High Performance and Parallel Computing

Semester 2, 2024

Associate Professor Wayne Kelly

Tarang Janawalkar

Contents

1 Processor Design

High performance programs must be designed to take advantage of the hardware they run on. This section discusses how Central Processing Unit (CPU) design influences how programs are executed, and why parallelisation is necessary for high performance computing.

1.1 Von Neumann Architecture

Most general purpose computers are based on the von Neumann architecture where instructions and data are kept in the same memory. In this model, the CPU performs a fetch-decode-execute cycle to execute instructions, as described below:

1. **Fetch**: The CPU fetches the next instruction from memory into the instruction register and increments the program counter.
2. **Decode**: The CPU decodes the instruction to determine what operation to perform. This may involve one of the following:
    - Load data from memory into a register.
    - Store data from a register into memory.
    - Perform an arithmetic or logical operation on data in registers.
    - Transfer control to another part of the program by changing the program counter.
3. **Execute**: The CPU performs the operation specified by the instruction. This may involve the Arithmetic Logic Unit (ALU) for arithmetic and logical operations, or the bus/memory module for memory operations.

1.2 CPU Clock

Each instruction in the fetch-decode-execute cycle takes a certain amount of time to complete. This time is determined by the speed of the CPU which is controlled by a clock that generates a series of pulses at a fixed frequency. These pulses are called **clock cycles** and the time taken for one cycle to complete is called the **clock period** of the CPU. This period must be long enough to:

- allow electrical signals to propagate through the CPU, and
- allow transistors to reach a steady state when switching between on and off.

An increase in clock frequency therefore results in a faster CPU, but due to the physical limitations of the speed of light and the size of transistors inside the CPU, single-core CPU performance has plateaued over the past decade.

Note in some cases, the CPU may be “over-clocked” to run at a higher frequency than it was designed for, but this can result in incorrect operation if the physical components are unable to switch states fast enough, and lead to overheating.

1.3 Von Neumann Bottleneck

Main memory resides outside the CPU, and is connected to the CPU via a shared bus. As the memory controller has a much lower clock frequency than the CPU, the CPU often stalls while a data operation is being performed between the CPU and memory. This is known as the **von Neumann bottleneck** and is a major limitation of the von Neumann architecture. To minimise performance loss, memory accesses should have:

- maximal throughout (i.e., transfer as much data as possible in one operation), and
- minimal latency (i.e., minimise the time taken to start and end an operation).

1.3.1 Cache Memory

A common way CPUs achieve this is by using **cache memory** to store frequently accessed data and instructions in small specialised blocks of memory that are close to each core. These caches provide higher throughput and lower latency than main memory, but are typically more expensive and therefore have limited capacity. A CPU often has multiple levels of cache, starting with the Level 1 (L1) cache which is the smallest and fastest, increasing in size and latency as the level increases. On modern CPUs, the L1 cache is typically split into two parts: one for *instructions* and one for *data*. Multi-core CPUs may also have unified L2 and L3 caches that are shared between cores.

Cache Operation The cache works by loading data in blocks of a fixed size called **cache lines**. When the CPU requests less than a cache line of data, it loads surrounding data into the cache to take advantage of locality of reference. This is done in the hope that the CPU will eventually request the surrounding data, which will already be in the cache and is therefore faster to access. When memory is requested, the CPU first checks if it exists in the cache, requesting it from main memory only if it is not found.

Cache Replacement When the cache is full, the CPU must decide what data to replace in the cache, through a *cache replacement policy.* Typically this is done using the Least Recently Used (LRU) policy, where the data that has not been accessed for the longest time is replaced. The *cache placement policy* determines where new data is placed, and can be:

- **Fully associative**, where any cache line can be replaced.
- **N-way set associative**, where the cache is divided into N sets, and the replacement policy is applied within each set.
- **Direct mapped**, where new data must be mapped to a specific cache line.

1.4 Instruction-Level Parallelism

Scalar processors are designed to process and execute instructions one at a time. High-performance processors take advantage of disjoint operations that can be performed in parallel through **instruction level parallelism** (ILP). This can be achieved in several ways:

- **Superscalar Execution**: Dispatching multiple scalar operations to different functional units (ALU, integer multiplication, FPU, store/load, etc.) during a single clock cycle. This is facilitated by hardware that determines dependencies between instructions and schedules them for parallel execution.

- **Out-of-Order Execution**: Executing other instructions while waiting for data required by the current instruction. As this can result in instructions being executed out of order, the processor must ensure that the final result is correct.
- **Instruction Pipelining**: Dividing instruction processing into multiple stages to keep the processor busy. For example the fetch-decode-execute cycle can be divided into three stages, with each stage being executed in parallel, allowing the processor to begin executing the fetch stage of the next instruction while the decode stage of the current instruction is being executed.

To keep the processor busy, processors may also employ **branch prediction** to predict the outcome of branch instructions and **speculative execution** to execute instructions that may not be required, but are likely to be executed.

Each of these techniques result more complex processor cores that occupy more space on the CPU chip, consume more power, and have longer cycle times.

1.5 Processor Architectures

There are two main types of processor architectures:

- **Complex Instruction Set Computing (CISC)**: Designed to execute complex instructions that can perform multiple operations in a single instruction. These instructions may be decoded into micro-operations that are executed by the CPU. These processors have large instruction sets and typically have slower clock speeds.
- **Reduced Instruction Set Computing (RISC)**: Designed to execute simple instructions that perform a single operation. These processors have smaller instruction sets and typically have faster clock speeds.

1.6 Multi-Core Processors

Moore's Law

Moore's Law states that the number of transistors on a CPU chip doubles approximately every two years. This has led to:

- Smaller circuits
- Faster clock speeds
- Additional complex hardware level optimisations

While this has held true for the past few decades, it is now increasingly difficult to reduce the size of transistors due to physical limitations, such as the size of a wire compared to the width of an atom.

Rather than allocating this additional space to a single core, CPUs now contain multiple cores that can execute an independent instruction stream[1]. Each core is itself a complete processor, with its own functional units, program counter, instruction register, general-purpose registers, and cache.

[1]Note that each core may execute the same stream if each stream operates on different data, as in Graphical Processing Units.

These cores can communicate with each other via shared memory, but this leads to the same bottleneck as before. To overcome this, cores may have their own private caches, and communicate with each other via a shared L2 or L3 cache.

1.6.1 Cache Coherence

When multiple cores share the same memory, it is important to ensure that each core has the most up-to-date copy of the data. Cache coherence is the consistency of data stored in multiple private caches that reference shared memory. Cache mechanisms are used to ensure that modifications to data in private caches are propagated to other cores to ensure other cores do not read stale data. This is typically done using *snooping* or *directory-based* mechanisms:

- **Snooping**: Each cache monitors the bus for access to memory locations that have been cached. If a write is detected, the cache either updates its data or invalidates it.
- **Directory-based**: A centralised directory keeps track of data shared between cores. Each processor must request permission to load an entry from a memory location. When an entry is updated, the directory either updates or invalidates other caches with that entry.

Caches also have inclusion/exclusion policies that determine whether data in a lower-level cache may be present in a higher-level cache.

1.7 Software Parallelism

Parallelism can be achieved at the software level through processes and threads. A **process** is an instance of a program being executed by the operating system that has its own memory space and threads of execution. Upon initialisation, a process creates a **thread** that executes its main function. This main thread can then spawn additional threads that execute independent streams of instructions.

A **thread** is a lightweight process that has its own program counter and local memory called the **stack**. The lifetime of the data in the stack is determined by the function that created it.

Every process also has a shared memory space called the **heap**, where data with indeterminate lifetimes is stored (i.e., dynamically allocated objects). This space can be accessed by threads within the same process, and by threads in other processes through inter-process communication APIs.

1.7.1 Mapping Threads to Cores

Operating systems achieve concurrency in threads through time slicing, where each thread is allocated a core for a fixed amount of time. When a time slice ends, the operating system performs a context switch to another thread by saving the current thread's state and loading the next thread's state. This allows multiple threads to make progress on a single core, and allows threads to be executed in parallel on systems with multiple cores. Note that for a CPU with N cores, a process with N threads may not always be granted all N cores, as the operating system may allocate some cores to other processes.

1.8 Simultaneous Multi-Threading

To reduce the overhead incurred during a context switch, some processors allow more than one thread to be executed on a single core at the same time by interleaving instructions from multiple threads. This is known as **Simultaneous Multi-Threading (SMT)** or **Hyper-Threading**. In these processors, each core has multiple sets of registers and program counters (typically two) for each thread, with a shared set of functional units, that allow multiple threads to be executed in parallel.