Memory Hierarchy

David May: April 25, 2013

The Memory Hierarchy

Most computers rely on a hierarchy of storage devices. The fastest and smallest are usually architectural registers explicitly identified by instructions. These hold data which is transferred to and from memory; however because it takes many processor cycles to access memory, it is normal to include one or more special high speed memories known as *caches* to hold data temporarily. The caches are not normally visible to the programmer and data transfers to and from them are managed automatically by the hardware.

Caches are almost always organised in *blocks* (also known as *lines*), with each block consisting of a small number of words (typically 4). Data is transferred between caches and memory in blocks. The reason for this is that

- there is little additional cost in transferring the extra words after the first, given that time is needed to gain access to the memory system and to transfer the (read or write) command and address.
- there is a high probability that the extra words will be needed anyway, given that many programs access in succession a series of locations at similar addresses. Obvious examples are access to instructions, access to items in a procedure stack frame (where parameters and variables are held) and access to items in a record. This is sometimes referred to as *spatial locality*.

If a particular location in memory is referenced, then its nearby locations are likely to be referenced, therefore prepare them for faster access

In addition to the storage locations in each line in the cache, there is a *valid* bit to indicate that the line contains a valid block, and also a *dirty* bit to indicate that the block has been written to by the processor since it was loaded from memory. The dirty bit is used to avoid un-neccessarily writing the block back to memory.

Most computers have a level 1 cache (L1 cache) which is integrated into the processor. In many computers, the level one cache is in fact two caches, one for the instructions (the i-cache) and one for the data (the d-cache).

L1 is sometimes broken down to two caches. One for instructions and one for data

This allows instructions and data to be fetched simultaneously. General purpose computers normally have a level 2 (L2) cache on the same chip as the processor and L1 cache(s). The L2 cache holds both instructions and data; this is known as a *unified* cache. Sometimes there is a large level 3 cache which is provided as separate memory chips.

Blocks are transferred automatically between the caches and between the caches and memory. Normally the blocks are transferred only between adjacent levels of the hierarchy (Register-L1-L2-L3-memory). The final component of the memory hierarchy is the disc (or some other large capacity storage device). In some systems the memory behaves like yet another cache with the data residing on the disc and being transferred automatically between disc and memory on demand. This is known as a *virtual* memory system. In a virtual memory system the blocks transferred between memory and disc are much larger than cache blocks and are often called *pages*.

A great deal of research has been done over the years on cache architectures and there are many different schemes. This reflects the fact that there is no best architecture; they all have strengths and weaknesses. It is important to understand that caches are essential in order to allow modern processors to run efficiently and work well on programs which re-use the same instructions and data over a short period of time (this is known as *temporal locality*). However, there are algorithms which do not have this property and these do not run well on modern processors; there is a lot of processor idling and/or under-used processor resources. An obvious example is processing large dynamically-allocated data structures such as lists and trees.

Temporal locality is when a processor, prepares data and instructions that are going to be reused over and over again in a short period of time.

Three common types of cache are *direct mapped* caches, *associative* caches and *set-associative* caches.

Direct Mapped Caches

In a direct mapped cache each line in memory is mapped to a unique line in the cache. As there are many more lines in the memory than there are in the cache, this means that many memory lines compete for the same cache line. If the memory has 2^m lines and the cache has 2^c lines, then 2^{m-c} lines are mapped to the same cache line. In order to identify which memory line is currently held in a cache line, each cache line has a stored *tag* of size $(m-c)$ bits. This tag is compared with the most significant bits of the address when the cache line is accessed using the least significant c bits. If there is a match, and if the valid bit is set, there is a cache *hit* and the access goes ahead. Otherwise there is a *miss*; if the valid bit indicates that there is a cached line and the dirty bit indicates that it has been updated it must be

written back to memory and replaced with the required line.

Notice that in the case of write miss the block must be fetched from memory before it is written with new data bacause the block contains data in addition to the word being written.

Direct mapped caches are simple but have an important drawback. There is a significant chance that there will be too much competition for a given cache line and that it will be continually replaced from memory, rendering the cache ineffective. There are several ways this can happen and the effects can be very unpredictable, possibly depending on the actual addresses occupied by data structures and programs. Examples include:

- Advancing through a data area (eg an array) in large steps. It may turn out that a program runs efficiently for all two-dimensional arrays of size $[p, q]$ except where $p = k \times 2^m$.
- Repeatedly accessing corresponding elements in two data structures. If it turns out that the start addresses of two arrays are separated by a multiple of the cache size (ie by $k \times 2^m$), every access will result in a collision. As the space for the arrays may have been allocated at arbitrary addresses at run-time, it follows that there may be wide variations in execution time.
- Accessing two sections of program within a loop. The effects may be very difficult to understand, with performance suddenly dropping as a result of the distance between procedure entrypoints being near a multiple of the cache size. This could result from a minor program modification.
- Accessing stack and program within a loop. This would be similar to the previous case with essential data being displaced by the instructions which operate on it and conversely.

An alternative to a direct mapped cache which avoids these problems is an associative cache.

Associative Caches

An associative cache allows any line in memory to be stored in any line in the cache. If the memory has 2^m lines each cache line has a tag of size m bits. The tags are compared with the address when the cache is accessed. If there is a match, there is a hit and the access goes ahead. Otherwise there is

Any memory line can be stored to any cache line. If the cache is full, then one line must be replaced:
1) Select a line randomly.
2) Select the least recently used line because this is the one least likely to be needed gain.

a miss and the required line must be brought into the cache. If the cache is full, one of the cached lines must be written back to memory (if it had been updated) and replaced with the required line.

The problem with associative caches is that it is difficult to select a line to replace when the cache is full. The best algorithm is to select the least recently used line as this is the one least likely to be needed again. However if there are a large number of lines in the cache it is complex (and potentially slow) to keep track of which is the least recently used line. For this reason, associative caches commonly use random replacement and a random number generator is used to select the line to be replaced.

Set-associative Caches

The set-associative cache achieves most of the benefits of an associative cache without the cost of large tags and complex replacement methods. It is similar to a direct mapped cache but provides a small set (typically 4) of cache lines for each line in memory. This means that the examples above which cause collisions in a direct mapped cache will not cause collisions in a set-associative cache; provided that there is an unfilled line in the set it will be used.

The cache is borken down to sets of lines Each line in memory is mapped to a set in the cache.

A potential conflict can be avoided if one of the lines in the set is not full.

If the memory has 2^m lines and the cache has 2^c lines and associativity 2^a, then 2^{a+m-c} lines are mapped to the same cache 2^a cache lines. In order to identify which memory line is currently held in a cache line, each cache line has a tag of size $(m-c)$ bits. The tags in each set are compared with the most significant bits of the address when the set is accessed using the least significant $(c-a)$ bits. If there is a match, there is a hit and the access goes ahead. Otherwise there is a miss and the required line must be brought into one of the lines in the set. If all of the lines are in use, one of the cached lines in the set must be written back to memory (if it had been updated) and replaced with the required line. In this case, as the number of lines in each set is small, it is practical to employ a least-recently used scheme to determine which line to replace.

Write-through vs Write Back caches

Write back caches write to memory only when it is needed.
e.g. cache is full

In the caches described above, data is only transferred from the cache to the memory when there is no space in the cache to store a required line. This means that a line can remain in the cache for a long time before it is transferred to memory. Such caches are known as write-back caches, in contrast to write-through caches which write data to memory as well as storing it in the cache. The advantages of write-through caches are that data becomes immediately visible to other devices sharing access to the memory; these may

include other processors or input-output devices. The disadvantage is that many more memory transfers are needed.

Protection, Memory Mapping and Virtual Memory

Most processors have facilities to support

- Protection, to prevent an incorrect program from affecting other programs including the operating system
- Memory mapping, to simplify the task of allocating memory to a collection of programs
- Virtual Memory, to allow execution of programs (and collections of programs) larger than the physical memory

In general purpose processors intended to support standard operating systems, it is common for all of these functions to be provided by a single Virtual Memory system

Protection

Protection mechanisms have three objectives

- Containment. It should be possible to prevent errors in programs from affecting other programs or input-output devices. This is important in most computers, including embedded control processors.
- Monitoring. It is usually important to detect (and report) that errors have occurred.
- Recovery. It is sometimes necessary to recover from errors, for example by removing a program from memory and re-setting input-output activity it has initiated.

The simplest form of protection involves dividing the address space in two and using one half for the trusted program(s) and the other half for the untrusted programs(s). The general principle is then:

- When the program counter (PC) is in the trusted region, any address can be written and jumped to.

- When the PC is in the untrusted region, only addresses in the untrusted region can be written and jumped to.

Of course, there must be some way of transferring control from the untrusted region to the trusted region, either voluntarily (this is often known as a *system call*) or when an error occurs in an untrusted program (this is often known as an *exception*). This special mechanism for transferring control involves jumping to a specific address (or one of a number of specific addresses) in the trusted region. These special addresses are often known as *entrypoints*. It is possible to ensure that the (trusted) programs at these entrypoints will perform correctly and recover from any errors in the untrusted program.

Exceptions normally include

- protection error, such as trying to address outside of the allocated memory, or trying to write to special control registers
- illegal instruction
- arithmetic errors, such as divide by zero and overflow.

The special transfer of control is often known as a *trap* although it is sometimes referred to as a *software interrupt* because its effect is similar to that of an interrupt.

The trusted region is normally used to execute the core functions of an operating system including creating, removing and scheduling tasks (sometimes known as processes) and providing communication between them. It also includes the interrupt service routines providing the interface to hardware devices. These, like the traps also involve transfers of control to specific entrypoints in the trusted region. In the simplest schemes, each trap or interrupt service routine is executed with traps and interrupts disabled on entry and re-enabled on completion (by a special return from trap/interrupt instruction); this makes it simple to ensure that the data structures providing communication between software and hardware are kept in a consistent state. The collection of task scheduling software, trap and interrupt service routines is often known as an operating system *kernel*.

sits between sftware and hardware

Relocation

It is convenient to provide a special register to hold the address which separates the trusted region from the untrusted region. This register must of

course only be writable when executing trusted programs. A further refinement is to cause the contents of this register to be added to all memory addresses when executing untrusted programs. The effect of this is that the address space of an untrusted (user) program starts from zero, no matter where it is actually placed in memory. The user program is *relocated* by the special register (the *base* register). By adding yet another special register (the *limit* register) it is possible to contain a user program within a region of memory, preventing it from accessing memory outside the allocated region (below the base or above the limit). This makes it possible to have several user programs in memory at the same time with them all protected from each other in addition to the kernel being protected from all of them.

base and limit registers are like an upper and lower limit for the allocated region of a user program.

Using a simple relocation and protection scheme makes it possible to implement simple task schedulers which

- successively load programs from a backing store such as a disc (or from a network) into regions of memory.
- maintain a scheduling list of tasks ready for execution
- execute each task having set the base and limit registers appropriately
- provide tasks with access to input and output devices (and ensure that they do not interfere with each others' input-output operations)

A more complex scheduler might add the ability to temporarily remove tasks from memory thereby enabling a larger collection of tasks to be executed at the same time. The relocation mechanism means that it does not matter that a task is reloaded at a different location in memory.

Segments and Pages

The scheme described above is a very simple example of a *multi-tasking* system using *segments*. A segment is a region of memory described by a base and limit (or a base and length). A *segment descriptor* may also contain other information to control the use of the segment (for example to specify whether it is readable, writable or executable).

segment and segment descriptor

Some architectures support execution of programs using a collection of segment descriptors held in special registers; these would normally provide access to a program segment, stack segment and other data segments. Obviously, the provision of several segment descriptor registers increases the

amount of information which must be saved and restored when switching tasks.

An alternative to using variable sized segments is to use fixed size *pages*. Pages normally contain 2^n bytes, where typically n ranges from 10 to 20. Some architectures support only one page size; others support a small range of different sizes. Pages have some advantages over segments:

- they only need a single word descriptor
- storage allocation is easy because they are all the same size (or even if more than one size is available it is possible to easily break large segments into a whole number of smaller ones)
- they can be arranged to have a simple relationship with the size of blocks held on disc

Paging schemes have been adopted in all recent architectures in preference to segmentation schemes.

Virtual Memory

Using a segmentation or paging system makes it possible to give a programmer the illusion that a computer has a much larger memory than it actually has. This is done by providing the user with a virtual address space and using the physical memory like a cache. Segments or pages are then transferred between physical memory and disc as needed, in much the same way as blocks are transferred between caches and memory in a cache system.

It has become common to combine all of the functions of protection, memory management and virtual memory in a single mechanism in which the addresses manipulated by the processor fall within a virtual address space much larger than the physical address space. In addition, each task can be provided with its own virtual address space.

Virtual and Physical addresses

In a virtual memory system, the *virtual addresses* manipulated by the processor must be translated into the *physical addresses* used to access memory. If the memory system is divided into pages of size 2^p, the virtual address space is of size 2^v and the physical memory is of size 2^m, then there will be 2^{v-p} pages in the virtual address space and 2^{m-p} pages in the physical memory. The virtual addresses can be translated to physical memory addresses using

Take the (v−p) significant bits of the virtual address and find the number of the corresponding physical page from the table. Then concatenate with the p least significant bits of the virtual address. Those p bits represent the offset within the selected page.

a *page table* with 2^{v-p} entries each of size $(m-p)$ bits. The most significant $(v-p)$ bits of the virtual address are used to select an entry in the page table producing the most significant $m-p$ bits of the physical address (these bits give the number of the physical page); the least sigificant p bits of the physical address are the same as the least significant p bits of the virtual address (these bits give the offset within the selected page).

In addition to the $(m-p)$ bits in each page table entry, there will be information to indicate whether or not the virtual page is currently held in the physical memory. If not, instead of holding the number of the physical page, the entry might hold information to allow the virtual page to be located on a disc or other mass storage device if this can not easily be deduced from the virtual address.

If the virtual address space is large and the pages are small (ie v is much larger than p), the page table which is of size 2^{v-p} will be very large. In this case it is possible to use a hash table which which contains information about the physical pages currently in use; for each physical page in use the entry will contain the corresponding virtual and physical page addresses. This is known as an *inverted* page table.

Accessing data in a virtual memory system

When the processor attempts to read or write using a virtual address, the page table is used to determine if the virtual page containing the virtual address is currently held in physical memory. If so, the entry in the page table is used to translate the virtual address to a physical address and the memory access goes ahead. Otherwise, the information in the page table must be used to transfer the virtual page into physical memory. This will take a relatively long time even if it is carried out automatically by hardware; it is limited by the speed of access to the disc (and the disc data transfer rate).

In most current systems, a trap is generated when an attempt is made to access a virtual page which is not in physical memory; the trap is normally referred to as a *page fault*. The operating system will then execute a trap handler to cause the virtual page to be transferred from disc. As this will take a long time, the operating system may de-schedule the task and activate another one.

Page Replacement

Obviously, as in the case of a cache, it is possible that there is no space left

in physical memory when a page fault occurs and a page must be transferred from disc. In this case a page must be transferred from memory to disc to make space. In this respect the virtual memory system behaves like an associative cache; virtual pages can be placed anywhere in physical memory. However, in the case of virtual memory the cost of a page fault is so large that it is worth trying to make the best possible selection of the page to be replaced. Instead of a random replacement scheme, it is normal to implement a *least-recently-used* scheme; this is also fairly easy as it is carried out using software in the operating system.

Least recently used

The least-recently-used page is determined using a list of the pages in physical memory. When a page is accessed, it is moved to the head of the list, so that un-accessed pages gradually move to the tail. The list is normally *doubly-linked* (linked in both directions) so that it is possible to efficiently remove a list entry from the middle of the list and insert it at the head of the list. In practice it is not possible to carry out this operation every time an access takes place, so instead each entry in the page table has a flag which is set when the page is accessed. These flags are periodically checked by the operating system (possibly using a timer interrupt); the pages which have been accessed are moved to the list head and the flags are cleared.

Translation Lookaside buffer

An important disadvantage of a page table held in memory is that every memory access takes two accesses in sucession. The first access is to the page table to translate the address; the second is to access the data in the physical page. As memory accesses account for around 30% of instructions executed, this is a significant performance penalty.

In order to speed up translation, it has become common to use a special kind of cache known as a *translation lookaside buffer* (TLB). The idea is based on the observation that if the addresses in programs have locality, so do their translations. In practice, a small number of entries (typically 32) is sufficient.

TLB is associative cache.

The TLB is a small associative cache loaded by the operating system. Like a cache each entry has a valid bit and each valid entry contains a translation consisting of a tag of size $(v - p)$ bits and a corresponding physical address of size $(m - p)$ bits.

On every memory access, a TLB lookup is made to derive the most significant $(m - p)$ bits of the physical address. Notice that provided that the page size

The TLB lookup will return a physical address of the memory. THe result can be used immediately to check if the address is currently in the L1 cache

is bigger than the size of the L1 cache, the TLB lookup can proceed alongside the access to the cache as the TLB result will only be needed for checking the L1 cache tag(s).

Of course, it is possible that the TLB lookup fails to find a translation, in which case a TLB miss occurs, causing a trap and entry to an operating system handler. This may cause a new entry to be loaded into the TLB. Like the page table, the TLB entries may also contain flags to record when they have been used to support a least recently used replacement algorithm. However another method is to simply disable the translations using the valid bits from time to time; any access will then generate a trap which will move the corresponding page to the list head and re-enable the translation.

Protection

A small extension to the virtual memory system described above allows protection of tasks from each other, and protection of the operating system from the tasks. By adding permission information to the page table entries or to the TLB entries, it is possible to restrict the accesses which can be made by the currently executing task. Typically, each entry would have three bits, one enabling the corresponding page to be read, one enabling it to be written and one enabling it to be executed.

The operating system can therefore determine which areas of memory may be used (and for what purpose) by a task in that it will only load permitted translations into the TLB when the task is started (or restarted).