling all involve temporal locality, where recently referenced memory locations are likely to be referenced again in the near future. Array traversal, sequential code execution, modules, and the tendency of programmers (or compilers) to place related variable definitions near one another all involve spatial locality - they all tend to generate clustered memory references. The principle of locality is particularly applicable to memories for two reasons. First, in most technologies, smaller memories are faster than larger memories. Second, larger memories need more steps and time to decode addresses to fetch the required data. [1]

2.1 CACHE

Cache is very simply a smallish chunk of RAM that can work as fast as the processor. If you could afford it, all of the memory in your machine could be cache RAM - only then we wouldn't call it cache just RAM. In practice only a few megabytes or so of cache RAM is generally used in a typical machine. What is the use of this tiny inadequate amount of RAM when compared with the Gigabytes a typical program needs? The answer is a very strange one but first we need to look a little more closely at how cache RAM is connected. It isn’t just an area of RAM separate from the main RAM. It is connected to the processor in such a way that the processor always tries to use the cache. When the processor wants to read a particular memory location it looks to see if the location is already in the cache. If it isn’t then it has to be read from main memory and there are wait states while the data is retrieved. Once the data is read it is kept in the cache and any subsequent request to read that particular location doesn’t result in a read from main memory and so any wait states are avoided. At this point you might be a little skeptical that cache memory is worth having. For example, when you want the content of a memory location for the first time you still have to wait for it. It is only when you want it a second time that you can have it fast. This seems like a worthless advantage. What might seem even worse is that simple cache systems do not perform cache writes. That is, when the processor wants to write to the memory it ignores the cache and puts up with the wait states. It all seems like complication and cost with virtually no pay off - but there you would be wrong. The reason you would be wrong is that you are not taking account of the way programs behave. The first thing to realize is that most memory accesses are reads. Why? The simple fact is that for every item of data written to the memory lots of locations storing the program, which the processor is obeying, are read. That is, 99% of all memory accesses are so that the processor can find out what to do next! For this reason the wait states generated when data has to be written to memory have a fairly minor impact on performance. The next big surprise is that when a memory location has been read the chances are that it will be read again, and again.

Every program has what is known as a “working set”. These are the memory locations that it re-uses time and time again. If you could set up a memory system that glowed every time a memory location was accessed then what you would see would be a patch of light that moved around as the program ran. There would be memory locations that would be accessed away from the working set but in the main this is where the action is. The reason for this behavior is that programs tend to repeat actions – they run in loops – and this means that the same chunk of memory is read over and over again.

To summarize most memory access is read not write - the speed of a write has very little impact on performance. Programs tend to use memory in small clusters of location - the working set. [3]

The table explains parameters of cache memory and comparison between processors of AMD, IBM and INTEL. As we know, there are three levels of cache L1, L2 and L3. Use of multiple levels of caches turnout to increase in total performance of cache. Now days, some of modern computers have included three levels of on chip cache, such as AMD Phenom II has 6 MB on chip level L3 cache and Intel i7 has 8 MB on chip level L3 cache.
L1 is "Level-1" cache memory, usually built onto the microprocessor chip itself. It is used for storing the microprocessor’s recently accessed information, thus it is also called the primary cache. Most of the recent microprocessors has L1 is in two equal parts. First cache is used to keep program data and second cache that is used to keep instructions for the microprocessor. L1 has very limited size as it has to be as fast as possible to increase the performance of CPU. For example, the Intel i7, AMD Opteron, IBM Power 5 microprocessor comes with 64 thousand bytes of L1.
L2 (that is, Level-2) cache memory is on a separate chip (possibly on an expansion card) that can be accessed more quickly than the larger "main" memory. Level 2 (L2) cache has more space than L1; it may be located on the CPU or on a separate chip or coprocessor with a high-speed alternative system bus interconnecting the cache to the CPU, so that it will not slowed by traffic on the main system bus. AMD Opteron has lowest on chip L2 cache among these three that is 512 KB whether IBM power 8 has 1.8 MB of L2. INTEL i7 is 1 MB of level 2 memory.
Level 3 (L3) cache is typically specialized memory that works to improve the performance of L1 and L2. It can be significantly slower than L1 or L2, but is usually double the speed of RAM. In the case of multicore processors, each core may have its own dedicated L1 and L2 cache, but share a common L3 cache. The L3 cache is usually built onto the motherboard between the main memory and the L1 and L2 caches of the processor module. When an instruction is referenced in the L3 cache, it is typically elevated to a higher tier cache. Typically L3 is an off chip cache memory. In IBM power 8 they included 36 MB of cache memory whether AMD Opteron and INTEL i7 inserted 6 MB and 8 MB of L3 cache memory respectively.
The page size of all the processors AMD Opteron, INTEL i7 and IBM power 8 has same page size of 4 KB.

Parameters	AMD Opteron	IBM Power 5	Intel Core i7
Instruction cache per processor	64KB, 2-way	64KB, 2-way	64KB, 8-way
Latency L1 I (clocks)	2	1	3
Data Cache per processor	64KB, 2-way	32KB, 2-way	64KB, 8-way
Latency L1 D (clocks)	3	2	3
Minimum page size	4KB	4KB	4KB
On-chip L2 cache	512KB	1.8MB, 10-way	1MB, 8-way

L2 banks	2	3	1
Latency L2 (clocks)	7	13	11
Off-chip L3 cache	6MB, max 48-way	36MB, 12-way	8MB, 16-way
Latency L3 (clocks)	29	87	39
Block size (L1 I/L1D/L2/L3, bytes)	64	128/128/128/256	64/64/128/128

Table 1 [1]

2.2 CACHE MAPPING

It is necessary as there are far fewer number of available cache addresses than the memory. Are the address' contents in cache? Cache mapping is used to assign main memory address to cache address and determine hit or miss. There are three basic techniques, Direct mapping, Fully associative mapping and Set-associative mapping. Caches are partitioned into indivisible blocks or lines of adjacent memory addresses usually 4 or 8 addresses per line.

In Direct Mapping the main memory address divided into 2 field's index which contains cache address, number of bits determined by cache and size tag which is compared with tag stored in cache at address is indicated by index. If tags match, we check for valid bit. Valid bit indicates whether data in slot has been loaded from memory. Offset is used to find particular word in cache line.

In Fully Associative Mapping Complete main memory address is stored in each cache address. All addresses stored in cache are simultaneously compared with desired address. Valid bit and offset are same as direct mapping.

In Set-Associative Mapping there is a Compromise between direct mapping and fully associative mapping. Index is same as in direct mapping. But, each cache address contains content and tags of 2 or more memory address locations. Tags of that set are simultaneously compared as in fully associative mapping. Cache with set size (N) are called N-way set-associative .2-way, 4-way, 8-way are common sets.

In Cache-Replacement Policy Technique is used for choosing which block to replace when fully associative cache is full, or when set-associative cache's line is full, or Direct mapped cache has no choice, Random replace block is chosen at random i.e. LRU: least-recently used, in which replaced block is not accessed for longest time, FIFO: first-in-first-out, in which push block onto queue when accessed and choose block to be replaced by popping the queue.

In Cache Write Techniques, When written, data cache must update main memory Write-through is write to main memory whenever cache is written, it is to easiest to implement. Only processor needs to wait for slower main memory write potential for unnecessary writes. In Write-back, main memory is only written when "dirty" (used) block is replaced and extra bit for each block is set, and also when cache block is written it reduces number of slow main memory writes. This increases access.

2.4 CACHE DESIGN

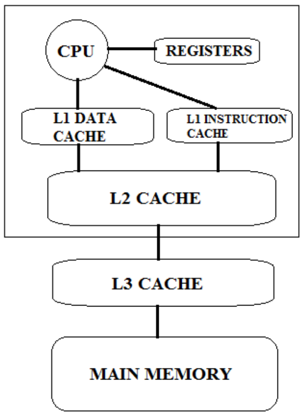


Fig 1[5]

Nowadays modern computer systems come equipped with 3 caches. L1 which is on chip and is separate for data and instruction. L2 is off chip cache in many computer except Intel. L3 cache is usually off chip. Cache reduces latency and more the number of on chip cache less is the latency.

3. CACHE OPTIMIZATION

Our basic aim is to minimize the hit time, increase bandwidth, drop miss penalty and reducing miss rate that is data access and data transfer. Cache optimization using Small and Multilevel cache, Pipelined Cache, Trace Cache, Non- Blocking Cache are surveyed in this section.

3.1 SMALL AND MULTI-LEVEL CACHE

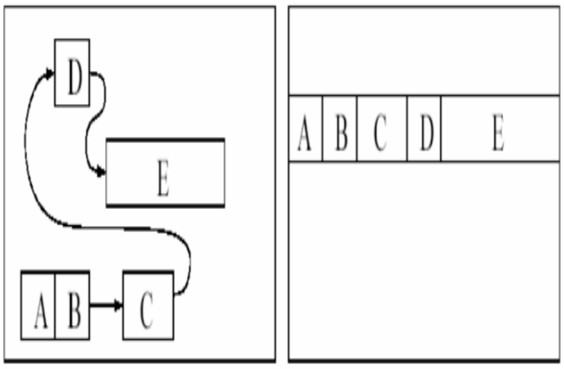
The idea behind this implementation is to minimize the latency of the data coming and going to main memory. As on chip cache increases the cost increases but speeds up the data inflow and outflow. Use of small Cache decreases hit time but on contrary increases miss rate to complement these tradeoffs we implement multilevel cache. Latest there are 3 caches included in modern computer systems.

Multi-level caches use locality of reference seen by each level and decreases as one gets deeper in the hierarchy. Recently referenced data are handled by the upper levels of the memory system. Requests that make it to the lower levels tend to be more widely distributed across the address space. [6].Caches with larger capacities tend to be slower and speed benefit of separate instruction and data caches are not as significant in lower levels of the memory hierarchy.

3.2 PIPELINED CACHE

To increase the clock frequency access, cache are pipelined which increases clock cycle time and high bandwidth. The main challenge in the architecture of the cache system is hiding the latency of the cache lookup and tags comparison. The proposed ongoing research is to reduce latencies by using the set prediction technique [8].

3.3 TRACE CACHE



(a) Instruction cache. (b) Trace cache.

An instruction cache in a microprocessor that stores dynamic instruction sequences after they have been fetched and executed in order to follow the instructions at subsequent times without needing to return to the regular cache or the memory for the same instruction sequence. An advantage of the trace cache is it reduces the required fetch bandwidth on the processing pipeline. Trace cache is accessed in parallel with instruction cache. Hit is equal to trace read into issue buffer and Miss is equal to fetch from instruction cache. Trace cache hit if Fetch address match and Branch predictions match. Trace cache is not on the critical path of instruction fetch.

3.4 NON-BLOCKING CACHE

A blocking cache stalls (make wait) the pipeline on a cache miss. Non-blocking cache or lockup free cache allow data cache to continue to supply cache hits during a miss. Non-blocking caches are an effective technique for tolerating cache-miss latency. They can reduce miss- induced processor stalls by buffering the misses and continuing to serve other independent access requests. Previous research on the complexity and performance of non-blocking caches supporting non-blocking loads showed they could achieve significant performance gains in comparison to blocking caches. However, those experiments were performed with benchmarks that are now over a decade old. [8]

3.5 MEMORY REPLACEMENT STRATEGIES

As memory management strategy memory replacement strategy is intensively studied for faster data access. Data that is not required is not read or written in cache and this saves the processor time. The replacement algorithms can be divided into static page replacement algorithms and dynamic page replacement algorithms.

3.5.1 Static Page Replacement Algorithms

Share frames equally among processes. Split m frames to n users: Each user gets m/n frames. Disadvantage: some applications require more frames than others. For this number of frames are decided at initial load time according to program size, or priority. In this section we will discuss (RAND) Random replacement algorithm, (FIFO) First In First Out replacement algorithm, and Least Frequently Used replacement algorithm. Random replacement algorithm simply chooses the page to be removed at random, hence there is equal chance of every page getting selected without consideration of stream or locality principle. In the case of FIFO, most used page is selected and removed as it is using the principle of typical queue. It is not used due to locality trends.In least frequently used replacement algorithm it selects the page that is not used for a while in past.

3.5.2 Dynamic Page Replacement Algorithm

It is difficult to allocate page when as full analysis of data to be accessed is rarely available for virtual memory controller [1]. Using dynamic page algorithm optimization and adjustment can be made depending upon reoccurring trends. SEQ proposes new replacement algorithm as like LRU and monitors page fault when they occur. Another algorithm is prefetching adaptive algorithm that measures disk transfer times and optimizes the system performance.

4.0 Memory Optimization

4.1 DRAM Chip Design

The main memory consists of DRAM (Dynamic Random Access Memory). The DRAM has large storage capacity compared to SRAM (Static Random Access Memory). DRAM is slower, cheaper and denser than SRAM. To design memory, the processor-memory gap, technologies are developed to have a greater bandwidth. First innovation includes revolutionary addressing by multiplexing row and column addresses. This design reduce the cost and space by filtering a 4K DRAM into 16 pin package instead of 22 pin package. Fast page mode is also one of the improvement on conventional DRAM when it deals with row- address are constant and data from multiple columns is read without wasting more access time. An additional basic change is Synchronous DRAM (SDRAM), unlike the conventional DRAMs are asynchronous to the memory controller, it includes a clock signal to the DRAM interface and it’s register holds a bytes-per-request value, and returns many bytes over several cycles per request. Another major innovation is to increase bandwidth is DDR (double data rate) SDRAM, which doubles the data bandwidth of the bus by reading and writing data on both the rising and falling edges of the clock signal. This techniques exploit more bandwidth with addition of very low cost to system.

4.2 Software Optimization

The software optimization is one of the approach to improve the memory performance with compiler optimization instead of hardware change. Code can easily be reorganize to reduce misses in temporal and spatial locality. For example, the branch straightening, which happens when the compiler predicts that a branch, it rearranges program code by swapping the branch target block with the block sequentially right after the branch.

Code and data reorganization are primarily used to loop transformations. Loop inter-change is to exchange the order of the nested loop, making the code access the data in the order they are stored. Loop fusion is a program transformation by fusing loops that access similar sets of memory locations. After fusion, the accesses are gathered together in the fused loop and can be reused easily and the results indicate that their approach is highly effective and ensures better performance.

4.3 Prefetching

Fetching instructions can used effectively by identifying information will be needed and fetch it in advance-prefetching. The runtime can be reduced by predicting correct instruction about future pages uses. Prefetching strategies should be designed carefully. If a strategy requires significant resources or inaccurately preloads which are not needed pages, it may have result in worse performance than in a demand paging system.

Prefetching includes two types:-

1. Hardware Prefetching
2. Software Prefetching

1. Hardware Prefetching: - This includes both instructions and data can be pre fetched. Instructions following the one currently being executed are loaded into instruction stream buffer. If the requested instruction is already in the buffer, no need to fetch it again but request the next prefetch. Hardware data prefetching is used to exploit of run-time information without the need for programmer or compiler intervention.

2. Software Prefetching: - Most of the software prefetching algorithms are working for data only, applying mostly within loops for large array calculations for both hand-coded and automated by a compiler applied prefetching for affine array references in scientific programs, locality analysis is conducted to find the part of array references suffered from cache misses.

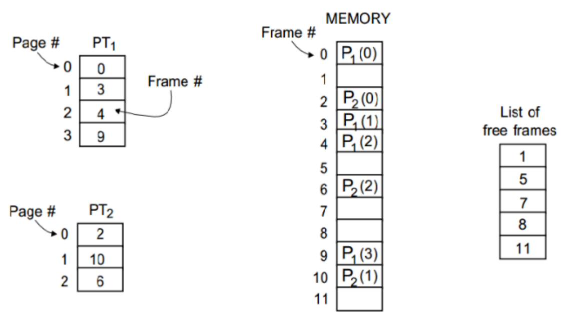
5.0 Virtual memory management

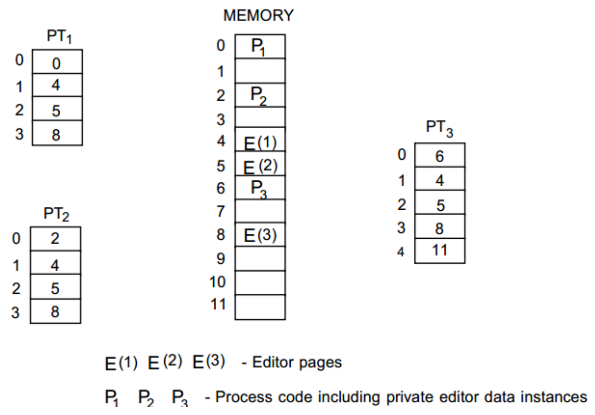
One of the most primitive forms of ``memory management'' is often implemented on systems with a small amount of main memory. This method gives the responsibility of memory management entirely to the programmer; if a program requires more memory than is free memory, the program must be broken up into separate, independent sections and one ``overlaid'' on top of another when that particular section is to be executed. This type of memory management, which is completely under the control of the programmer, is sometimes the only type of memory management available for small microcomputer systems. Modern memory management schemes, usually implemented in mini - to mainframe computers, employ an automatic, user transparent scheme, usually called ``virtual memory''.

Virtual Memory management includes
Paging
Segmentation

5.1 Paging

Memory-management technique that permits the physical address space of a process to be non-contiguous. Each P is divided into equal-sized pages. Memory is also divided into equal-sized frames. (Process pages and memory frames are equal in size.) Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory. Paging is similar to fixed partitioning, only the Ps extend across several frames. Paging allows that only parts of P must be present in memory, while the other parts can be out-swapped, thus freeing the memory for other Ps. Pages of a P mapped in memory don't have to be consecutive. Each P has a page table (PT) which maps pages onto memory frames. A P can access only those memory frames which are listed in its page table. Typical size of frames and pages are 29 = 512 bytes, 210 = 1 Kb, 211 = 2 Kb and 212 = 4 Kb (VAX - 512, NT - 4 Kb) Page size is always power of two! Small pages decrease internal fragmentation, but introduce overhead in space: page tables become larger. Also, disk I/O is less efficient when transferring smaller blocks of data.



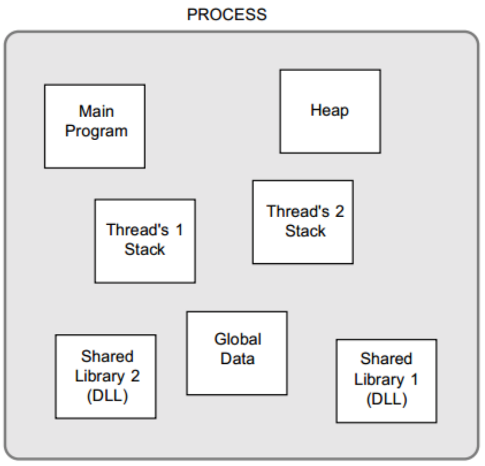


PT1
PT2
MEMORY
P1
P2
E(1)
E(2)
P3
E(3)
PT3
E(1) E(2) E(3) - Editor pages
P1 P2 P3 - Process code including private editor data instances

5.2 Segmentation

Segmentation is involved with loading programs into memory. This does not imply that all of the program needs to be loaded at once. It is possible to load only part of the program into primary memory and this part then calls up whatever extra code is required at that point in time. For example, Dynamic Link Libraries can reside on the hard disk until called up into main memory by an executing program. In paging approach the memory is viewed somehow as a linear structure. This is not how the programmer sees his/her program, which rather consists of several unordered and unequal sized modules, which could be residing in different segments of memory. Division of a program into collection of segments is normally done automatically by the compiler.

Segmentation is similar to dynamic partitioning, only this time process can have several segments, while in dynamic partitioning a partition is accommodating the entire process.



5.3 Segmentation with paging

It exploits the common good features of paging and segmentation. In paging system principle says an application's virtual address space is divided up into equally sized pages. However, these page sized chunks do not match the logical way in which an ordinary process would be broken up—programmers think of a process's memory space being divided into regions for code, global variables, the stack, and the heap for dynamic data structures. Segmentation tries to match the programmer's view by dividing the address space of a process into multiple variable-length segments, one for each of the categories just described. This can simplify how addresses are created. For example, it may make sense to address an array by using an offset from the start of the global variables segment.

To combine segmentation and paging we divide each segment into pages. In systems that combine them, virtual memory is usually implemented with paging, with segmentation used to provide memory protection. The virtual address space is treated as a collection of segments of arbitrary sizes, and the physical memory is treated as a sequence of fixed size page frames. A segment usually spans many pages, and these pages within a segment are mapped onto actual physical page frames. There may be a segment corresponding to each logical view of a process heap segment, code segment, data segment, stack segment etc. This allows protection or sharing mechanisms to be applied at the granularity of segments, rather than individually for each page.

6.0 Conclusion

7.0 References

[1] Henessey and Patterson, Computer Architecure.

[2] Bansal, S. and Modha, D.S. (2004) CAR: Clock with Adaptive Replacement, Proceeding of USENIX Conference on File and Storage Technologies, 187-200, San Francisco, CA.

[3] Belady, L.A. (1966) A Study of Replacement Algorithms for Virtual Storage Computers, IBM Systems, 25(5), 491-503.

[4] Belady, L.A. (1969) Dynamic Space Sharing in computer System, Communications of the ACM, 12(5), 282-288.

[5] Belayneh, S. and Kaeli, D. (1996) A Discussion on Non-blocking/lockup-free Caches, ACM SIGARCH Computer Architecture News, 24(3), 18-25.

[6] Bennett, B.T. and Franaczek, P.A. (1976) Cache Memory with Prefetching of data by Priority, IBM Tech. Disclosure Bull, 18(12), 4231-4232.

[7] Bennett, B.T., Pomerene, J.H., Puzak, T.R. and Rechtschaffen (1982) R.N. Prefetching in a Multilevel Memory Hierarchy, IBM Tech. Disclosure Bull, 25(1), 88.

[8] Bergey, A.L. (1978) Increased Computer Throughput by Conditioned Memory Data Prefetching, IBM Tech. Disclosure Bull, 20(10), 4103.

[9] Bhandarkar, D.; Ding, J. (1997) Performance Characterization of the Pentium Pro Processor, Proceeding of High-Performance Computer Architecture, 1-5 Feb, 288-297, San Antonio, Texas, USA.

[10] Black, B., Rychlik, B., and Shen, J.P. (1999) The Block-based Trace Cache, Proceeding of the 26th Annual International Symposium on Computer Architecture, 196-207.

[11] Burger, D., Goodman, J.R. and Sohi, G. S. (1997) Memory Systems. The Computer Science and Engineering Handbook, 47-461.

[12] Cao, P., Felten, E.W., Karlin, A.R. and Li, K. (1995) A Study of Integrated Prefetching and Caching Strategies, Proc. of ACM SIGMETRICS, 188-197.

[13] Carr, W.R., Hennessy, J.L. (1981) WSClock -A Simple and Effective Algorithm for Virtual Memory Management, Proc. of the ACM SOSP, 87-95.

[14] Carr, W.R. (1984) Virtual Memory Management. UMI Research Press.

[15] Chang, D.C., et al (1995) Microarchitecture of HaL's Memory Management Unit, Compcon Digest of Papers, 272-279.

[16] Chan, K.K., et al. (1996) Design of the HP PA 7200 CPU, Hewlett-Packard Journal, 47(1), 25-33.

[17] Chen, W.Y, Mahlke, S.A, Chang, P.P., Hwu, W.W. (1991) Data Access Microarchitectures for Superscalar Processor with Compiler-Assisted Data Prefetching, Proceeding 24th International Symposium on Microcomputing, Albuquerque, NM, November, 69-73.

[18] Chen, T.; Baer, J. (1992) Reducing Memory Latency via Non-blocking and Prefetching Caches, Proceeding of ACM SIGPLAN Notice, 27(9), 51-61.

[19] Chen, T.; Baer, J. (1994) A Performance Study of Software and Hardware Data Prefetching Schemes, Proceeding of the 21st Annual International Symposium on Computer Architecture, Chicago, IL, April, 223-232.

[20] Cho, S. (2007) I-Cache Multi Banking and Vertical Interleaving, Proceeding of ACM Great Lakes Symposium on VLSI, March 11-13, 14-19, Stresa-Lago Maggiore, Italy.

[21] Chu, W.W. and Opderbeck, H. (1974) Performance of Replacement Algorithms with Different Page Sizes, Computer, 7(11), 14-21.

[22] Conti, C.J. (1969) Concepts for Buffer Storage, IEEE Computer Group News, 2(8), 9-13.

[23] Dahlgren, F. and P. Stenstrom (1995) Effectiveness of Hardware-based Stride and Sequential Prefetching in Shared-memory Multiprocessors,Proc. First IEEE Sympo- sium on High- Performance Computer Architecture, Raleigh, NC, Jan. 68-77.

[24] Daley, R., Dennis, J.B. (1968) Virtual Memory Processes, and sharing in Multics, Communication of the ACM, 11(5), 306-312.

[25] Dennis, J.B. (1965) Segmentation and the Design of Multi-programmed Computer System, Journal of ACM, 12(4), 589-602.

[26] Denning, P.J. and Van Horn, E.C. (1966) Programming Semantics for Multi-programmed Computations, Communications of the ACM, 9(3), 143-155.

[27] Denning, P.J. (1968) The Working Set Model for Program Behavior, Communications of the ACM, 11(5), 323-333.

[28] Denning, P.J. (1972) On Modeling Program Behavior, Proc Spring Joint Computer Conference, 40, 937-944.

[29] Denning, P.J. (1980) Working Sets Past and Present, IEEE Transactions on Software Engineering, SE6(1), 64-84.

[30] Denning, P.J. (2005) The Locality Principle, Communications of the ACM, 48(7), 19- 24.

[31] Doran, R.W. (1976) Virtual Memory, Computer, 9(10), 27-37.

[32] Farkas, K.I.; Jouppi, N.P.; Chow, P. (1994) How Useful Are Non-Blocking Loads, Stream Buffers, and Speculative Execution in Multiple issue Processors? Western Research Laboratory Research Report 94/8.

[33] J. Feldman, Computer Architecture, A Designer's Text Based on a Generic RISC Architecture, 1st edition, McGraw-Hill, 1994.

[34] Foss, R.C. (2008) DRAM - A Personal View, IEEE Solid-State Circuits Newsletter, 13(1), 50-56.

[35] Fu, J.W.C, Patel, J.H., and Janssens, B. L. (1992) Stride Directed Prefetching in Scalar Processors, Proceeding of 25th International Symposium on Microarchitecture, Portland, OR, December, 102-110.

[36] Fukunaga, K. and Kasai, T. (1977) The Efficient Use of Buffer Storage, Proc. ACM 1977 Annual Conference, 399-403.

[37] Galazin, A.B., Stupachenko, E.V., and Shlykov, S.L. (2008) A Software Instruction Prefetching Method in Architectures with Static Scheduling, Programming and Computer Software, 34(1), 49-53.

[38] Gelenbe, E. (1971) The Two-Thirds Rule for Dynamic Storage Allocation Under Equilibrium, Information Processing Letters, 1(2), 59-60.

[39] Gelenbe, E. (1973) The Distribution of a Program in Primary and Fast Buffer Stor- age, Communications of the ACM, 16(7), 431-434.

[40] Gelenbe, E. (1973) Minimizing Wasted Space in Partitioned Segmentation, Communications of the ACM, 16(6), 343-349.

[41] Gelenbe, E., Tiberio, P. and Boekhorst, J.C. (1973) Page Size in Demand-paging Systems, Acta Informatica, 3, 1-24.

[42] Gelenbe, E. (1973) A Unified Approach to the Evaluation of a Class of Replacement Algorithms, IEEE Transactions on Computers, 22(6), 611-618.

[43] Gelenbe, E. and Zhu, Q. (2001) Adaptive Control of Prefetching, Performance Evaluation, 46, 177-192.

[44] Ghasemzadeh, H., Mazrouee,S., Moghaddam, H.G., Shojaei, H. and Kakoee, M.R. (2006) Hardware Implementation of

Stack-Based Replacement Algorithms, World Academy of Science, Engineerin g and Technology, 16, 135-139.

[45] Glass, G. and Cao, P. (1997) Adaptive p age replacement based on memory refer ence behavior, Proc of the 1997 ACM S IGMETRICS, 25(1), 115-126.

[46] Gibson, D.H. (1967) Consideration in B lock-oriented Systems Design, Proc Spr ing Jt Computer Conf, 30, Thompson B ooks, 75-80.

[47] Gupta, R.K. and Franklin, M.a. (1978) Working Set and Page Fault Frequency Replacement Algorithms: A Performan ce Comparison, IEEE Transactions on C omput- ers, C-27, 706-712.

[48] Handy, J (1997) The Cache Memory Bo ok, Morgan Kaufmann Publishers.

[49] Hanlon, A.G. (1966) Content-Addressa ble and Associative Memory System - A Survey, IEEE Transactions on Electroni c Computers, 15(4), 509-521.

[50] Hennessy, J.L. and Patterson, D.A. (200 7) Computer architecture: a quantitative approach, Morgan Kaufmann Publisher s Inc., San Francisco, CA, 4th Edition.

[51] ILiffe, J.K. (1968) Basic Machine Princ iples. American Elsevier, New York.

[52] Jacob, B. and Mudge, t. (1998) Virtual Memory: Issues of Implementation, IEE E Computer, 31(6), 33-43.

[53] Jayarekha, P.; Nair, T.R (2009) Proceed ing of InterJRI Computer Science and N et- working, 1(1), Dec, 24-30.

[54] Johnston, J.B. (1969) The Structure of Multiple Activity Algorithms, Proc. Thi rd Annual Princeton Conf. 80-82.

[55] Jones, F. et al., (1992) A New Era of Fa st Dynamic RAMs, IEEE Spectrum, 43-49.

[56] Kaplan, K.R. and Winder, R.O. (1973) Cache-based Computer Systems, IEEE Computer, 6(3), 30-36.

[57] Kane, G. and Heinrich J. (1992) MIPS R ISC Architecture, Prentice Hall.

[58] Karedla, R., Love, J.S. and Wherry, B. (1994) Caching Strategies to Improve Di sk System Performance, IEEE Compute r, 27(2), 38-46.

[59] Karlsson, M., Dahlgren, F., and Stenstro m, P. (2000) A Prefetching Technique f or Irregular Accesses to Linked Data St ructures, Proceeding of the 6th Internati onal conference on High Performance C omputer Architecture, Toulouse, France , January, 206-217.

[60] Lam, M.; Rothberg, E.; Wolf, M.E. (19 91) The Cache Performance and Optimi zation of Blocked Algorithms, Proceedi ng of the Fourth International Conferen ce on ASPLOSIV, Santa Clara, Apr, 63 -74.

[61] Linduist, A.B., Seeder, R. R. and Come au,L.W. (1966) A Time-Sharing System Using an Associative Memory, Proceed ing of the IEEE, 54, 1774-1779.

[62] Lin, Y.S. and Mattson, R.L. (1972) Cos t-Performance Evaluation of Memory H ierarchies, IEEE Transactions Magazine , MAG-8(3), 390-392.

[63] Luk, C.K and Mowry, T.C (1996) Com piler-based Prefetching for Recursive D ata Structures, Proceeding of 7th Confer ence on Architectural Support for Progr amming Languages and Operating Syst ems, Cambridge, MA, October, 222-23 3.

[64] Mackenzie, F.B. (1965) Automated Sec ondary Storage Management, Datamati on, 11, 24-28.

[65] Mahapatra, N.R., and Venkatrao, B. (19 99) The processor-memory bottleneck: problems and solutions, Crossroads, 5(3).

[66] Mahajan, A.R. and Ali, M.S. (2007) Op timization of Memory System in Real-ti me Embedded Systems, Proceeding of t he 11th WSEAS International Conferen ce on Computers. 13-19.

[67] Marshall, W.T. and Nute, C.T. (1979) A nalytic modeling of working set like rep

lacement algorithms, Proc. of ACM SIG METRICS, 65-72.

[68] Megiddo, N.; and Modha, D.S. (2003) A RC:A Self-tuning, Low Overhead Repla cement Cache, Proceeding of 2nd USE NIX conference on File and Storage Te chnologies, 115-130, San Franciso, CA.

[69] Mowry, T. (1991) Tolerating Latency th roughSoftware-controller Prefetching in Shared-memory Multiprocessors, Journ al of Parallel and Distributed Computin g, 12(2), 87-106.

[70] Ng, Ray (1992) Fast Computer Memori es, IEEE Spectrum, 36-39.

[71] Olukotun, K., Mudge, T., and Brown, R . (1997) Multilevel Optimization of Pipl elined Caches. IEEE Transactions on C omputers, 46, 10, 1093-1102.

[72] Organick, E.I. (1972) The Multics Syste m: An Examination of Its Structure, MI T Press.

[73] Ou, L., He, X.B., Kosa, M.J., and Scott, S.L. (2005) A Unified Multiple-Level C ache for High Performance Storage Sys tems, mascots, 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecomm unication Systems, 143-152.

[74] Panda, P.R. et al (2001) Data and Mem ory Optimization Techniques for Embe dded System, ACM Transactions on De sign Automation of Electronic Systems, 6(2), 149-206.

[75] Perkins, D.R. (1980) The Design and M anagement of Predictive Caches, PH.D dissertation, Univ of Calif., San Diego.

[76] Peters, M. (2000) Enhanced Memory Sy stems, Personal Communications, Septe mber.

[77] Prieve, B.G. and Fabry, R.S. (1976) VM IN - An Optimal Variable Space Page R e- placement Algorithm, Communicatio ns of the ACM, 19(5), 295-297.

[78] Ramirez, A., Larriba-Pey, J.L., and Val ero, M. (2005) Software Trace Cache, I EEE Transactions on Computer, 54(1), 22-35.

[79] Randell, B. and Kuehner, C.J. (1968) D ynamic Storage Allocation Systems, Co mmunications of the ACM, 297-305.

[80] Rivers, J.A.; Tyson, G.S.; Dividson, E.S .; Austin, T.M. (1997) On High-Bandwi dth Data Cache Design for Multi-Issue Processors, Proceeding of International Sympo- sium of Microarchitecture, Dec , 46-56.

[81] Rotenerg, E., Bennett, S., and Smith, J. E. (1999) A Trace Cache Microarchitec ture and Evaluation, IEEE Transactions on Computers, 48(2), 111-120.

[82] Roth, A. and Sohi, G., (1999) Effective Jump-pointer Prefetching for Linked Da ta Structures. Proceeding of the 26th Int ernational Symposium on Computer Ar chitecture, Atlanta, GA, May, 111-121.

[83] Pas, R. (2002) Memory Hierarchy in Ca che-Based Systems, Sun Microsystems.

[84] Sadeh, E. (1975) An Analysis of the Per formance of the Page Fault Frequency (PFF) Replacement Algorithm, Proc of t he fifth ACM SOSP, 6-13.

[85] Samples, A.D.; Hilfinger, P.N. (1988) C ode Reorganization for Instruction Cach es, University of California at Berkeley, Berkeley, CA.

[86] Serhan, S.I. and Abdel-Haq, H.M. (200 7) Improving Cache Memory Utilizatio n, World Academy of Science and Tech nology, 26, 299-304.

[87] Shemer, J.E. and Shippey, B. (1966) Sta tistical Analysis of Paged and Segmente d Computer Systems. IEEE Trans. EC-1 5, 6, 855-863.

[88] Shiell, J. (1986) Virtual memory, virtua l Machines, Byte, 11(11), 110-121.

[89] Qi Zhu , Ying Qiao , "A Survey on Com puter System Memory Management and Optimization Techniques", American Jo urnal of Computer Architecture, Vol. 1 No. 3, 2012, pp. 37-50. doi: 10.5923/j.a jca.20120103.01.

[90] Smaragdakis, Y., Kaplan, S. and Wilson , P. (2003) The EELRU Adaptive Replacement Algorithm, Performance Evaluation, 53(2), 93-123