Memory Systems: Overview and Trends

Saurabh Chheda Jeevan Kumar Chittamuru Csaba Andras Moritz
Laboratory of Software Systems and Architectures
Dept. of Electrical and Computer Engineering
University of Massachusetts
Amherst, Massachusetts 01002
{schheda, jchittam, andras}@ecs.umass.edu

Abstract

Computer pioneers have correctly predicted that programmers would want unlimited amounts of memory. An economical solution to this desire is the implementation of a Memory Hierarchical System, which takes advantage of locality and cost/performance of memory technologies. As time has gone by, the technology has progressed, bringing about various changes in the way memory systems are built. Memory systems must be flexible enough to accommodate various levels of memory hierarchies, and must be able to emulate an environment with unlimited amount of memory. For more than two decades the main emphasis of memory system designers has been achieving high performance. However, recent market trends and application requirements suggest that other design goals such as low-power, predictability, and flexibility/reconfigurability are becoming equally important to consider.

This paper gives a comprehensive overview of memory systems with the objective to give any reader a broad overview. Emphasis is put on the various components of a typical memory system of present-day systems and emerging new memory system architecture trends. We focus on emerging memory technologies, system architectures, compiler technology, which are likely to shape the computer industry in the future.

1 Introduction

A computer system can be basically divided into three basic blocks. They can be called the Central Processing Unit, commonly called the *Processor*, the Memory subsystem and the IO subsystem. Figure 1 illustrates this division.

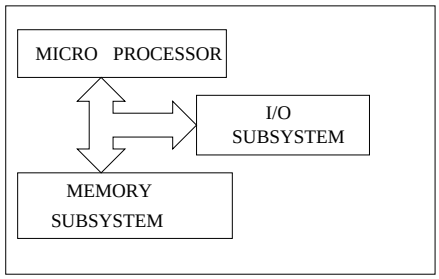

Figure 1: Structure of a Typical Computer System

The Memory subsystem forms the backbone of the whole system storing vital and large amount of data that can be retrieved at any given time for processing and other related work. In the development of memory systems, two major issues have been increase in availability of amount of memory and fast access of data from the memory. Thus the motto of any memory system developer can be very safely said to be *UNLIMITED FAST MEMORY*.

Before delving into the depths of a memory system, it is important to acquaint ourselves with the basic underlying principles on which a memory system is built. These are called the *Principles of Locality*. There are two types of locality:

1. **Temporal Locality -** This law states that if an item is referenced, it will tend to be referenced again.
2. **Spatial Locality -** This law emphasizes on the fact that if an item is referenced, items with addresses close by will tend to be referenced in the near future.

Most programs contain loops, so instructions and data are likely to be accessed repeatedly, thus showing Temporal Locality. In addition, instructions or data are accessed in sequence, thus exhibiting Spatial Locality. These two principles, plus the fact that smaller memory is faster has lead to development of *Hierarchy - Based Memories* of different speeds and sizes. We can broadly divide the Memory subsystem into large blocks, as seen in Figure 2. Levels of memory, in this type of structure, tend to be much bigger and considerably slower as their relative position from the CPU increases.

The microprocessor industry is progressing at a very rapid pace. As a result, the speed of processors far outstrips the speed, or access times, of the memory. Hence it is impervious to establish a hierarchical structure of memory with very fast memory close to the processor while slower memory, which is accessed relatively less, away from the processor. Though memory hierarchy contains many levels, data is typically transferred between two adjacent levels only.

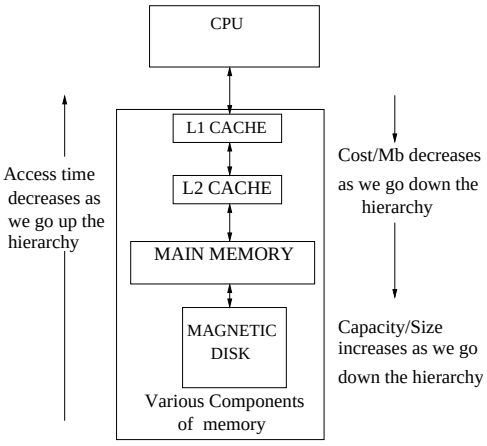

Figure 2: A typical Memory Hierarchical System

An important thing to note is that data transferred between two levels is always in the form of blocks of data instead of single - byte

Category	SPECint92	SPECfp92	Database	Sparse
Clocks Per Instruction CPI	1.2	1.2	3.6	3.0
I cache per 1000 instructions	7	2	97	0
D cache per 1000 instructions	25	47	82	38
L2 cache per 1000 instructions	11	12	119	36
L3 cache per 1000 instructions	0	0	13	23
Fraction of time in Processor	0.78	0.68	0.23	0.27
Fraction of time in I cache misses	0.03	0.01	0.16	0.00
Fraction of time in D cache misses	0.13	0.23	0.14	0.08
Fraction of time in L2 cache misses	0.05	0.06	0.20	0.07
Fraction of time in L3 cache misses	0.00	0.02	0.27	0.58

Table 1: Time spent on different blocks of memory & processor in ALPHA 21164 for four programs (from [17]).

transfers. Performance of any level is defined by the number of hits and misses and speed at which these occur. *Hit Time* is the time to fetch or store data at that level of memory. *Miss Time* is the time required to fetch or store a required block of data from lower levels in memory hierarchy.

The concepts used to build memory systems affect the overall system as a whole, including how the operating system manages memory, how compilers generate code, and how applications perform on a given machine. As mentioned earlier the various layers of a memory system have different performance characteristics. This can be illustrated, as shown in Table 1 which shows the behavior of an Alpha 21164 for four different programs. This processor uses three levels of memory on-chip in addition to external main memory.

As seen, Memory Systems are critical to performance. This can be judged from the fact that the advantage of having a fast processor is lost if the memory system is relatively slow to access and if applications spend considerable fraction of their execution time in the memory system.

This paper focuses on the different types of memory systems in use as of today. Section 2 describes the concept of caches and how they improve the performance. Section 3 focuses on the concept of virtual memories, a mechanism that gives the programmer the sense of unlimited address space. Section 4 is a guide to the different technologies and devices used for making various levels of a memory system. Section 5 will focus on the importance of compiler technologies in building an efficient memory system. Finally, Section 6 will introduce new developments in memory system designs which also emphasize on power consumption aspects and matching application requirements better.

Finally, we must reiterate a statement put forward by Maurice Wilkes in the " Memories of a Computer Pioneer", 1985.

> .... the single one development that put computers on their feet was the invention of a reliable form of Memory ....

2 Cache Subsystem

As already seen, the time required to access a location in memory is crucial to the performance of the system as a whole. Since it is not feasible to implement the address space as a a single large contiguous memory block, due to constraints of memory access latency and cost, the Memory Subsystem must be arranged in several levels in a way that is cost performance efficient. According to the principles of Locality of Reference, a program tends to access the same memory locations several times over a period of time and these references tend to be sequential. Hence, designers use a small fast memory close to the processor to improve memory access times. This memory holds the most frequently accessed data which can be retrieved by the processor with very low latency. Such a memory is called *Cache Memory*.

Every time the processor attempts to fetch from or write to main memory, a check is simultaneously performed in the cache to determine if the required memory reference is present in the cache. In case of a *Hit*, the word is directly fetched from the cache instead of the main memory. In case of a *Miss*, the word is fetched from memory and given to the processor. At the same time, it is stored into the cache so that any future reference to the same location will result in a cache hit. As a result of this, any valid data contained in the cache is a copy of some data located in the next level of the memory hierarchy.

In its simplest organization the cache is structured as a list of blocks indexed with a portion of the program address. Hence, there are many program addresses that are mapped to the same cache block. This simple organization is called direct mapped, it has the fastest access time, but also the disadvantage of possible mapping conflicts between addresses that map to the same cache block. Mapping conflicts degrade performance because the replaced block is often referenced again shortly after.

A solution used to avoid mapping conflicts is to divide the cache into a number of sets. To determine what location is mapped currently to the associated entry in the cache, each cache entry is accompanied by a set of bits which act as a *Tag Field*.

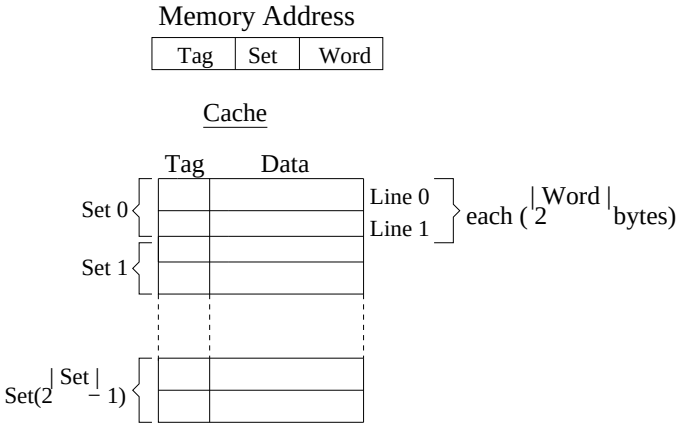

Figure 3: This figure shows how a main memory address is split into Tag field, Set field and the word field so as to facilitate the determination of probable location of accessed word in the cache memory.

The main memory is divided into blocks The cache is subdi-

vided into sets, each of which can contain multiple blocks or lines. Each cache line can store a single block of memory at a time. As mentioned earlier multiple blocks from the main memory may contend for the same location in the cache. The way in which a memory block can be mapped to a cache line is determined by the mapping function or cache organization. The three ways of cache organizations are illustrated below, with their relative merits and disadvantages shown in Table 2.

Direct Mapped Cache
As mentioned earlier, the direct mapped cache is divided into a maximum number of sets, each containing a single line. Two blocks mapping into the same cache line can't be stored in the cache simultaneously, giving rise to intrinsic and conflict misses. Hence, if a program accesses two words from blocks of memory that map to the same cache line alternatively, a large amount of time will be spent transferring data between the cache and Main Memory (or other cache layers). This phenomenon may cause significant performance degradation if it happens in program loops and it is often referred to as *thrashing*.

Fully Associative Cache
The fully associative cache consists of a single set with multiple cache lines. As the name suggests, a block from main memory can be placed anywhere in the cache. In fully associative caches there are therefore no conflict misses only capacity related misses. A big drawback of such a cache mapping scheme is the larger latency involved in tag checking. This is due to the fact that the tag component of each memory reference has to be checked with all the tag entries in the cache to determine a hit or a miss. Unfortunately, this latency also increases with the size of a cache. Additionally, the hardware incorporated to perform such a tag-matching is fairly complex and relatively expensive to fabricate on a chip.

Set Associative Cache
If a block from main memory can be placed in a restricted number of sets in the cache, the cache is said to be set associative. A set is a group of cache lines. There are many different ways that set associativity is achieved. Commonly, a block from main memory is first mapped onto a set, and then the block can be placed in any line within the set. In this technique the tag comparison is to be done for *K* entries at a time (assuming K lines per set) for a K-Way Associative Mapping.

The range of caches from direct-mapped to fully associative is really a continuum of levels of associativity: Direct mapped is a simple one-way set associative cache, and a fully associative cache with *m* lines could be called a m-way set associative cache. The vast majority of processors today use direct mapped, two-way set associative, or four-way set associative caches. Adding more associativity typically gives diminishing returns in eliminating conflict misses.

2.1 Replacement Algorithms

When a cache miss occurs, the cache controller must select a block to be replaced with the new data. In Direct-mapped caches, since there is just one line per set, there is no replacement policy. For other cache configurations, two primary replacement policies are used.

Random Replacement
To spread allocation uniformly, candidate lines for replacement are randomly selected. A disadvantage of such a policy is that the most recently updated or used lines might be replaced, causing a miss when the same line is accessed the next time. Thus, this replacement policy may not be suitable for applications with good locality.

Least Recently Used Replacement
To reduce the chance of throwing out information that will be needed soon, accesses to blocks are recorded. The block replaced is the block that has been unused for the longest time. This technique exploits the fact that any line that has been accessed recently is more likely to be used again as compared to a line which has not been accessed for a long time.

In addition to the primary techniques of replacement described above, other techniques such as *FIFO: First In First Out*, and *LFU: Least Frequently Used* can be employed. More details about these policies can be found in [19].

2.2 Write Policy

In case of a write access to the cache, certain policies have to be defined so that *cache coherence* is maintained, i.e the data in the cache is consistent with corresponding data in main memory. There are two basic write policy options.

Write Through
In this policy, the information is written to both the line in cache and the line in main memory. As a result, the write latency is comparable to writing directly to the main memory. At the same time, this type of policy results in a large amount of memory traffic.

Write Back
This policy is widely employed in the cache designs of present day. The information is written only to the line in cache. When the line in cache is being purged out of the cache or is to be replaced by another line, the line that is about to be replaced is first written back to the main memory. To implement such a scheme, a separate bit, called the *Dirty Bit*, is associated with each line in cache. If the Dirty bit is set, the information in that line has been modified since the time it was fetched and does not match the value in the main memory. Another advantage of this scheme is that it reduces the traffic between the cache and main memory substantially.

The problem of managing cache validity is more significant in the case of multiprocessors where a set of machines have a common shared memory. A change in a word of cache of one processor causes the same word in other caches to be invalidated in addition to the invalidation of the word in main memory. So, additional methods must be incorporated along with the basic write policies to maintain cache coherence.

Some possible approaches to achieve cache coherence include:

Bus Watching
The address lines of the main memory are monitored by the cache controller of each of the machines to detect whether any other bus master has written to the main memory. If a change is made to the location of shared memory, that also exists in its cache, the corresponding entry in the cache is invalidated. This method requires the use of write-through process by all cache controllers.

Hardware Transparency
In this method, additional hardware is used to ensure that all updates to the main memory via cache of any processor, are reflected on all caches. Thus, in case of modifications to any of the words in main memory, the matching words in the other caches also gets updated.

Non-Cacheable Memory
In this method only a portion of the memory is shared between the processors. Consequently any reference to that portion of memory will be a miss in the cache, since that portion of memory cannot be written to the cache. This non-cacheable memory can be identified using chip-select logic or high address bits.

Organization	# sets	# lines/set	Hit latency	Checking method	Miss rate
Direct Mapped	highest	1	least	simple	highest
Set Associative	greater than 1	greater than 1	medium	less simple	medium
Fully Associative	1	=# cache lines	highest	exhaustive	least

Table 2: Comparison of various cache organizations.

2.3 Performance Issues

The performance of a cache is of major concern to computer architects as the clock cycle time and speed of the CPU are tied to the cache. The performance of the cache is mainly considered to be a function of the miss rate, miss penalty, and the hit time. Understanding of these three factors have helped architects in coming up with optimization techniques. These cache optimizations can be divided into three categories:

Miss Rate Reduction Techniques
Miss Rate is the ratio of the number of memory references not found in the cache to the total number of memory references. There are three kinds of misses which must be tackled to reduce the miss rate. They are known as the *compulsory*, *capacity* and *conflict* misses. Of the many factors that influence the miss rate, the most important of them are the size of the cache, associativity, the behavior of the program and the replacement algorithm used. Using larger block sizes helps reduce miss rate by reducing the number of compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality. At the same time larger blocks increase miss penalty. Additionally, if the cache size is small, increase in cache block size might increase the conflict and capacity misses, thus increasing the miss rate. Another method to decrease the miss rate relies on the fact that increasing associativity decreases the conflict misses. But a drawback of this scheme is that it may increase the hit latency. Hence, some modern techniques have been devised which help decrease the miss rate without incurring the above mentioned side-effects.

One way to decrease the conflict misses and hence the miss rate is by adding a small, fully associative cache between the cache and its refill path. This cache is called the *Victim Cache* [3] and it only contains blocks that have been recently discarded. On the next miss, they are checked to see if they have the desired data before going down to the next level of the memory hierarchy. Another technique of reducing miss rate is the use of *Pseudo Associative* or *Column-Associative* Caches [1]. These caches are a fit between the direct-mapped caches and the two-way set associative caches. Cache accesses in Column-Associative caches proceed in the same way as in a normal direct-mapped cache. On a miss, however, before going to the next level of memory hierarchy, another cache entry is checked to see if the memory reference matches there.

Miss rate reduction can also be achieved by other methods which are compiler-based and are discussed in Section 5.

Miss Penalty Reduction
The miss penalty can be reduced by using techniques such as Sub-Block Placement, Early Restart and Critical Word First. In the *Sub-Block Placement* scheme, a valid bit is added to units smaller than the block. Each of these units is called a *sub-block*. Thus it can be said that a block is made up of sub-blocks, each associated with their valid bits. So in case of a single sub-block being invalidated, only a single sub-block has to be fetched from the next level of memory in the hierarchy instead of fetching the whole block. This results in reduced miss penalty as the the transfer time of the sub-block is smaller than the transfer time of a block would have been. In the *Early Restart and Critical Word First* method, the word requested by the processor is fetched first and sent to the processor for execution while the other words of the same block are being fetched. This technique does not reduce the memory access time in case of a miss, but instead reduces the CPU wait time in case of a miss. Another common technique employed in processors today is the use of a larger second-level cache between the first cache and the main memory. This second-level cache has an access time greater than the first-level cache, but much lower compared to main memory access time.

Hit time Reduction
As already discussed, the Hit Time increases with associativity and so small and simple caches are to be used. Other approaches include using a technique called *Pipelined Writes* and avoiding address translation during the cahe indexing.

Interested readers can go through [19] for more information on these techniques.

Single & Multi-Level Caches
The gap between processor speeds and memory performance is increasing the importance of miss latencies to the main memory. In case of a single L1 cache, all the misses are to be served by the main memory, which has large latency. Hence, the number of misses in the L1 cache are to be reduced. This can be achieved by using a larger L1 cache which increases the availability of data/instructions in it and thus reduces the number of memory accesses. However, a larger L1 cache affects the CPU cycle time. So a better solution would be a two-level cache organization with a smaller and faster L1 cache, and a larger but slower L2 cache that stores and supplies data at lesser latencies compared to the main memory. Datapoints related to performance improvements obtainable with L2 caches can be found in Przybylski [20]. In case of applications with unpredictable access patterns, it was observed that it will be advantageous even to have a third level cache. For example, it can be seen from Table 1 that the usage of multilevel caches can improve processor utilization by more than a factor of 2, depending on the application.

Unified and Split caches
With the advent of superscalars, a unified Level-1 cache is being replaced by split caches. One cache is dedicated to instructions while the other is dedicated to data. Superscalar machines emphasize on parallel instruction execution requiring a constant stream of instructions. Hence it is advantageous to implement a separate cache for instruction memory to prevent bottlenecks and hardware resource contention.

3 Virtual Memory

In the last section we saw how a cache helps improve memory access speed. In this section we will focus on a mechanism which helps to give rise to an illusion of unlimited amount of memory, and at the same time helps prevent undesirable data sharing in multi-process computer systems. Virtual Memory helps in protecting the code and data of user-end applications from actions of other programs. In addition to this it also allows programs to share their address spaces if desired. As the name suggests, virtual memory systems create an illusion of a single-level store with the access

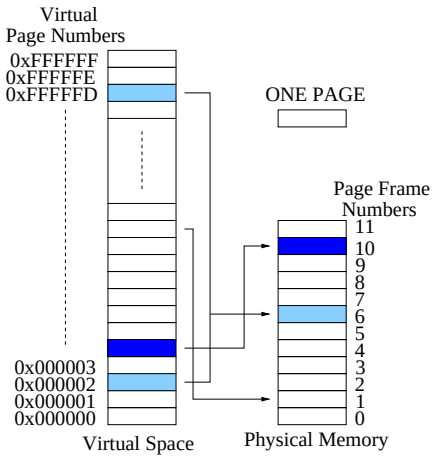

Figure 4: Typical structure of Virtual Memory mapped to Physical Memory.

time that equals that of main memory. The operating system and the hardware, together support this illusion by translating virtual addresses to physical ones. Each process running on the processor is assigned a block of virtual memory addresses. The total virtual memory assigned to *n* processes will be considerably larger than the total main memory available. The virtual memory address space is divided into uniform *virtual pages*, each of which is identified by a *virtual page number*. Similarly, the physical memory can be said to be divided into uniform *page frames*, each of which is identified by a *page frame number*. This is illustrated in Figure 4. Thus we can say that virtual memory is a mapping of virtual page numbers to page frame numbers. A very important thing to note is that a virtual page of a process has a unique physical location and it is also possible that different virtual pages, not necessarily of different processes, may have the same physical location. Main memory contains only the active portions of many programs. This allows efficient sharing of memory by different programs. So when a new page is required, the operating system fetches the required pages from disks, or any secondary storage device.

In virtual memory, the address is broken into a *virtual page number* and a *page offset*. Figure 5 shows the translation of a virtual page number, which is generated by the processor, to a physical page frame number, which is used to access the memory. The physical page frame number constitutes the upper portion of the physical address, while the offset, which does not change, constitutes the lower portion. The translation is carried out by hardware that implements a *hashing function*. A virtual memory miss is called a *page fault* and typically it takes millions of cycles to process. Thus, the page size of virtual memory must be large enough to amortize the high access time. Also, page faults are handled by software because the overhead will be small compared to the access time of the disc. Since the writes take too long, virtual memory systems use write-back.

3.1 Page Table Organization and Entries

Mapping information is organized into *page tables*, which are a collection of *page table entries*. Each program has its own page table which resides in the memory. The page table can be a single-level table or a hierarchical page table. Single-level page tables were used long ago, when the requirements of a memory space were small. Nowadays, a hierarchical structure is followed. A typical

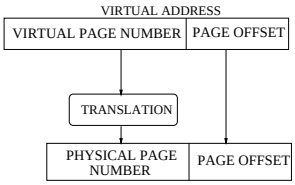

Figure 5: The Virtual Address translated to a Physical Address. An important aspect of such a translation is that the Offset does not change for both, the virtual address and the physical address.

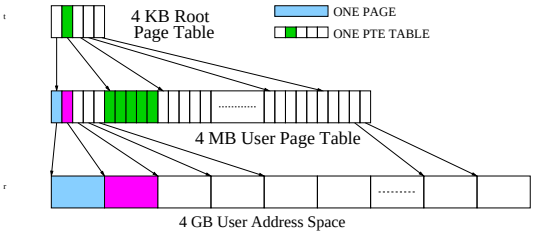

Figure 6: A hierarchical page table structure (from [5]).

hierarchical structure is shown in Figure 6, and it is based on the idea that a large data array can be mapped by a smaller array, which in turn can be mapped by an even smaller array. For example, if we assume 32-bit addresses, byte addressing, and 4-Kbyte pages, the 4-Gbyte memory address space is composed of a million pages. As seen in the figure, by using a two-level page table hierarchy, the root-level table, the one which is stored in main memory, is of considerable smaller size compared to the root-level table which would have been generated if a single-level hierarchy were used.

Each page table entry typically maintains information about one page at a time. It contains information that tells the operating system the following things:

- The ID of the page's owner i.e. the process to which this page belongs.
- The virtual page number.
- The location of the page, whether it is in memory or in disc.
- A reference bit to signify whether it was accessed in the near past.
- Page-protection bits which determine the level of accessibility of the page.
- Modify bit to signify whether the page was written to or not.

Thus, all the information needed by the operating system is stored in the page table entry.

3.2 Translation Look Aside Buffers

Since pages are stored in main memory, every memory access by a program can take at least twice as long: once to fetch the physical address and the other access to fetch the data. This results in reduction of speed by a factor of two. To improve access performance, one must utilize the locality of reference of a page table which says that if a translation of a page number is needed, it will probably be needed again. To Improve speed of translation, most

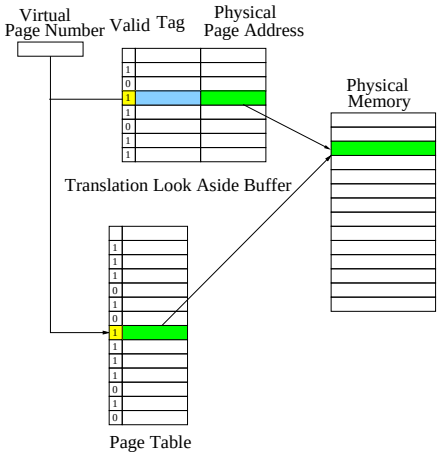

Figure 7: The Translation Look aside Buffer is a small cache, which can typically hold 32 - 64 page table entries (from [19]).

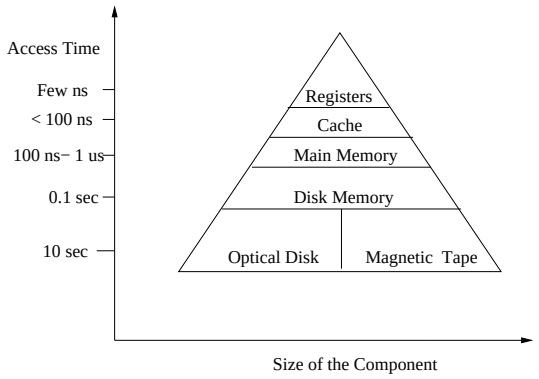

Figure 8: Memory hierarchy (from [21]).

of the modern-day systems provide a cache for page table entries called the *Translation Look aside Buffer*, or commonly called the TLB. The TLB contains a tag entry, which stores a portion of the Virtual address, and a data entry, which contains the corresponding physical page number. Such a type of organization is shown in Figure 7. On every physical reference, we look up the virtual page number into the TLB. If we get a hit, the physical address used to form the address is fetched immediately. In case of a miss, it is determined, either by hardware or software whether it is a page fault or a TLB miss. The TLB miss rate should be small. Hence TLBs are normally fully-associative caches of sizes varying from 32 to 4096 entries.

4 Technology

The speed of a level of the memory hierarchy is largely affected by the technology employed. This section introduces the various technologies employed, and gives a brief overview of various implementations.

Figure 8 shows the relation between component size and access times for registers, caches, main memory and auxiliary storage devices such as magnetic disks. The amount of investment that can be made and the speed of the technology that is to be employed for a particular level of memory system is dependent on the relative frequency of accesses to that level. A faster memory like the register file is used at the upper level. However, registers are expensive and so larger amounts of memory are built using SRAMs and DRAMs. The next level of hierarchy stores large amounts of data at a smaller cost. The accesses to it are few, and so a slower and less expensive memory like the disk memory is used.

SRAM Technology

In SRAM technology, the binary values are stored using traditional flip-flop logic gate configurations. SRAMs can store data as long as the power supply is available. Its operation doesn't involve any charging phenomena, therefore it is fast. As it can store a binary 1 without discharging, no refresh circuitry is needed. However, its structure requires 6 transistors for storing a single bit, making it expensive to use it for larger memories.

DRAM Technology

In the DRAM technology, the data is stored as charge on the capacitors. The presence or absence of data is treated as a 1 or 0. Because of the natural tendency of capacitors to discharge, the DRAM cell needs periodic refreshment. As the DRAM stores data as a charge on the capacitors, it is slower compared to the SRAM. As each DRAM cell can be made out of a single transistor, it is more dense and less expensive compared to SRAMs.

A comparison between SRAM and DRAM can be seen in Table 3. Due to the higher speed of operation of SRAMs, SRAMs can be used for faster memories such as cache memories. DRAM technology is preferred for implementing main memory. However, recent trends indicate that DRAM will become the preferred choice even as onchip memory.

The performance of a DRAM cell can be improved by performing architectural modifications to the basic organization of a memory array.

In order to eliminate the need for frequent refreshment, the DRAM memory is organized so that multiple cells can be refreshed at the same time. This is achieved by using a square organization as shown in Figure 9.

DRAM memory speeds can be improved by using special techniques listed below. More about these techniques can be found in [4] and [19]

Two Step Decode Process
The DRAM memory chip is organized as a square or rectangular block with the number of rows almost equal, but less than that of the columns. The address is split into two parts, and the Row address and column address are selected in successive cycles. Hence, a single row can be selected and refreshed at a time.

Support for Fast Page Mode access
The DRAM memory is organized such that a single row address can be used for making multiple accesses to the various columns of the same row, such that the Row address needs not be changed for every reference, thus reducing the delay. This DRAM is called Fast Page Mode DRAM.

However, the Column Address Strobe needs to be changed every time. This can be eliminated by removing the need to toggle the column address strobe every time. This technique is called Static Column Technique.

Extended Data Out DRAM(EDO DRAM)
The Page Mode DRAMs have the drawback that the column address for the next request can't be latched in the column address buffer until the previous data is gone. The EDO DRAM eliminates this problem, by using a latch between the output pins and the sense amplifier which holds the state of the output pin, thus fa-

Technology	Nature	cost	Transistors/bit	Refreshing	capacity
SRAM	Static	8-16 times that of a DRAM	4 to 6	Not needed	less
DRAM	dynamic	less	1	Needed (Every 2 ms,typically)	4-8 times that of SRAM

Table 3: Comparison of the typical SRAMs and DRAMs.

cilitating the Column address to be deasserted as soon as the output is latched. Hence, the precharging of the memory array can occur soon improving the latencies and in turn the access time.

All the above techniques use an asynchronous control scheme and utilize hardware already present on the circuit to give an improvement in bandwidth of about 2-4 times. An alternative is to use a synchronous control scheme that enables the DRAM to latch information to and from the control bus during the clock signal transition. Synchronous DRAM (SDRAM), Enhanced synchronous DRAM (ESDRAM) and Synchronous link DRAM (SLDRAM) belong to this class. These techniques can be used to achieve much better gains than the Page mode techniques. The SDRAM has a programmable register which stores the number of bytes per request value and returns many bytes extending over several cycles. Hence it eliminates the delay involved in charging the column access strobes for successive accesses of data from the same row. The ESDRAM is much like the SDRAM with an EDO DRAM advantage built into it. Hence, the internal timing parameters are an improvement over SDRAM, and thus help facilitate faster access of data.

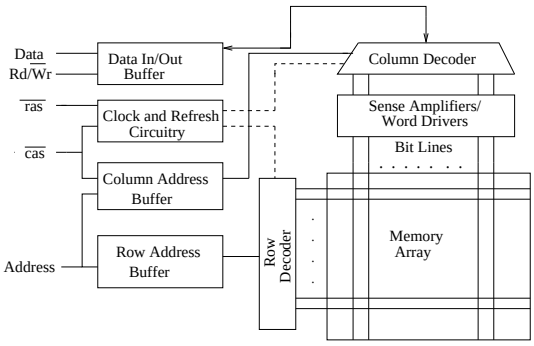

Figure 9: Conventional DRAM block diagram (from [4]).

The above mentioned techniques are used to improve the performance of the conventional DRAM structure shown in Figure 9. Even though the performance gains achievable with the above techniques are good, the increasing rate of processor speeds demand even higher performance. The RambusDRAM(RDRAM) and the Direct RambusDRAM(DRDRAM) are deemed to be the DRAM technologies of modern generation.

Rambus DRAM
This new breed of DRAM produced by RAMBUS, uses the standard DRAM core and provides a new interface which causes the chip to act like a memory system instead of a memory component. The chip can perform its own refresh and can send variable amounts of data per request. RDRAM uses a packet-switched or split transaction bus instead of the row/column address strobe employed by a typical DRAM.

Direct Rambus DRAM
This is also similar to a Rambus DRAM except for the use of half row buffers instead of full row buffers. This causes the die area occupied by the row buffers to be smaller, which in turn reduces the cost, even though at the expense of increased miss rates. Further, it uses a 3 byte wide channel (2 bytes for address and 1 byte for data) as compared to an RDRAM that uses a byte wide channel. Since the channel is split into 3 parts, a DRDAM bus can be used by three transactions at a time, which can improve performance.

5 Compiler Techniques

In this section we discuss the role of a compiler in exploiting memory system resources efficiently.

The compiler contributes to improving performance by:

- overlapping memory latencies with prefetch instructions
- reordering instructions to improve locality

Prefetching In software prefetching, the compiler brings data blocks into the cache so that regular loads/stores can result in hits [13]. When the block is actually referenced it is actually found in the cache, rather than causing a cache miss. Prefetching of useless data/instructions can cause cache pollution and an increase in miss rate.

Instruction reordering The compiler can also perform optimizations like code reordering so as to reduce conflict misses. McFarling [10] looked at using profile information to determine likely conflicts between groups of instructions. This type of reordering was found to reduce the miss rate by 50% for a 2 KB direct mapped I-cache and by 75% in case of 8 KB cache.

Program transformations can also be performed to improve spatial and temporal locality of data. Some of the transformations that can be used are given below.

Merging arrays

This technique aims at reducing cache misses by improving spatial locality. The accesses to multiple arrays in the same dimension with same indices have the potential to interfere, leading to conflict misses. The compiler combines the independent matrices into a single compound array so that a single cache block can contain the desired elements.

Loop Interchange

This compiler technique reorganizes the nesting of loops, to make the code access the data in the order in which it is stored. Thus this technique improves the spatial locality.

Loop Fusion

This technique combines different sections of the code, so that the data that are fetched into the cache can be used repeatedly before being swapped out. Hence this technique exposes temporal locality.

Blocking

It is a technique in which the compiler performs reordering of instructions in order to increase spatial and temporal locality in the sequence of memory accesses. Particularly, the iterations of loop nests are reordered to move multiple references to a single location closer in time (Temporal Locality) or to move references to adjacent memory locations closer in time (Spatial Locality).

6 Reconfigurable Memory

This section examines some of the latest designs and ideas that implement some kind of modularity and reconfigurability, at a memory hierarchical level, to suit a variety of applications. For example, applications that need large memory accesses use configurations involving allocation of larger memory.

6.1 FlexCache

A fully automated software-only solution for caching is designed in the MIT Raw Machine [12] and suggested in the FlexCache framework [11]. The key idea in software caching is the address translation that is inserted at compile-time through a software-managed translation table. Apparently, software-managed caching provides more flexibility with the tradeoff of software mapping costs. The software mapping cost however can be efficiently eliminated as shown in the Hot Pages compiler [11].

Architecture Support for Software Cache Partitions The Hot Pages techniques [12] reduced the software overhead per memory access, in a software managed cache to four operations by register promoting address translation of memory accesses that likely result in cache hits. With architecture support a FlexCache completely eliminates the four operation overhead. Figure 10 shows three different implementations of the Hot Pages loads. If the Hot Pages load is executed on a single-issue processor the additional overhead is 4 instructions. Executing the same code on a superscalar processor and exploiting ILP (Instruction Level Parallelism) reduces this overhead to two instructions. This figure also shows that this code, if implemented as a special memory instruction, *hplw*, can eliminate the runtime overhead completely.

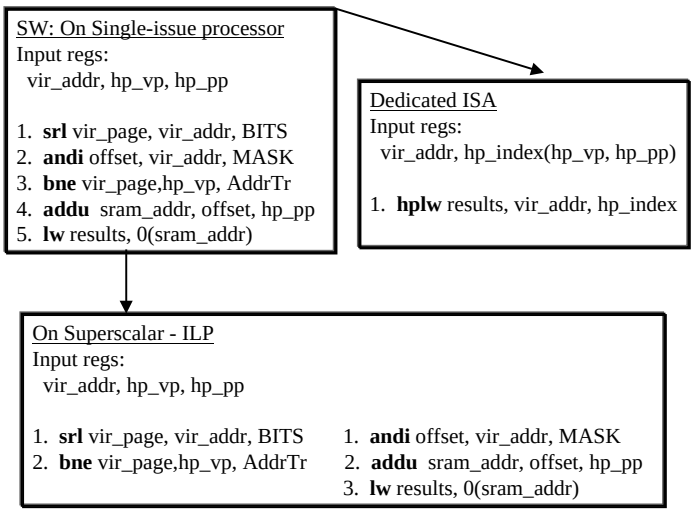

Figure 10: Three different possible implementations of Hot Pages loads. First, we have a load with 4 additional instructions overhead on single-issue processors. On a superscalar processor this overhead is reduced to two instruction per load because of the ILP in the check code. With the FlexCache ISA the overhead is completely eliminated.

6.2 Smart Memories: A Modular Reconfigurable Architecture

A Smart memory chip is made up of many processing tiles, each containing local memory, local interconnect, and a processor core. For efficient computation under a wide variety of applications, the memories, the wires and the computational model are altered to match the application requirements. This kind of structure, as illustrated in Figure 11, can be thought of as a Distributed Memory Architecture in which an array of processor tiles and on-chip DRAM memories are connected by a packet-based, dynamically routed network. The tile is a compromise between VLSI layout constraints and computational efficiency. Four tiles are grouped together to form a Quad Structure. As seen in Figure 12, a tile contains memory which is divided into small blocks of few KB. Larger blocks are then formed by interconnecting these blocks. Each of these small memory blocks is called a Memory Mat. These mats can be configured to implement a wide variety of caches, from simple, single ported, direct-mapped structures to set-associative multi-banked designs. The dynamically routed crossbar switches connect different tiles to form multi-layer memory hierarchies.

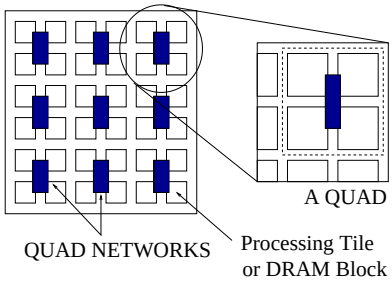

Figure 11: A typical Smart Memories chip (from [9]).

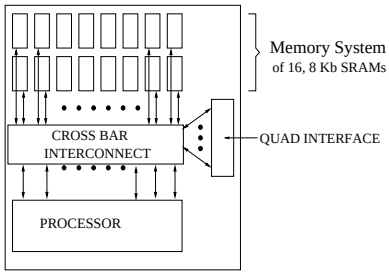

Figure 12: Tile Floor plan (from [9]).

6.3 IRAM: Intelligent RAM

Another trend towards Memory-Processor integration is based on the idea of implementing the processor in the DRAM. The Intelligent RAM (commonly called as IRAM) [17] [18], is an example of such a scheme. Since the performance gap between memories and processors is increasing by around 50 percent per year [19], it has become increasingly vital to find out new techniques for reducing this gap. Thus, most researches believe that it is better to unify logic and DRAM. The reason to put the processor on the DRAM instead of increasing on-chip SRAM, is the fact that DRAMs are much denser than SRAMs. Thus, a larger amount of on-chip memory can be made available as compared to conventional architectures. Other advantages include:

1. *Large potential bandwidth* can be made available. This is possible because DRAMs inherently have large internal bandwidths.

2. *Lower Latencies* result because the processor does not have to go off-chip so often.

3. *Higher energy efficiency* results due to the fact that the processor does not have to drive off-chip buses, which consume most of the energy in the system.

To test the performance of an IRAM, an Alpha 21164 microprocessor was implemented on a DRAM chip [17]. Various benchmarks were then used to evaluate the performance of this kind of implementation. The results obtained are shown in Table 4. The performance for database programs and applications increased by a factor of 50 percent. For the other applications, there was performance degradation. But, this can be credit to the fact that the Alpha was implemented, on the DRAM, along with it's original memory organization. But, in practice a new memory hierarchical structure, better suited for an IRAM, has to be incorporated to give a better performance. The power consumption in an IRAM is illustrated in Table 5. It can be seen that there is an improvement in energy savings.

In spite of the obvious advantages of the IRAM, it too has some shortcomings, which must be overcome. The most critical drawback is the decrease of retention time of DRAM when operating at high temperatures. It decreases by half for every 10 degrees rise in temperature, causing refresh time to go up. Another drawback is the scalability of the system as that the maximum memory obtained from an IRAM is less then 128 Mbytes.

6.4 FlexRAM

Another method of achieving modularity and reconfigurability is to use the current state-of-the-art MLD/(Merged Logic DRAM/) technology for general-purpose computers. To satisfy the requirements of general purpose system with low cost, PIM /(Processor in Memory/) chips are placed in the Memory System and they are made to default to plain DRAM if the application is not enabled for Intelligent Memory. In simple terms, a small but simple processor with limited functionality is implemented in a DRAM chip along with a large amount of memory. This chip is included as a part of the memory hierarchy instead of treating it as a processing element. Thus, when an application needs intelligent memory, it is used as one. But if an application does not require an intelligent memory, the chip defaults to a normal DRAM, where only the memory content of the chip is used. Such a chip is called a FlexRAM [7]. A typical structure of a FlexRAM can be seen in the Figure 13. In a typical FlexRAM model, each chip consists of very simple compute engines called P.Arrays, each of which is interleaved with DRAM macro cells. Each of these P.Arrays have access to limited amount of on-chip memory. A low-issue super scalar RISC core, called P.Mem, is introduced on the chip to increase the usability of the P.Arrays. Without the P.Mem, all the tasks to co-ordinate the P.Arrays would be carried out by the Host processor, P.Host. To keep the ratio of the logic area to DRAM area, P.Array must be very simple. Therefore P.Array supports only integer arithmetic. For maximum programmability, P.Mem and P.Arrays use virtual addressing. To obtain a high bandwidth, multi-arrayed DRAM structure is used. Communication between the P.Host and the P.Mems is achieved by writing to a special memory-mapped location.

6.5 Reconfigurable Caches

This approach mitigates the problem of non-effective utilization of caches by different applications. As it is known, most of the current processor designs devote most of the on-chip area to caches. But,

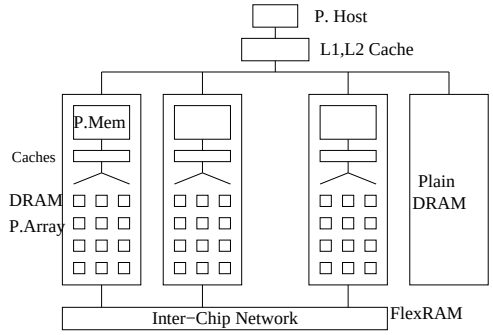

Figure 13: Overall organization of a FlexRAM-based memory system (from [7]).

many workloads do not make effective utilization of the caches. This results in inefficiencies which are proposed to be overcome by using a design that enables on-chip cache SRAM arrays to be dynamically divided into multiple partitions which can be used simultaneously for different processor activities. In contrast to the designs elaborated above, this design uses the conventional cache design with minor hardware and accompanying software. One key aspect of this design is that the reconfigurability is implemented in custom hardware at design time. Thus, only a limited number of possible configurations are possible. Figure 14 shows how a typical 2-Way cache can be divided into a reconfigurable cache which can have at most two partitions. In the case of the reconfigurable as well as the conventional cache, same number of the bits of address field would be used as tag, index, and block offset bits. The only changes to the conventional cache organization are:

1. A reconfigurable cache with up to N partitions must accept up to N input addresses and generate N output data elements with N Hit/Miss signals /(one for each partition/).

2. A special hardware register must be maintained to track the number and sizes of partitions and also control the routing of the above signals.

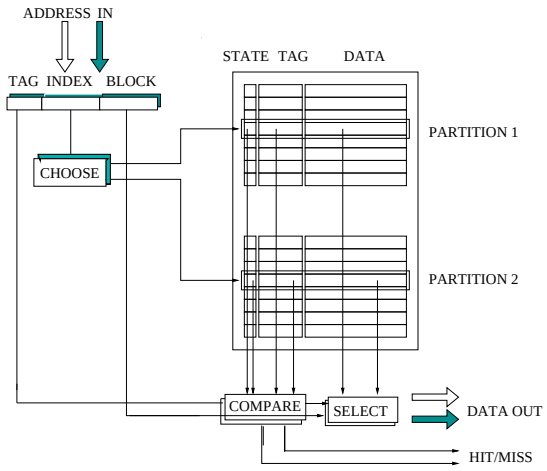

Figure 14: Typical Reconfigurable cache organization for a 2-Way Set Associative cache (from [16]).

Category	SPEC int92		SPECfp92		Database		Sparse	
	Opt.	Pes.	Opt.	Pes.	Opt.	Pes.	Opt.	Pes.
Fraction of time in Processor	0.12	1.57	0.89	1.36	0.30	0.46	0.35	0.54
Fraction of time in I cache misses	0.04	0.05	0.01	0.01	0.18	0.21	0.00	0.00
Fraction of time in D cache misses	0.14	0.17	0.26	0.30	0.15	0.18	0.08	0.10
Fraction of time in L2 cache misses	0.05	0.05	0.06	0.06	0.30	0.030	0.07	0.07
Fraction of time in L3 cache misses	0.00	0.00	0.00	0.00	0.03	0.05	0.06	0.12
Total = ratio of time vs. alpha	1.25	1.83	1.21	1.74	0.85	1.10	0.56	0.82

Table 4: Optimistic and Pessimistic parameters to estimate IRAM performance for four programs (from [17]).

Benchmark	perl	li	gcc	hyfsys	compress
IRAM, 16:1 memory	1.0	1.7	1.8	2.3	1.7
IRAM, 32:1 memory	1.5	2.0	2.5	2.5	4.5

Table 5: Comparison of the energy efficiency of Digital StrongArm memory system with an IRAM memory system (from [17]).

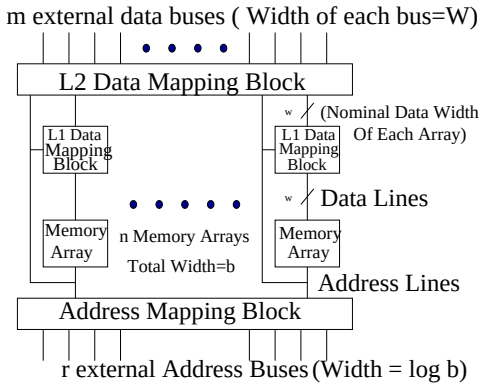

Figure 15: Overall Architecture of a Reconfigurable Memory (from [24]).

6.6 Field Configurable Memory

This section describes a family of Centralized architectures similar to the FiRM FCM described in [14]. This of architecture uses FPGAs to model the reconfigurability of the memory system along with their interconnect networks and the logic associated with it. Before describing the overall architecture of an FCM-based system, it is important to acquaint ourselves with some notations. A set of memories required by an application is referred to as *logical memory configuration*. Each independent memory within a logical memory configuration is referred to as *logical memory*. The architecture, illustrated in Figure 15, consists of *b* bits divided evenly among *n* arrays that can be combined (using the address and data mapping blocks) to implement logical memory configurations. Since each logical memory requires at least one array, one address bus, one data bus, the maximum number of logical memories that can be implemented on this architecture is the minimum of the three.

Flexibility is achieved by: allowing the user to modify the output data bus widths of each array, and by using various combinations of arrays to obtain larger memories. Figure 16 shows a typical configuration in which arrays are combined to implement larger memories. Here four 1024x8 arrays are combined to implement a 1024x32 logical memory. As seen, a single external address bus ID connected to each array, while the data bus from each array is connected to separate external data buses. It is very interesting to note that if a faster memory architecture is needed, there will be a drop in the flexibility of the architecture [24].

6.7 Active Pages: A Computation Model for Intelligent Memory

It is a computation model which addresses the *processor-memory performance gap* by shifting the data-intensive computations to the memory system. In simpler terms, this is a computation model which partitions the applications between a processor and an intelligent memory system. Normally, all data manipulations are offloaded to the logic in memory. To simplify integration with general microprocessors and systems, the interface to the Active Pages is designed to resemble a conventional virtual memory interface. This interface includes standard memory interface functions, a set of functions available for computations on a particular page and allocation functions which assign each Active Page a virtual address.

As mentioned, the application is partitioned between the processor and the Active Pages. This partitioning is relative to the application. If the application uses a lot of floating point operations, then the partiotioning is more *processor-centric*. The goal of such a partitioning is to provide the processor with enough operands to keep it running at peak speeds. If the application has a lot of data manipulations and integer arithmetic, then the partitioning is more *memory-centric*. The goal of such a type of partitioning is exploitation of parallelism and use as many Active Pages as possible. The Active Pages is implemented in RADram (Reconfigurable Architecture Dynamic RAM). The RADram system associates 256 LEs (Logical Elements) to a 512Kbyte of memory block. This can be seen in Figure 17.

7 Low Power Techniques

With the increase in complexity and speed of modern microprocessors, power dissipation is also increasing. Further, the emergence of embedded and portable applications which should have less power dissipation, becomes a motivating factor for developing low-power architectures.

Significant amount of the total power consumption of a modern microprocessor chip can be attributed to the on-chip memory, i.e., the cache memory, and the need to drive the large off-chip buses for accessing the main memory.

Power reduction in cache memory can be achieved by semiconductor process improvement, voltage reduction, optimizing the cache structure for higher energy savings, and the like. The cache structure plays a significant role in determining the maximum power reduction for a given technological improvement. The architectural modifications to the cache structure suggested for mini-

mization of power consumption include:

Simpler Cache Design
As discussed in Section 2 increasing set associativity and line size above a certain limit results in diminishing performance returns. Additionally, larger cachesconsume more power. In general set associative caches with an associativity of 2 to 4 and of sizes ranging from 4K to 16K tend to be the most power efficient. [23]

Novel Cache Designs
It was found that the power consumption of the caches can be minimized by using noveldesign approachessuch as:

1. Vertical Cache Partitioning
   It is a technique [6] in which a buffer is used inbetween the cache and the CPU. The buffer can supply data if the access is to the latest accessed cache block.
2. Horizontal Cache Partitioning
   The data array of a cache is partitioned into several partitions that are powered and can be accessed individually [22]. Hence only the cache partition containing the accessed word is powered.
3. Gray Code Addressing
   The power consumption of buses depend on the number of bit switchings on the bit lines. The use of the Gray code addressing schemes over the conventional 2's complement addressing schemes reduce the number of switchings for instructions located sequentially.

Filter Cache
In this approach a small cache called the *filter cache* [8] is inserted between the L1 cache and processor. The L1 cache is turned off while the data is being supplied by the filter cache. Hence if most of the accesses are supplied by filter cache, significant energy gains can be achieved. However, as the latency to the L1 cache is increased the performance is degraded. Thus this option is feasible when performance can be traded for energy savings.

Loop Cache
During execution of instructions in a loop, the I-cache unit frequently repeats the previous tasks unnecessarily resulting in more power consumption. So, a small cache (typically of 1 KB size) called Loop cache (L-cache) inbetween the I-cache and the CPU is proposed in [2]. Energy reduction is achieved by more judicious use of I-cache during the execution of a loop. As such, this approach will give maximum savings for applications with regular loop patterns.

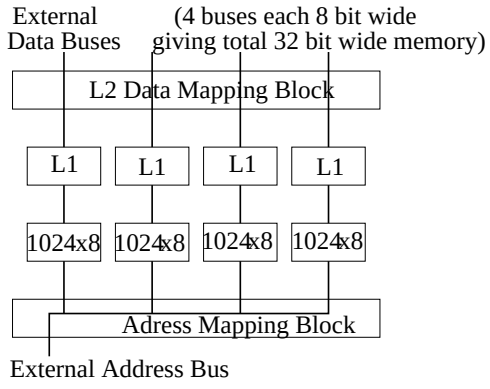

Figure 16: A typical example of a FCM configured for 32-bit wide data bus (from [24]).

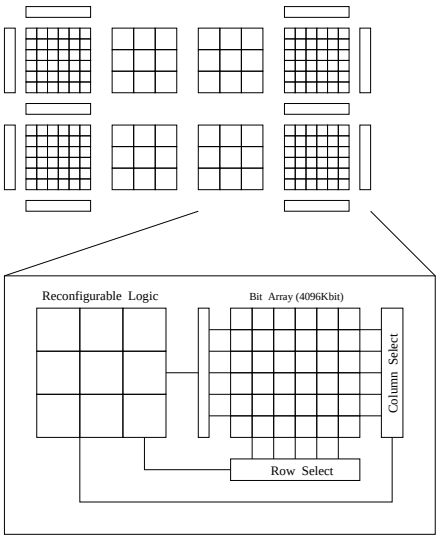

Figure 17: The RADram Structure used for implementing an Active Page (from [15]).

8 Summary

In this paper an overview of memory systems has been given. Emphasis has been put on describing how various components of a memory system affect the overall performance of computer systems, and what new architectural trends are emerging.

References

[1] A. Agarwal and S. Pudar. Column Associative Caches: A Technique for reducing the Miss Rate for Direct-Mapped Caches. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, San Diego, Calif., May 1993. ACM Press.

[2] N. Bellas, I. Hajj, C. Polychronopoulos, and G. Stamoulis. Architectural and Compiler Support for Energy Reduction in the Memory Hierarchy of High Performance Microprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design,1998*, Monterey, Calif., July 98. ACM Press.

[3] I. B. Gianluca Albera. Power/Performance Advantages of Victim Buffer in High Performance Processors. In *Proceedings of the IEEE Alessandro Volta Memorial Workshop on Low-Power Design*, Como, Italy, March 1999. IEEE Computer Society.

[4] B. Jacob, B. Davis, and T. Mudge. A Performance Comparison of Contemporary DRAM Architectures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Atlanta, May 1999. IEEE Computer Society.

[5] B. Jacob and T. Mudge. Virtual Memory: Issues of Implementation. *IEEE Computer*, June 1998.

[6] J.Bunda, W.C.Athas, and D.Fussel. Evaluating Power Implication of CMOS MicroProcessor Design Decisions. In *Proceedings of the International Work Shop on Low Power Design, 1994*, April 1994.

[7] Y. Kang, W. Huang, S.-M. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrelas. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada., June 2000. IEEE Computer Society.

[8] J. Kin, M. Gupta, and W. MangioneSmith. The Filter Cache:An Energy Efficient Memory structure . In *IEEE International Symposium on Microarchitecture,1997*. IEEE Press, December 1997.

[9] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada., June 2000. IEEE Computer Society.

[10] S. McFarling. Program Optimization for Instruction Caches. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, April 1989. ACM Press.

[11] C. A. Moritz, M. Frank, and S. Amarasinghe. FlexCache: A Framework for Flexible Compiler Generated Data caching. In *In Proceedings of the 2nd Intelligent Memory Workshop, IRAM00*, Cambridge, Nov 2000. IEEE Computer Society.

[12] C. A. Moritz, M. Frank, V. Lee, and S. Amarasinghe. HotPages: Software Data caching for Raw Microprocessors. In *MIT LCS Technical Memo LCS-TM-599*, Cambridge, Aug 1999.

[13] T. Mowry and C.-K. Luk. Co-operative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, Boston, Massachusetts, April 1989. ACM Press.

[14] T. Ngai, J. Rose, and S. Wilton. An SRAM-Programmable Field-Configurable Memory. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, Santa Clara, Calif., May 1995. IEEE Micro.

[15] M. Oskina, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, Barcelona, Argentina., June 1998. IEEE Computer Society.

[16] N. P. J. Parthasarathy Ranganathan, Sarita Adve. Reconfigurable Caches and Their Application to Media Processing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, Vancouver, Canada., June 2000. IEEE Computer Society.

[17] D. Patterson, T. Andersson, N. Cardwell, R. Fromm, K. Keeton, and C. K. and. A Case for Intelligent RAM: IRAM. *IEEE Micro*, April 1997.

[18] D. Patterson, K. Asanovic, A. Brown, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, C. Kozyrakis, D. Martin, S. Perissakis, R. Thomas, N. Treuhaft, and K. Yelick. Intelligent RAM: The Industrial Setting, Applications, and Architectures. In *Proceedings of the 27th International Conference on Computer Design*, Austin, Texas, October 1997. IEEE Computer Society.

[19] D. A. Patterson and J. L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[20] Przybylski. Cache design: A performance-directed approach. Morgan Kaufmann Publishers, San Monteo, Calif., 1990.

[21] W. Stallings. *Computer Organization and Architecture*. Prentice Hall International, 1996.

[22] C. L. Su and A. M. Despain. Cache Design Trade-offs for Energy Efficiency . In *Proceedings of the 28th Hawaii International Conference on System Science,1995*, 1995.

[23] C. L. Su and A. M. Despain. Cache Design Trade-offs for Power and Performance Optimization . In *Proceedings of the International Symposium on Low Power Electronics and Design,1995*, Monterey, Calif., 1995. ACM Press.

[24] S. Wilton, J. Rose, and Z. G. Vranesic. Architecture of Centralized Field-Configurable Memory. In *Proceedings of the 3rd ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, Monterey, Calif., February 1995. ACM Press.