Memory hierarchy

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Javier García Blas

Computer Architecture
ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid

1. Module structure

This module is structured in three lessons:

1. **Basic cache memory**. Reviews the basic cache memory concepts and introduces a set of basic cache memory optimizations.
2. **Advanced memory cache optimizations**. Introduces a number of more advanced optimizations associated to cache memory.
3. **Virtualization and memory hierarchy**. Reviews concepts related to hardware support for virtual memory and introduces the hardware support mechanisms for virtual machine monitors.

2. Basic cache memory

This lesson has the following general structure:

1. Introduction.
2. Policies and strategies..
3. Basic optimizations.
   a) Reducing miss rate.
   b) Reducing miss penalty.
   c) Reducing hit time.

2.1. Introduction

If we consider performance as the inverse of latency and we observe the period from 1980 to 2010, processor performances has increased by several orders of magnitude compared to memory system performance. This wall has increased even more with the emergence of muti-core processors as the

place more pressure on the memory system. One of the reactions to this problem has been the use of cache memories.

Cache memories are based on the principle of locality. That principle has two versions: the spatial locality principle and the temporal locality principle. In both cases the use of a temporal memory which is smaller but faster helps to significantly improve performance of the memory system.

A cache memory contains a set of memory blocks or lines. When a datum is accessed, if it is in cache memory the access is said to be a hit. If the datum is not in cache memory it is said to be a miss and it is necessary to bring the corresponding block from the next level in the memory hierarchy.

Consequently, the average memory access time depends on access time to cache memory t_A, the hit rate h and the miss penalty time t_F.

This makes that the number of cycles needed to execute each instruction is increased by the number of stalled cycles due to waits for the memory system. This depends on the number of memory accesses performed, the hit rate, and the miss penalty.

2.2. Policies and strategies

In general, the design of an element in the memory hierarchy depends on the answers to four basic questions. This can be applied to any level in cache memory as well as to the case of virtual memory.

Where is a block placed in the upper level? In cache memory this question leads to the **block placement policy**. The *set associative mapping* policy can be seen as a trade-off between *direct mapping* policy and the *fully associative mapping policy*.

How is a block located in the upper level? In cache memories this questions leads to **block identification** policy. The memory address is split in three fields: tag, index and offset.

Which block must be replaced upon miss? In cache memory this question leads to the **block replacement** policy. This policy is relevant when associative mapping or set associative mapping policies are used for placement. Most common block replacement policies are FIFO and LRU.

What happens on write? In cache memory this question leads to **write strategy**. Here, there are two alternatives: write through and write back. In write through, all writes are sent to memory, while in write back, only when a block is replaced it is written to memory.

2.3. Basic optimizations

2.3.1. Reducing the miss rate

An **increase in block size** leads to a lower miss rate, and consequently, improves an efficient use of spatial locality. However, it also leads to a higher miss penalty.

An **increase in cache size** also decreases miss rate as more blocks may be stored in cache memory. However, it may increase hit time.

An **increase in associativity** also may decrease miss rate, as it decreases the number of conflicts when more ways may be used in a set.

2.3.2. Reducing miss penalty

Using **multilevel caches** allows to keep a balance between cache size increase and cache access time reduction.

Prioritizing read misses over writes avoids that a read miss has to wait until a write finishes.

2.3.3. Reducing hit time

Avoiding address translation during indexing allows to speedup the access time in case of a hit. To do so, virtual addresses are used to index in the cache.

3. Advanced cache memory optimizations

This lesson has the following general structure:

1. Introduction.
2. Advanced optimizations.

3.1. Introduction

Advanced cache memory optimizations try either to decrease metrics like hit time, miss rate or miss penalty, or to increase cache memory bandwidth.

3.2. Advanced optimizations

Using **small caches** allows to decrease hit time. Lookup process includes selecting a line, using the index field, reading line's tag and comparing to the tag from the address. Using a smaller cache, the needed lookup hardware is simpler and it also allows the cache memory to be integrated in the processor chip.

way prediction may be performed by storing additional bits to predict the way that will be selected in the next access. In this way, the block access may be prefetched and the comparison is performed against a single tag. If after that, a miss is detected comparison with the rest of tags is performed.

To increase cache bandwidth the cache access may be **segmented** in several cycles. Consequently, a new access may be initiated every cycle achieving a higher bandwidth, although latency is also increased.

Cache misses cause stalls until the required block is obtained. To hide stalls it is usual to execute instructions out of order. However, this may cause problems if a miss is being resolved. A **non-blocking cache** may allow hits during a miss. A more complete solution is to allow overlapped misses (miss during a miss), although at the cost of a higher complexity.

Another technique to improve bandwidth is to allow simultaneous accesses to different cache memory locations. This may be achieved by using **multi-bank caches**, splitting cache memory in multiple banks.

Processors usually require a single word from the cache block. Consequently, an improvement may be not tow wait until the whole memory block is obtained to deliver the datum. Alternatives to achieve this are to deliver **critical word first** or to perform an **early restart**.

To reduce write miss penalty a **write buffer** may be used. In this case, as soons as the datum has been written to the buffer, the write is considered to have been performed.

There is a variety of cache optimizations that can be performed by the **compiler**. To reduce instruction misses the compiler may perform procedure reordering, it may align code blocks to cache line limit, or to perform branch linearization. To reduce misses on data accesses it may perform array merges, loop exchanges, loop merges or blocked accesses.

Finally, using **hardware prefetching** for instructions or data allows to bring to cache instructions or data before the access to them has been generated.

4. Virtualization and memory hierarchy

This lesson has the following general structure:

1. Virtual memory.
2. Policies.
3. Page table.
4. Virtual memory.
5. VMM: Virtual memory monitors.
6. Virtualization hardware support.
7. Virtualization technologies.

4.1. Virtual memory

Virtual memory emerges as a mechanism to solve problems arising from the limitations on the addess space, besides of offering protection, translation, and sharing mechanisms. In this way, virtual memory allows to overcome the physical memory size limitations running programs in a normalized virtual address space.

In contrast to the case of cache memory, in this case cooperation between hardware and operatin system is required. Thus, hardware performs the address trasnlation, and operating system manages those addresses.

4.2. Policies

In the case of virtual memory, its design may be defined answering the very same questions posed for the case of cache memory in the previous lesson. In this case, the block or unit of transfer is a page. Processes are organized int pages and memory is organized into page frames.

The **page placement policy** is always fully associative. Each page may be placed in any page frame in main memory. The goal is to minimize, as much as possible, miss rate, as miss penalty is very high due to intrinsic characteristics for sencondary storage devices.

For **page identification** a page table is kept on per process basis. This table keeps the mapping betwing page identifiers and page frame identifiers. In principle, an additional memory access would be required to perform a lookup in that table. In practice, to decrease translation time a translation buffer is used: the *Translation Lookaside Buffer* or TLB, which keeps most frequently used translations.

The **page replacement policy** is usually defined by the operating system and the most used one is LRU (*Least-recently used*). However, the architecture needs to offer support to the operating system.

Finally, **write strategy** is always *write-back* as the cost of a write on disk is very high.

4.3. Page table

It is important to keep in mind that the page table is really a data structure from the operating system. However, it is special in that it requires hardware support.

On one hand, the memory management unit has a register with the address where the current page table starts (the PTBR or *page table base register*). Each time a context switch happens, the operating system has the responsibility to update the register with the start address of the page table from the corresponding process.

On the other hand, a TLB must also be offered to keep the most frequent translations. Otherwise, memory access would have an unacceptable cost.

In any case, when processes address spaces are very large, page tables need many entries and a great amount of pages might be unused. To solve this problem, most typical solutions include inverted page tables o multi-level page tables.

4.4. Virtual machines

The idea of virtual machine emerged in the 60's when it was first used in *mainframe* environments. However, the idea was otherwise practically ignored in singe-user environments until late 90's.

There are several reasons behind the popularity recovery of virtual machines. On one hand, the increasing importance of isolation, security, and reliability, as well as the need to share a single computer by multiple users or multiple subsystems. On the other hand, the high increase in processor performance has made that the use of virtual machine monitors is now more acceptable.

A virtual machine offers the illusion to users to have a complete computer at their sole disposal, including their own copy of the operating system. A computer may run multiple virtual machines, each one with its operating system, while all operating system share the use of the same hardware. In this context, we usually call *host* to the underlying hardware platform and *guest* to each of the executed virtual machines.

4.5. VMM: Virtual machine monitors

A hypervisor or virtual machine monitor is the software that supports virtual machines. In this way, different virtual machinies are executed on top of the hypervisor, which is the only one with direct access to hardware. The VMM determines the mapping between virtual resources and shared physical resources.

Depending on the physical resource time sharing (e.g. CPU), partitioning (e.g. disk) or software emulation (e.g. virtual network cards) may be used.

Overheads derived from uisng a virtual machine depends on the kind of workload used. For workloads that are highly compute bound, overhead is usually negligible. In contrast, for programs that are I/O intensive the virtualization overhead may be very high.

Besides protection, virtual machines offer other uses as software management allowing a combined deployment of operating system and applications or hardware management allowing the execution of separated software stacks on the same hardware.

4.6. Hardware support for virtualization

If the virtual machine is considered when designing the processor instruction set, it is relatively simple to decrease the number of instructions which need to be executed by the VMM. In other case, the VMM must intercept problematic instructions and introduce virtual resources which need to be the only ones that are visible vy virtual machines.

One case of resources virtualization is memory. The virtual monitor introduces the concept of real memory which is an intermediate layer between virtual memory and physical memory. In principle, this would lead to translation to be performed in two stages. To avoid it, VMM make use of a shadow page table. Another more complex case is virtualizing input/output. In this case, the most general part from the driver is left in the guest keeping the specific part in the VMM.

4.7. Virtualization technologies

Impure virtualization is one solution that has been used for those architectures that are not directly virtualizable. There are two alternatives: paravirtualization and binary translation. With the paravirtualization, guest operating system code is ported to a modified ISA including VMM invocations. With binary translation, non virtualizable instructions are replaced by emulation code or by calls to the VMM.

Another alternative that has been used by processors manufactures like Intel or AMD is the instruction set extension to support virtualization. Intel provides Intel-VT (*Intel Virtualization Technology*) and AMD provides AMD SVM (*Secure Virtual Machine*). Both technologies are based in offering a differentiated execution mode for the VMM.