Some background on computer architecture

Amitava Datta

School of Computer Science and Software Engineering
University of Western Australia

July 29, 2014
What is a computer?

![Diagram](image.png)

Figure 1: A schematic model of a computer.
What is a computer?

- Conceptually, a computer has three parts, the Central Processing Unit (CPU) or processor, the memory (RAM) and some input/output (I/O) devices.
- The connection between the CPU and the RAM is called the system bus.
- For executing a program, it has to be first stored in the Memory or RAM and then executed by the CPU.
- This is the famous ’von Neumann’ architecture that is used in all forms of computers that you see around you.
- It was first proposed by John von Neumann.
A CPU looks like this

![Diagram of CPU registers](image-url)

- R1
- R2
- R16
- PC
- SP
- PSW

Figure 2: A CPU has several registers and an arithmetic-logic unit (ALU) not shown in this figure.
What kind of programs computers can execute?

- Though you write programs in many languages, e.g., Java, C, Python, Fortran, a particular computer can execute programs that is written in just one particular language.
- It is called the *machine language* for that particular computer.
- A special program called a *compiler* translates a program written in a *higher level* language like C or Fortran into a machine language program for a particular computer.
- This machine language program is then stored in the RAM and executed by the CPU, instruction by instruction.
- An example machine language program \(A = A + B\):
  
  MOV A, R1
  MOV B, R2
  ADD R1,R2
  MOV R1, A
What is a process?

- A program is the code, or the machine code after it has been translated into machine language.
- A process is a *program in execution*.
- A process requires resources for its execution by the machine (we will sometime call a computer as a machine).
- The operating system allocates memory, allows the process to make *system calls* for performing input/output operations.
- A process can access memory that is allocated to it by the operating system.
- The operating system usually terminates a process if the process tries to access memory that is not allocated to it.
How a process is executed

➤ A processor usually executes processes in a round-robin fashion.
➤ Usually many processes wait for execution in a queue.
➤ The operating system gives a fixed time for the process at the head of the queue. If the process does not complete execution, it is suspended and the next process is given time and so on.
➤ The suspended process goes to the back of the queue and get scheduled to execute later.
➤ The operating system stores the execution status of a process for resuming it later.
Process states

- blocked
- running
- ready to run
- new process
- terminated
What is a thread?

- A process can be considered as an independent execution environment in a computer system.
- There are usually many processes in a system at any time, each with its own memory space. Each executes a sequence of instructions (the machine language program).
- Threads are also independent execution environments, but with a shared memory space (or address space).
- Each thread executes its own sequence of instructions (or machine language code), but they can access a set of common memory locations.
- Usually a process creates multiple threads and these threads share the address space of the parent process.
Threads

Process address space

- Global variables (shared)
- PC
- Stack
- Shared code

Thread 1

Thread 2

Thread n
The memory hierarchy
The memory hierarchy

- Most computers have a very fast memory called the *cache*.
- The access time for the cache memory is orders of magnitude faster.
- The fastest access time is for registers, 1-2 nano seconds, then cache, 10 nano seconds, then the RAM, 100s of nano seconds to microseconds.
- When the CPU needs an instruction or data from the RAM, the cache is looked up first.
- If the item is present in cache, the memory access is very fast.
- How fast a process will run depends on how much of the memory access is satisfied from the cache.
The memory hierarchy

- Why not build computers entirely with fast memories like the cache?
- The cost will be high, consumer hardware will not be competitively priced.
- The cost is due to the cost in manufacturing hardware, caches are usually "on-chip" with the processor.
Multi-level cache

- Multi-core processors usually have three levels of cache, L1, L2 and L3.
- The L1 cache is private for each core, each pair of cores share an L2 cache and all cores share the L3 cache.
- L1 cache is the smallest, the L2 cache is larger than the L1 cache and the L3 cache is larger than the L2 cache.
- The memory access pattern is, first look-up the L1 cache, then the L2 cache, then the L3 cache and finally the RAM.
Multi-core and multi-threading

- The purpose of multi-core, multi-threaded programming is to make processes run faster.
- We partition the work so that multiple threads can share the work. (more on this later, the entire unit is about this).
- However, it is no good if multiple threads all run on the same CPU. They will just get round-robin access to the CPU. The performance will not improve.
- The performance can be improved if multiple threads run on multiple cores, simultaneously in parallel.
Cache hit and cache miss

- Let us consider just a single level of cache and the RAM.
- When the instruction or data is not found in the cache, the memory access request from the CPU goes to the RAM.
- At this point, the cache is filled with a block of instructions or data from the RAM (This is called the *cache line*). This is a contiguous block of instructions or data adjacent to the request that the CPU made.
- The current block of instruction or data in the cache is over-written (and lost).
- It is a *cache-hit* if the required information is found in the cache, and a *cache-miss* if it is not found in the cache.
The expectation is that the process will need instructions and data (in the short and medium terms) adjacent to the instruction or data that caused the cache miss.

This is based on the principle of referential locality.

Process execution is mostly localized in code. In other words, machine instructions are mostly executed one after another.

The exceptions are conditional statements like if and function or procedure calls, where jumps in process code occur.

But these jumps are infrequent.

Similarly, data usage by a process is also mostly sequential.
Multi-threading performance

- The cache access pattern is similar when there is a hierarchy of cache.
- First the L1 cache is looked up, then the L2 cache, then the L3 cache and finally the RAM.
- Whichever cache access results in a cache-hit, will also result in filling up the lower caches.
- For example, if there is a cache-hit in the L3 cache, the L2 and L1 caches will filled up with blocks from the L3 cache, depending on their capacities.
- The performance of multi-threaded programs depend almost completely on how they utilize the multi-level caches.
- In fact, quite often multi-threaded programs run slower than single-threaded (or sequential) programs if the programs are not carefully written or data structures are not carefully designed.
An intuitive example

- Consider a simple example of adding the numbers stored in an array (I hope all of you have seen arrays in whatever language you know).
- Suppose the array is very large: `int a[100000]`
- A natural way of adding the elements in this array is to use two threads, $t_1$ and $t_2$. $t_1$ will add the elements $a[1..50000]$ and $t_2$ the elements $a[50001..100000]$
- the program should run twice as fast as the work is halved between the two threads.
- So we write a program where the two threads run on two cores that share an L1 cache.
An intuitive example

- Threads $t_1$ and $t_2$ execute different loops:
  - $t_1$ : for($i=0;i<50000;i++)$
    \quad \text{localSum1 = localSum1+a[i];}
  - $t_2$ : for($i=50000;i<100000;i++)$
    \quad \text{localSum2=localSum2+a[i];}
- Finally, one of $t_1$ or $t_2$ computes the sum:
  \text{sum=localSum1+localSum2;}
- Though it seems a beautiful partition of work between the two threads, this will result in very poor performance. The program will run much slower compared to the sequential program (just one thread).
An intuitive example

- Consider an L1 cache that these two threads are sharing. Assume that the L1 cache can hold 100 elements from the array $a[]$.

- In the sequential program (a single thread), the element $a[0]$ will be accessed in the very first iteration of the loop.

- However, this will be a cache-miss (as the cache is empty in the beginning) and hence elements $a[0]..a[99]$ will be brought to the L1 cache.

- The next 99 elements can now be accessed from the L1 cache, i.e., there will be 99 consecutive cache-hits after this.

- This will result in very fast execution of the sequential algorithm.
An intuitive example

- Consider now the two threads $t_1$ and $t_2$.
- When $t_1$ executes the first iteration of its loop, elements $a[0]..a[100]$ will be brought to the L1 cache.
- When $t_2$ executes the first iteration of its loop, elements $a[50000]..a[50099]$ will brought to the cache.
- This *ping-pong* effect will continue and the cache will be filled again and again by the elements each thread requires.
- In effect, almost every iteration of each thread’s loop will result in a cache-miss and the overall execution time will be much longer compared to the sequential algorithm.