Chapter 5

Overview

- We wish to create the illusion of a large memory that we can access as fast as a very small memory
- The principle of locality states that programs access a relatively small portion of their address space at any instant of time:

Temporal/Spatial Locality

  - Temporal locality:
    - If a data location is referenced then it will tend to be referenced again soon.
  - Spatial locality:
    - If a data location is referenced, data locations with nearby addresses will tend to be referenced soon
- We take advantage of locality by implementing a memory hierarchy:
  - A structure that uses multiple levels of memories; as the distance from the processor increases, the size of the memories and the access time both increase

Memory technologies

- Memory technologies:
  - Cache: SRAM; 0.5 - 2.5 ns access time
  - Main memory: DRAM; 50 - 70 ns access time
  - Lowest level: Magnetic Disk; 5,000,000 - 20,000,000 ns
- Goal is to present the user with as much memory as is available int he cheapest technology while providing access the speed offered by the fastest memory

Memory Hierarchies

- The data residing in a higher level of the memory hierarchy is a subset of any level further down
- The minimum unit of information that can be either present or not present in a cache is a **block** or **line**
- If the data requested by the processor appears in some block in the upper level (cache), this is called a “hit”
- The **hit rate** is the fraction of memory accesses found in the proper level
  - The **miss rate** (1 - hit rate) is the fraction of memory accesses not found in the upper level
- **Hit time**: The time required to access a level of the memory hierarchy, including the time needed to determine where the access is a hit or a miss
  - **Miss penalty**: The time required to fetch a block into a level of the memory hierarchy from a lower level, including the time to access the block, transmit it from one level to the other, insert it in the level that experienced in the miss, and then pass the block to the requester

Caches

- Cache is the level of the memory hierarchy between the processor and main memory
- A direct mapped cache is a cache structure in which each memory location is mapped to exactly one location in the cache
- The mapping to find a block in the cache is given by:
  - (Block Address) modulo (Number of blocks in the cache)

There are three sorts of cache misses; compulsory, capacity, and conflict/collision

- Compulsory misses are caused by the first access to a block that has never been in the cache (cold-start)
- Capacity misses are caused when the cache cannot contain all the blocks needed during the execution of a program
- Conflict misses are caused by set-associative or direct-mapped caches when multiple blocks compete for the same set.

- Each cache location can contain the contents of a number of different memory locations
    - Use tags to identify whether a word in the cache corresponds to the requested word
        - The tag is a field in a table used for a memory hierarchy that contains the address information required to identify whether the associated block in the hierarchy corresponds to a requested word
            - The tag needs only to contain the upper portion of the address, corresponding to the bits that are not used as an index/offset into the cache
- We need a valid bit as well that indicates that the associated block in the hierarchy contains valid data
    - This is used for cases like when you initially start up the processor and the cache does not have valid information

Field breakdown

- The index of a cache block, together with the tag contents of that block, uniquely specifies the block address of the data contained in the cache block
- If we have the following:
    - 32-bit byte addresses
    - A direct mapped cache
    - The cache size is 2^n blocks so *n* bits are used for the index
    - The block size is 2^m words (2^{m+2} bytes), so *m* bits are used for the word within the block and two bits are used for the byte part of the address
        - The size of the tag field is: $32 - (n + m + 2)$
        - Total number of bits in a direct-mapped cache:
            - 2^n x (block size + tag size + valid field size)
- The address of the block is given by:
    - (Byte Address) {floor} (Bytes Per Block)
        - This contains all addresses between [(Byte Address) {floor} (Bytes Per Block) x Bytes per Block] and [(Byte Address) {floor} (Bytes Per Block) x Bytes per Block + (Bytes per block - 1)]

Optimizations

- Larger blocks exploits spatial locality to lower miss rates
    - The miss rate may go up eventually if the block size becomes a significant fraction of the cache size
        - The number of blocks that can be held in the cache will become small
        - A block will be bumped out of the cache before many of its words are accessed
- The cost of a miss increases

Increasing associativity decreases conflict misses, and full-associativity eliminates it completely, but this may affect access time.
Capacity misses can be easily reduced by enlarging the cache, but that may affect access time.
Compulsory misses are generated by the first reference to a block so it can be reduced by increasing the block size. This will reduce the number of references required to touch each block of the program once, because the program will consist of fewer cache blocks. Increasing the block size too much can have a negative effect because of the increase in miss penalty.

  - The miss penalty is determined by the latency to the first word and the transfer time for the rest of the block
    - The bigger the block size, the grater the transfer time
- One way to improve upon this is an early restart
  - Simply resume execution as soon as the requested word of the block is returned rather than wait for the entire block

Cache Misses

- The control unit must detect a miss and process the miss by fetching the requested data from memory or lower-level cache
  - Cache miss handling is done in collaboration with the processor control unit and a separate controller that initiates the memory access and refills the cache
- A miss the pipeline, essentially freezing the contents of the registers while we wait for memory

- In an instruction miss, we instruct main memory (lower level memory) to perform a read on PC - 4 (the PC is incremented in the first clock cycle of execution)
- We wait for the memory to complete its access
- We write the cache entry, putting the data from the memory into the data portion f the cache entry, and writing the upper bits of the address (from the ALU) into the tag field and turn the valid bit on
- We restart the instruction execution at the first step, which will refetch the instruction
  - We will then get a cache hit this time and proceed as normal

Cache Writes

- In writes, say we wrote the data into only the data cache (without changing main memory)
  - The cache and memory are said to be inconsistent
- The simplest way to keep the main memory and the cache consistent is by the **write-through** approach:
  - A scheme in which writes always update both the cache and the next lower level of the memory hierarchy, ensuring that data is always consistent between the two
    - Every write causes the data to be written to main memory, which leads to massive loss of time/performance
  - We could use a **write buffer**, which is a queue that holds data while the data is waiting to be written to memory
    - If the rate at which the memory can complete writes is less than the rate at which the processor generates them, or if the writes occur in bursts, there is nothing we can do to prevent stalls

- Another approach is the **write-back** approach:
    - A scheme that handles writes by updating values only to the block in the cache, then writing the modified block to the lower level of the hierarchy when the block is replaced
- In a write-through cache, writes can always be done in one cycle
    - We read the tag and write the data portion of the selected block
        - If the tag matches the address of the block being written, the processor can continue normally, since the correct block has been updated
        - If the tag doesn't match, the processor generates a write miss to fetch the rest of the block corresponding to that address
- In a write-back cache, writes take two cycles
    - We cannot overwrite automatically like in a write-through because we don't have consistency with the lower levels in memory
        - We would lose the data if the tags don't match up
    - We need one cycle to determine if it's a hit followed by a cycle to actually perform the write
    - We could use a store buffer that effectively allows the write-back to only take one cycle by letting the buffer take care of the write on the next unused cache access cycle

Performance

- CPU time can be divided into clock cycles that the CPU spends executing the program and the clock cycles that the CPU spends waiting for the memory system
    - CPU time = (CPU execution clock cycles + Memory-stall clock cycles) x Clock cycle time
        - Assume that cost of cache accesses that are hits are part of normal CPU execution cycles
- Memory-stall clock cycles = Read-stall cycles + Write-stall cycles
- Read-stall cycles:
    - (Reads/Program) x Read miss rate x Read miss penalty
- Write-stall cycles
    - [(Writes/Program) x Write miss rate x Write miss penalty] + Write buffer stalls
- We will assume that write buffer stalls are negligible and that read and write miss penalties are the same. Therefore:
        - Memory-stall clock cycles = (Memory access/Program) x Miss Rate x Miss Penalty
        - Memory-stall clock cycles = (Instructions/Program) x (Misses/Instruction) x Miss Penalty
- If the processor is made faster, but the memory system is not, then the amount of time spent on memory stalls will take up an increasing fraction of the execution time

Write-through advantages include:
misses are simpler and cheaper because they never require a block to be written back to the lower level.
easier to implement than write-back , although to be practical, still need to use a write buffer.
Write-back advantages include:
Individual words can be written by the processor at the rate that the cache, rather than the memory, can accept
Multiple writes within a block require only one write to the lower level in the hierarchy
When blocks are written back, the system can make effective use of a high-bandwidth transfer, since the entire block is written.

- We must also take into account the hit time, because the hit time can definitely increase if, for example, we have a larger cache
- Therefore, we evaluate caches by their average memory access time:
    - AMAT = Time for a hit + Miss rate x Miss penalty

Fully Associative Caches

- At the other extreme from direct-mapped caches are fully associative caches
    - A cache structure in which a block can be placed in any location in the cache
- To find a given bock in a fully associative cache, one must search all entries in the cache
    - Even though the search is done in parallel, there is comparator associated with each cache entry
        - This increases the hardware cost, effectively making fully associative placement practical only for caches with small numbers of blocks

Set Associative Caches

- The middle range is a set associative cache
    - A cache that has a fixed number of locations where each block can be placed
    - An n-way set associative cache consists of a number of sets, each of which consists of n blocks
        - Each block in memory maps to a unique set in the cache given by the index field, and a block can be placed in any element of the set
            - Combines direct-mapped (directly mapped to a set) with fully associative (search all blocks in the set)
    - The set containing a memory block is given by:
        - (Block Number) modulo (Number of sets in the cache)
- Can think of a direct mapped cache as a one-way set associative cache and a fully associative cache with m entries as an m-way set associative cache
- The advantage of increasing degree of associativity is to decrease miss rate

- To find a block in a cache that is set associate, we index into the appropriate set
    - The tag of every cache block within the appropriate set is checked to see if it matches the block address from the processor
- If the total cache size is kept the same, increasing the associativity increases the number of blocks per set, which is the number of simultaneous compares need to perform the tag search in parallel
    - Each increase by a factor of 2 in associativity doubles the number of blocks per set and halves the sets
        - Accordingly, there is one less index bit and one more tag bit

There are three main ways of placing blocks - direct-mapped, set-associative, and fully associative. The advantage of increasing the degree of associativity is that it usually decreases the miss rate. This comes from reducing misses that compete for the same location. Potential disadvantages include increased cost and slower access time.

- The choice among direct-mapped, set-associative, or fully associative mapping in any memory hierarchy will depend on the cost of a miss versus the cost of implementing associativity, both in time and in extra hardware

Replacement

- In a set-associative cache, unlike a direct mapped cache, blocks that are brought up due to cache misses have multiple options to be placed in
- Most commonly used scheme is **least recently used** (LRU)
    - The block replaced is the one that has been unused for the longest time
- LRU replacement is implemented by keeping track of when each element in a set was used relative to the other elements in the set
    - This gets quite complex and inefficient for larger set sizes

L2 Cache

- Most microprocessors support an additional level of caching
    - This second-level cache is usually on the same chip and accessed whenever a miss occurs in the primary cache
    - If the second-level cache contains the desired data, the miss penalty for the first-level cache will be essentially the access time of the second-level cache
        - If neither levels have the data, then a main memory access is required and thus incurs a larger miss penalty
- A two -level cache structure allows the primary cache to focus on minimizing hit time to yield a shorter clock cycle or fewer pipeline stages, while allowing the secondary cache to focus on miss rate to reduce the penalty of long memory access times
- The global miss rate is the fraction of references that miss in all levels of a multilevel cache and dictates how often we must access the main memory

Virtual Memory

- The main memory can act as “cache” for the secondary storage (which is usually implemented with magnetic disks)
    - This technique is known as virtual memory
        - Allows for efficient and safe sharing of memory among multiple programs
        - Removes the programming burdens of a small, limited amount of main memory
- For multiple programs to share the same memory, running all at once on a computer, they can only read and write portions of main memory that have been assigned to it
    - Just like a cache, main memory need contain only the active portions of the many programs

The two primary strategies for replacement in set-associative or fully-associative caches are random replacement and least recently used. LRU, although ideal, is too costly to implement for hierarchies with more than a small degree of associativity, since tracking the usage information is costly. For larger associativity, either LRU is approximated or random replacement is used. As the caches become large, the absolute difference between the two strategies becomes small. In virtual memory, some form of LRU is always approximated, since even a tiny reduction in the miss rate makes a significant dent given the cost of miss being so enormous.

      - Therefore, virtual memory is enabled by the principles of locality
- We cannot know at compile-time which programs will share the memory with other programs
  - All interaction is dynamic
    - We would like to compile each program into its own address space
      - Virtual memory implements the translation of a program's address pace to physical addresses (addresses in main memory)
      - This translation process enforces protection of a program's address space from other programs
- Virtual memory also allows a single user program to exceed the size of physical memory
  - Before, programmers had to divide the program into mutually exclusive modules and account for the lack of main memory themselves

Fundamentals

- A virtual memory block is called a page
- A virtual memory miss is called a page fault
  - This occurs when an accessed page is not present in main memory
- With virtual memory, the processor produces a virtual address
  - This address corresponds to a location in virtual space and is translated by address mapping to a physical address
    - This physical address is used to access main memory
- Address translation is the process by which a virtual address is mapped to an address used to access memory.
- Both the virtual memory space and the physical memory space are broken into pages, so that a virtual page is mapped to a physical page
  - It is possible for a virtual page to be absent from main memory
    - In this case, the page resides on the disk
- Virtual memory simplifies loading the program for execution by providing relocation
  - Relocation maps the virtual addresses used by a program to different physical addresses before the addresses are used to access memory
    - This allows us to load the program anywhere in main memory
    - Because everything is in terms of fixed-size pages, there is no need for a contiguous block of memory to allocate to a program
      - Instead, the operating system need only find a sufficient number of pages in main memory

Virtual memory is the name for the level of memory hierarchy that manages caching between the main memory and disk. Virtual memory allows a single program to expand its address space beyond the limits of main memory. More importantly, virtual memory supports sharing of the main memory among multiple, simultaneously active processes, in a protected manner.

Translation

- In virtual memory, the address is broken into a virtual page number and a page offset
- In the translation from virtual to physical, we convert the virtual page number to the physical page number and leave the page offset untouched
    - The physical page number accounts for the upper bits while the page offset is the lower bits
    - The number of bits in the page offset determines the page size
- The number of pages addressable with the virtual address doesn't need to match the number of pages addressable with the physical address
    - That is the purpose of virtual memory
    - Therefore, the VPN may have more bits than the PPN
- Sometimes, the processor address size is small relative to the actual physical memory
    - In these cases, no single program can benefit, but a collection of programs running at the same time from not having to be swapped to memory or by running on parallel processors

Key Optimizations

- Because of the enormous miss penalty for page faults, dominated by the time to get the first word for typical page sizes, there are several key decisions:
    - Pages should be large enough to try to alleviate the high access time.
    - Organizations that reduce the page fault rate are attractive.
        - Primary technique is to allow fully associative placement of pages
    - Page faults can be handled in software because the overhead will be small compared to the disk access time.
        - Software can afford to use clever algorithms for choosing how to place pages because even small reductions in the miss rate will pay for the cost of the algorithms
    - Write-through will not work because of the steep miss penalty. Instead we will implement write-back.

- Because of the high penalty for page faults, we optimize page placement to reduce frequency of page faults
    - Allow a virtual page to be mapped to any physical page
        - That way, the OS can use sophisticated algorithms and such that track page usage to replace a page that is the most beneficial (not used in a long time)
- Because the physical memory is now fully-associative, we need an efficient way of locating pages
    - Can't do a full search because that is just too intensive
    - Instead, locate pages by using a table that indexes the memory called a **page table**

Managing the memory hierarchy between the main memory and disk is challenging because of the high costs of page faults. Several techniques are used to reduce the miss rate:

- Pages are made large to take advantage of spatial locality and to reduce the miss rate.
- The mapping between virtual addresses and physical addresses, which is implemented with a page table, is made fully associative so that a virtual page can be placed anywhere in main memory.
- The operating system uses techniques, such as LRU and a reference bit, to choose which pages to replace.

Page Table

        - The page table contains the virtual to physical address translations in a virtual memory system
        - The page table is stored in memory and is indexed by the VPN
            - Each entry in the table contains the PPN for the virtual page if the page is currently in memory
- Each program has its own page table that maps the virtual address space of that program to main memory
    - The page table may contain entries for pages not present in memory
- There is a valid bit in each page table entry
    - If the valid bit is off, the page is not present in main memory and a page fault has occurred
- The page table contains a mapping for every possible virtual page so no tags are required

State

- To indicate the location of the page table in memory, the hardware includes a register that points to the start of the page table
    - This is known as the page table register
- Each program has state
    - Defined as the page table, the PC and the registers
- To switch between processes, the OS must save this state and then restore it to continue execution
- Rather than save the entire page table, the OS simply loads the page table register to point to the page table of the process it wants to make active

Page Faults

- If the valid bit for a virtual page is off, then a page fault occurs
    - The OS must be given control via an exception mechanism
    - It must find the page in the next level in the hierarchy (traditionally the magnetic disk) and decide where to place he requested page in main memory
- The virtual address does not tell us immediately where the page is stored on the disk
- We do not know ahead of time when a page in memory will be replaced
    - OS creates a swap space for each process
        - This is the space on the disk reserved for the full virtual memory space of a process (all the pages of a process)
    - Also creates a data structure to record where each virtual page is stored on the disk
        - Could be part of the page table or an auxiliary data structure
- OS must also create a data structure to track which processes and which virtual addresses use each physical page
    - If a page fault occurs and all of the memory is used, then choose a place to replace using LRU to minimize the number of page faults
    - Replaced pages are written to swap space on disk

Writes to disks are expensive, so virtual memory uses a write-back scheme an also tracks whether a page is unchanged (using a dirty bit) to avoid writing unchanged pages back to disk.
The virtual memory mechanism provides address translation from a virtual address used by the program to the physical address space used for accessing memory. This address translation allows protected sharing of the main memory and provides several additional benefits, such as simplifying memory allocation.

- Implementing a completely accurate LRU is too expensive, so instead, one alternative is that:
    - OS keeps track of which pages have and which pages have no been recently used
    - Mark this with a reference bit, or use bit
    - Periodically clears the reference bits and later records them to determine which pages were touched during a particular time period
        - With this information, the OS can select a page that is among the least recently referenced (reference bit is off)

Write-Back

- For writes, because of the millions of processor clock cycles it takes to get to disk, we forgo write through for write back
    - An optimization is that we only write back the pages that have been modified or written since it was read into memory
        - We keep a dirty bit in the page table that is set whenever any word in a page is written

TLB

- Since page tables are stored in main memory, every memory access by a program can take at least twice as long
    - One access to obtain physical address and second access to get the data
- To optimize the locality of the references to words and how it relates to pages, we include a special cache called the **translation-lookaside buffer**.
    - Like a little piece of paper we use to record the location of a set of books we look up in the card catalog
    - It is a cache that keeps track of recently used address mappings to try to avoid an access to the page table
- Each tag entry of the TLB holds a portion of the VPN and each data entry holds a PPN
    - The TLB will need to include other status bits, such as dirty, and the reference bits
- On every reference to a memory address, we look up the VPN in the TLB
    - If there is a hit, the PPN is used to form the address and the corresponding reference bit is turned on
        - If it is a write, the dirty bit is turned on as well
    - If there is a miss, we must determine if it is a page fault or just simply a TLB miss
        - If the page exists in memory, then the TLB miss indicates that only the translation is missing
        - Load the translation from the page table into the TLB and then try to reference again

Ensuring that the processes are protected from each other requires that only the OS can change the address translations, which is implemented by preventing user programs from changing the page tables. Controlled sharing of pages among processes can be implemented with the help of the operating system and access bits in the page table that indicate whether the user program has read or write access to a page.

TLB Misses

- Because the TLB has many fewer entries than number of pages in main memory, the TLB misses will be much more frequent than the true page faults
- After a TLB miss occurs, and the missing translation has been retrieved from the page table, we choose a TLB entry to replace
    - The only bits of the TLB that can be changed are the dirty bits and the reference bits, so these are the only bits that need to be copied back into the page table
        - We use a write-back approach here
- The amount of associatively in the TLB depends on the size of the TLB and the architecture
    - It is hard to maintain what is needed for an LRU for the TLB so often at times the system will randomly choose an entry to replace

- To maintain the hierarchy, when a page is migrated to the disk, the OS will flush the contents of that page from the cache.
    - This is so because if the page is not in main memory, it cannot be in the cache either
    - At the same time, the OS modifies the page tables and TLB, so that an attempt to access any data on the migrated page will generate a page fault.

Scenarios

- The best case scenario:
    - A virtual address is translated by the TLB and sent to the cache where the appropriate data is found, retrieved, and sent back to the processor
- The worst case scenario:
    - A reference misses in all three components of the memory hierarchy: the TLB, the page table, and the cache

Protection

- The protection mechanism must ensure that although multiple processes are sharing the same main memory, one renegade process cannot write into the address space of another user process or into the OS
    - The write access bit in the TLB can protect a page from being written
- The hardware must provide the following:
    - Support at least two modes - a supervisor process (or a kernel process) for the OS and a user process
    - Provide portion of the processor state that a user process can read but not write
        - Includes the user/supervisor mode bit, page table pointer, and the TLB
    - Provide mechanisms to switch between user mode and supervisor mode and back

If a processor had to access a page table resident in memory to translate every access, virtual memory would be too expensive, as caches would be pointless. Instead, a TLB acts as a cache for the translations from the page table. Addresses are then translated from virtual to physical using the translations in the TLB.

        - An example is a system call exception that transfers control from user mode to a dedicated location in supervisor code space, invoking the exception mechanism in the process
- We also want to prevent a process from reading the data of another process
    - If the OS keeps the page tables organized so that the independent virtual pages map to disjoint physical pages, one process will not be able to access another's data
        - The mechanisms provided by the hardware ensure that only the OS makes changes to the page table and not the user process
- A process can ask the OS to allow them to share a page with another process
- When the OS decides to do a process switch, it must ensure that the TLB is cleared and that the new page table is being pointed to

TLB Misses/Page Faults

- A TLB miss occurs when no entry in the TLB matches a virtual address. A TLB miss can indicate one of two possibilities:
    - The page is present in memory and we only need to create the missing TLB entry
    - The page is not present in memory, and we need to transfer control to the operating system to deal with a page fault
- When we have a TLB miss, we look for a page table entry to bring into the TLB
    - If the matching page table entry has a valid bit that is turned off, then the corresponding page is not in memory
        - We therefore have a page fault, rather than just a TLB miss
    - If it entry has a valid bit that is on, we can simply retrieve the desired entry
- We bring the page table entry from memory through software into the TLB and re-execute the instruction that caused the TLB miss
    - We will now get a TLB hit
        - If the page table entry indicates that the page is not in memory, this time we will get a page fault exception
- Handling a TLB miss or page fault requires the use of an exception mechanism to interrupt the active process and transfer the control over to the OS
    - Later we will rescue the execution of the interrupted process
- A page fault will be recognized sometime during the clock cycle to access memory
    - To restart the instruction, we must save the current PC of the instruction that caused the page fault
- Once the operating system knows the virtual address that caused the page fault, it must:
    - Look up te page table entry using the virtual address and find the location of the referenced page on disk

- Choose a physical page to replace; if the chosen page is dirty, it must be written out to disk before we can bring a new virtual page into the physical page
- Start a read to bring the referenced page from disk into the chosen physical page
    - Step 2 may take millions of clock cycles if the page is dirty, and step 3 will definitely do so
        - We usually switch to another process to execute in the processor until the disk access completes
        - When the read is complete, the OS will restore the state of the original process and execute the instruction that returns from the exception

Thrashing

- Thrashing occurs when a program would be continuously swapping pages between memory and disk
    - THe performance difference between disk and memory means that if a program routinely accesses more virtual memory than it has physical memory, it will run very slowly.
    - One solution is to run it on a computer with more memory/buy more memory for your computer
    - Another solution is to re-examine your algorithm and data structures if you can change the locality and thereby reduce the number of pages that your program set.