CS3014 Concurrent Systems I

Harshvardhan Pandit
Ph.D Researcher
ADAPT Centre, Trinity College Dublin

CS3014

5 ECTS

30 lecture hours (1 lab ~ 2hrs)

100 marks (80mks paper, 20mks assignment)

> Basics of practical parallel programming with the goal of achieving real speedups on multi-core systems.

> Understanding of computing architecture

Why write a parallel program?

- To solve your problem more quickly
  - Divide work across parallel units
  - Main focus of this course
- But normally we say that you shouldn't worry about running speed
  - At least until you discover it is a problem
  - But certain types of applications benefit a lot from speed optimizations

Why write a parallel program?

- Parallelizing a program is a speed optimization like any other
- Some parallelization is automatic
  - By compiler or processor itself
  - This is easy and good
  - But limitations on what can be done automatically

Why write a parallel program?

- Parallelism is also tied to energy
  - A parallel computer with several processors is often more energy efficient than a similar machine with one super fast processor

Long long ago in a distant time of single-core computer architecture

Notes borrowed from

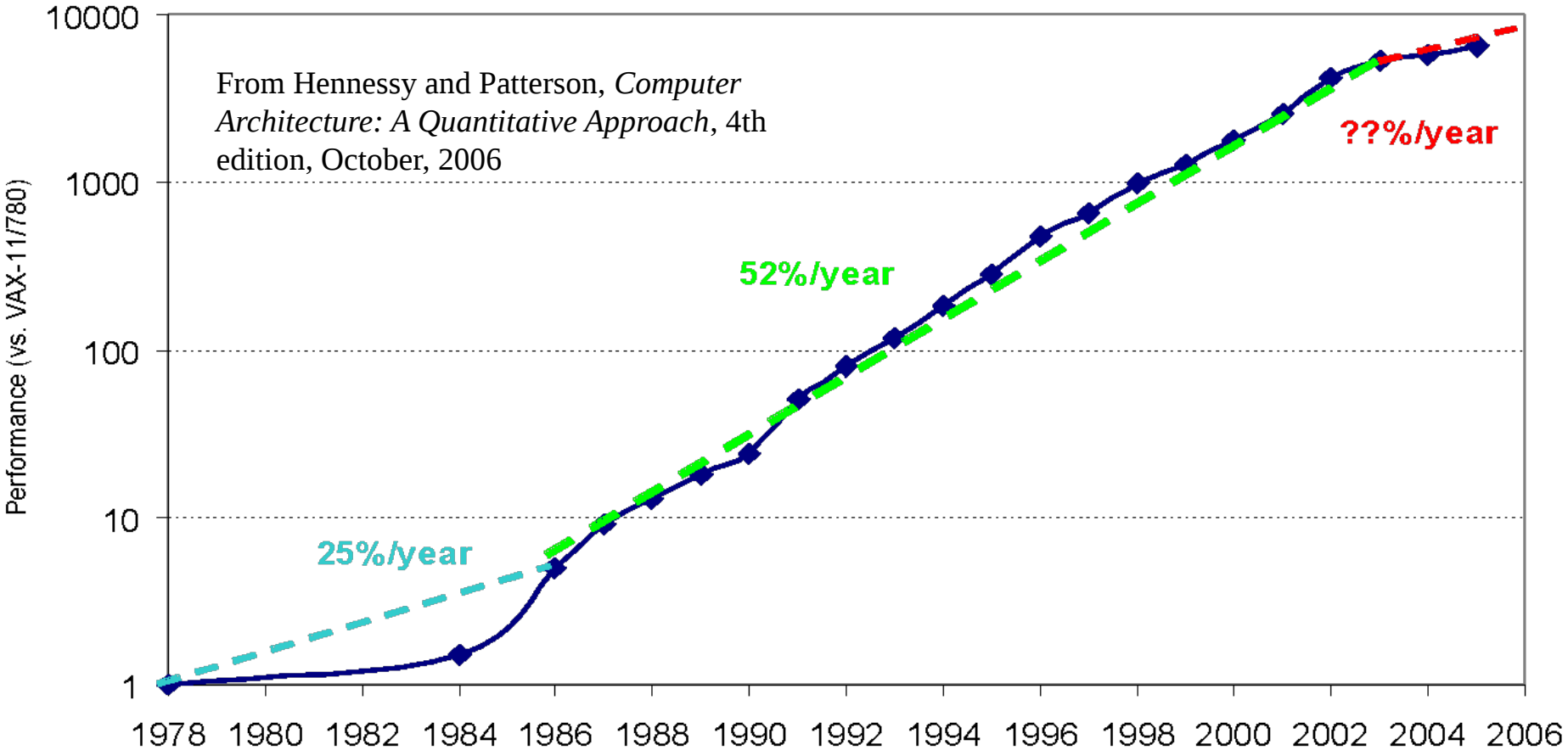
David Patterson
Electrical Engineering and Computer Sciences
University of California, Berkeley

Crossroads 2005: Conventional Wisdom in Comp. Arch

- Old Conventional Wisdom (CW): Power is free, Transistors expensive
- New Conventional Wisdom: "Power wall" Power expensive, transistors free (Can put more on chip than can afford to turn on)
- Old CW: Sufficiently increasing Instruction Level Parallelism via compilers, innovation (Out-of-order, speculation, VLIW, …)
- New CW: "ILP wall" law of diminishing returns on more HW for ILP
- Old CW: Multiplies are slow, Memory access is fast
- New CW: "Memory wall" Memory slow, multiplies fast
  (200 clock cycles to DRAM memory, 4 clocks for multiply)
- Old CW: Uniprocessor performance 2X / 1.5 yrs
- New CW: Power Wall + ILP Wall + Memory Wall = Brick Wall
  - Uniprocessor performance now 2X / 5(?) yrs

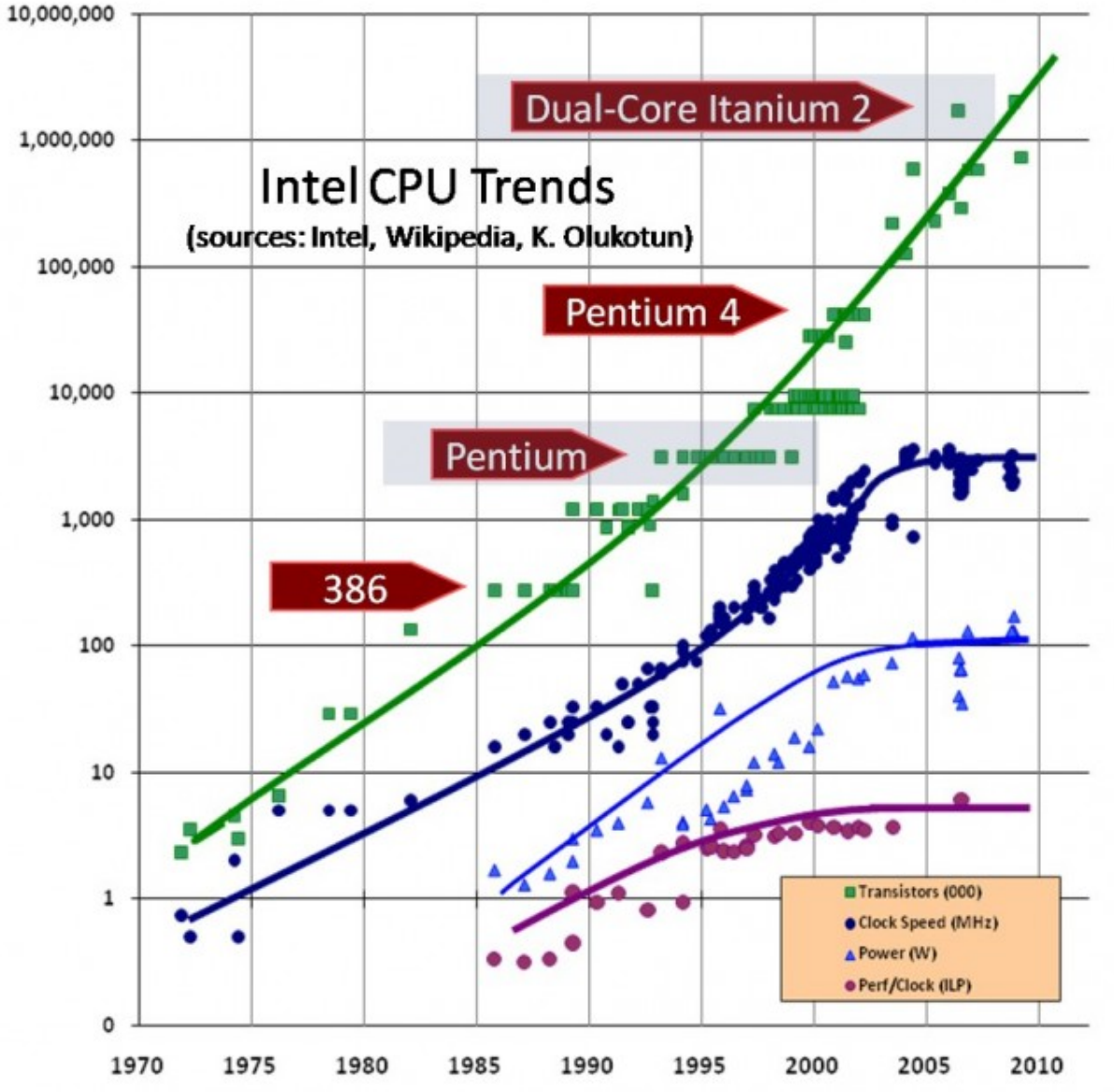
  ⇒ Sea change in chip design: multiple "cores"
  (2X processors per chip / ~ 2 years)
    - More simpler processors are more power efficient

Crossroads: Uniprocessor Performance

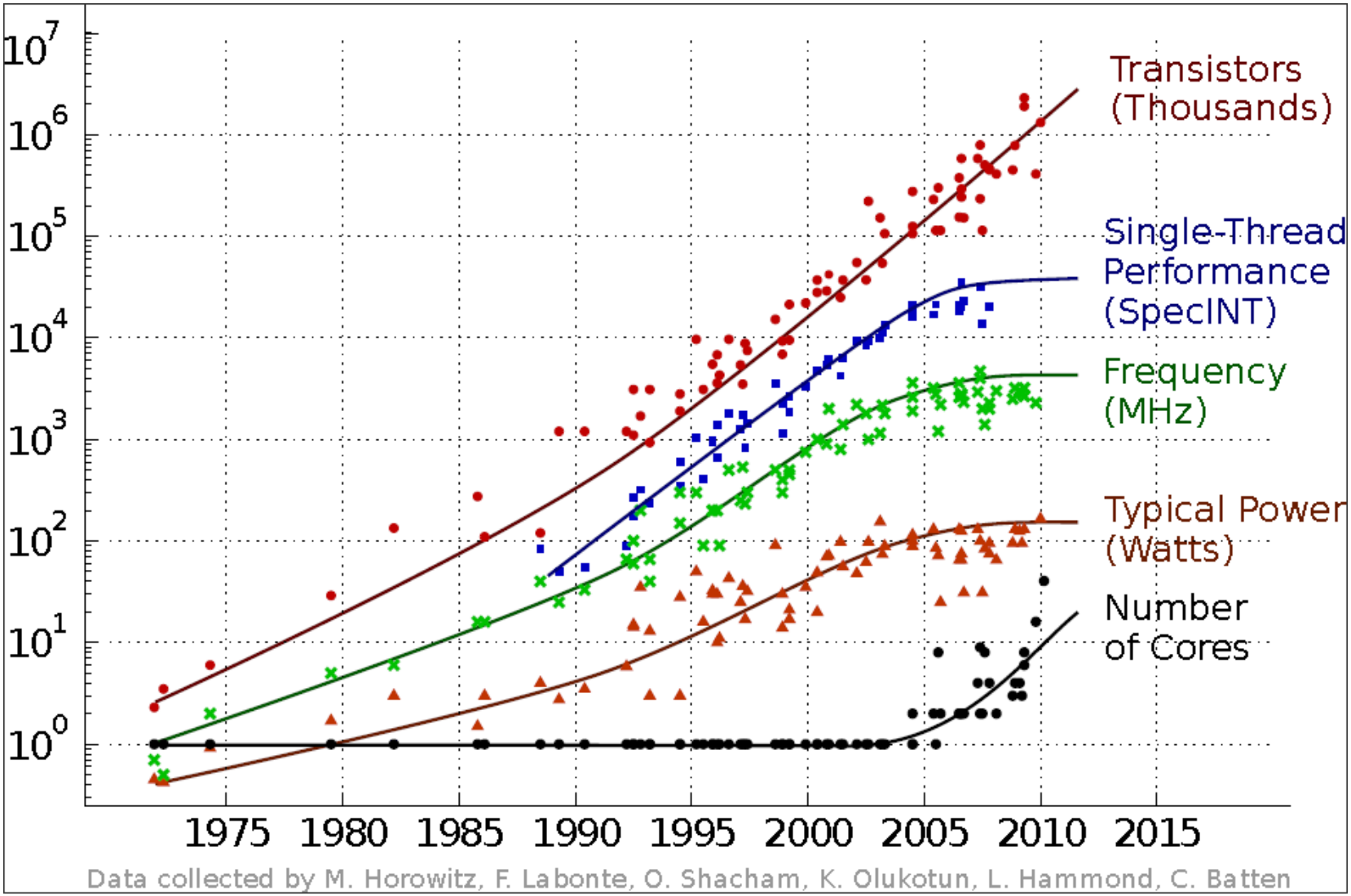


- **VAX : 25%/year 1978 to 1986**
- **RISC + x86: 52%/year 1986 to 2002**

The death of CPU scaling: From one core to many — and why we're still stuck - Joel Hruska



DATA PROCESSING IN EXASCALE-CLASS COMPUTER SYSTEMS - **Chuck Moore**

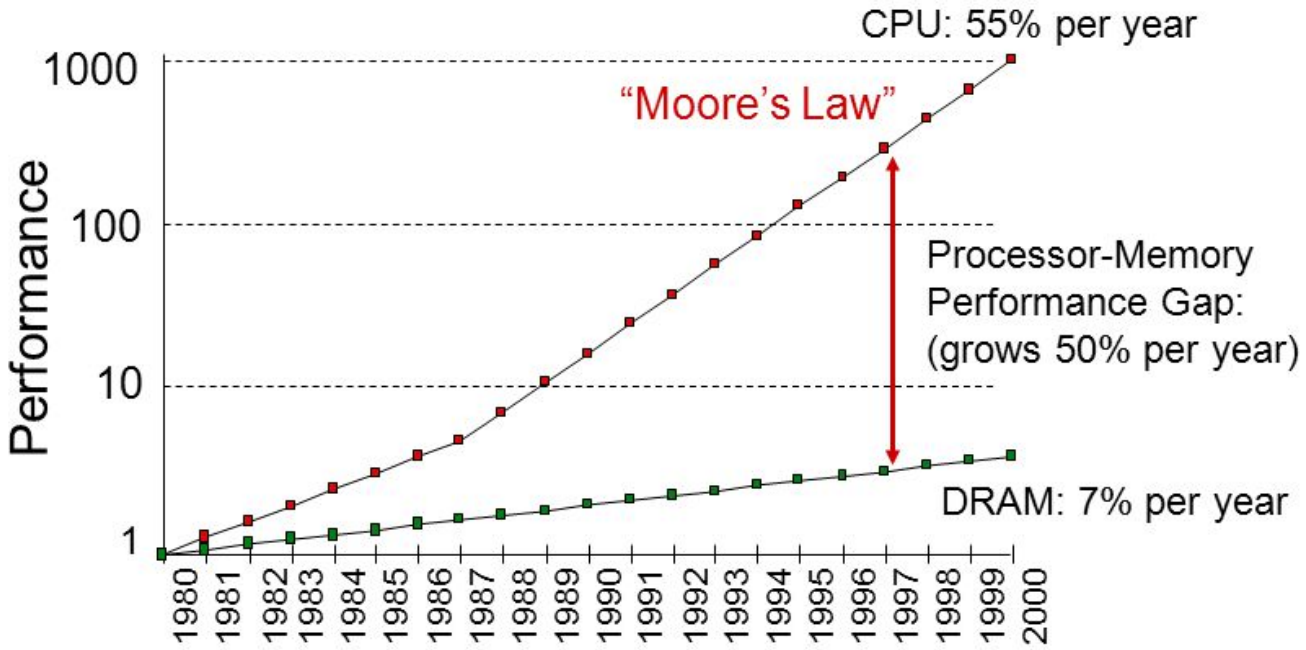


1. Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten. Dotted line extrapolations by C. Moore

Some of the key performance issues

- Use an efficient algorithm
  - and data structures
  - with efficient implementation
- Locality
  - It is difficult to underestimate the importance of locality
  - But it is almost invisible to the programmer
- Parallelism
  - Multiple threads (everyone talks about this)
  - Vector parallelism (just as important)

Processor vs Memory Performance



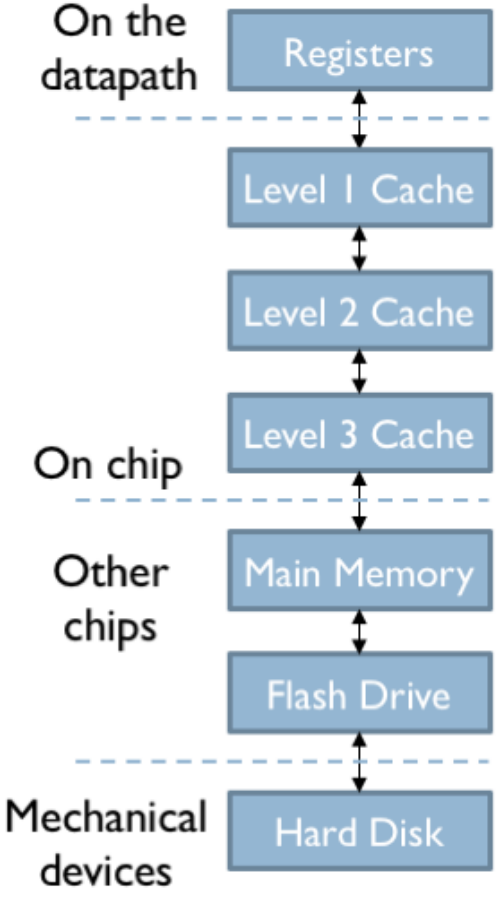
- 1980 – No cache in microprocessor
- 1995 – Two-level cache on microprocessor

CPU
60% per yr
2X in 1.5 yrs

DRAM
9% per yr
2X in 10 yrs

A Typical Memory Hierarchy

- Everything is a cache for something else…



Access time	Capacity	Managed By
1 cycle	1 KB	Software/Compiler
2-4 cycles	32 KB	Hardware
10 cycles	256 KB	Hardware
40 cycles	10 MB	Hardware
200 cycles	10 GB	Software/OS
10-100us	100 GB	Software/OS
10ms	1 TB	Software/OS

The Principle of Locality

- **The Principle of Locality:**
  - **Program access a relatively small portion of the address space at any instant of time.**

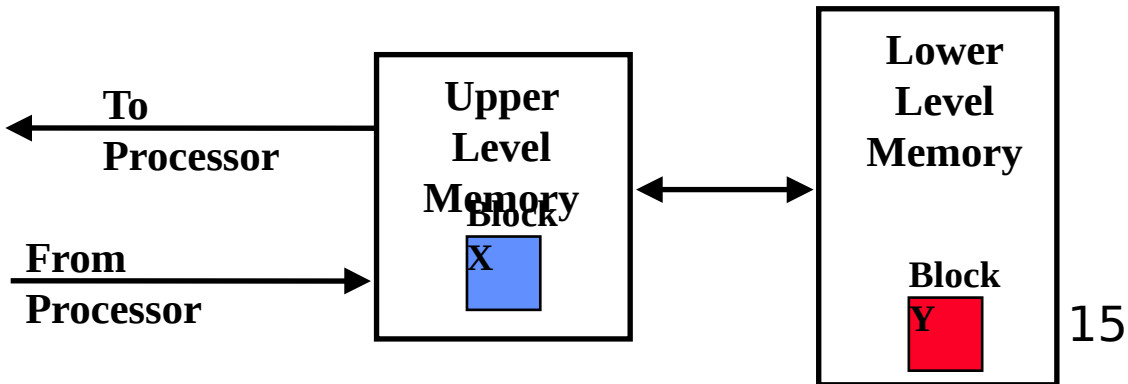
- **Two Different Types of Locality:**
  - **Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)**
  - **Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)**

- **Last 15 years, HW relied on locality for speed**

Locality is a property of programs which is exploited in machine design.

Memory Hierarchy: Terminology

- **Hit: data appears in some block in the upper level (example: Block X)**
  - **Hit Rate: the fraction of memory access found in the upper level**
  - **Hit Time: Time to access the upper level which consists of**
    RAM access time + Time to determine hit/miss
- **Miss: data needs to be retrieve from a block in the lower level (Block Y)**
  - **Miss Rate = 1 - (Hit Rate)**
  - **Miss Penalty: Time to replace a block in the upper level +**
    Time to deliver the block the processor
- **Hit Time << Miss Penalty (500 instructions on 21264!)**



Cache Measures

- ***Hit rate*: fraction found in that level**
  - **So high that usually talk about *Miss rate***
  - **Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory**
- **Average memory-access time**
  = Hit time + Miss rate x Miss penalty
  (ns or clocks)
- ***Miss penalty*: time to replace a block from lower level, including time to replace in CPU**
  - ***access time*: time to lower level**
    = f(latency to lower level)
  - ***transfer time*: time to transfer block**
    =f(BW between upper & lower levels)

Types of Cache Miss – the 3 C's

- **Compulsory Misses**
  - The first time a piece of memory is used, it is not in the cache
  - No choice but to load used memory at least once
- **Capacity Misses**
  - The size of the cache is limited
  - Once the cache is full, you must kick out something to load something else
- **Conflict Misses**
  - Most caches are not fully associative
  - Cache lines get swapped out even if cache is not full
  - If two regions of memory are mapped to the same cache line, then they conflict
- **Coherency Misses**
  - Cache invalidation due to cache coherency
  - Data is written by another core or processor

Basic Cache Optimizations

- **Reducing Miss Rate**

1. **Larger Block size (compulsory misses)**
2. **Larger Cache size (capacity misses)**
3. **Higher Associativity (conflict misses)**

- **Reducing Miss Penalty**

4. **Multilevel Caches**

- **Reducing hit time**

5. **Giving Reads Priority over Writes**
   - **E.g., Read complete before earlier writes in write buffer**

So why should the programmer care?

- **Cache misses can have a huge impact on program performance**
  - **L1 hit costs maybe two cycles**
  - **Main memory access costs maybe 400 cycles**
  - **Two orders of magnitude**

- **We can’t understand the performance of a program that accesses any significant amount of data without considering cache misses.**

What can the programmer do?

- **Sometimes there isn't much that can be done**
  - **Data access patterns are difficult to change**
- **But programmer decisions can have a huge impact on performance**
  - **Design of data structures**
  - **Order of access of data structures by the code**
- **Quite simple changes to the algorithms and/or data structures can have a very large impact on cache misses**

Data structures

- How can we design data structures to reduce cache misses?
- Make data structures more compact
  - E.g. use smaller data sizes or pack bits to reduce size
- Separate “hot” and “cold” data
  - Some parts of a data structure may be used much more than others. For example, some fields of a structure may be used on every access, whereas others are used only occasionally.
  - Consider whether to use structures of arrays rather than arrays of structures.
  - This may require very large changes to your program

Data structures

- **Collapse linked data structures**
  - **Linked lists can have terrible locality, because each element can be anywhere in memory.**
  - **Replacing linked lists with arrays usually improves locality because (at least in C/C++) arrays are contiguous in memory**
  - **Linked lists are not always a good replacement for arrays, so consider hybrid data structures.**
  - **E.g. linked lists where each element is a short array of values.**
- **Other linked structures can also be collapsed**
  - **Consider replacing binary trees with BTrees, etc.**
  - **Standard libraries should do more of this**

Data access patterns

- In many cases it is not feasible to change the data structures
- Instead change order of access
- E.g. matrix by vector multiplication

```
for ( i = 0; i < N; i++ ) {
    y[i] = 0;
    for ( j = 0; j < N; j++ ) {
        y[i] = y[i] + x[j] *A[i][j];
    }
}
```

Programs and the cache

- **For some types of programs the a cache-efficient version can be as much as five times faster than a regular version.**
  - **E.g. Matrix multiplication**
- **When you start to work with multiple cores, considering the cache becomes even more important.**
- **Different cores may compete for space within shared caches. L1 and L2 caches are usually private to the core. L3 is usually shared among all cores.**