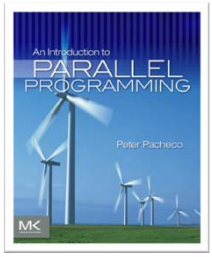
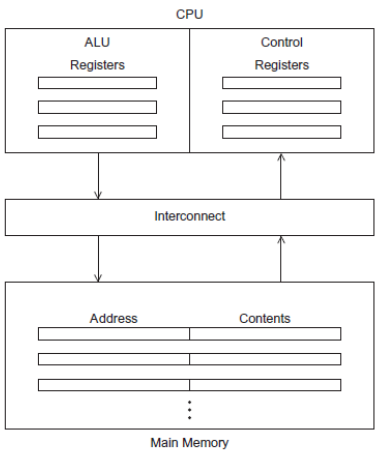


Chapter 2

Parallel Hardware and Parallel Software

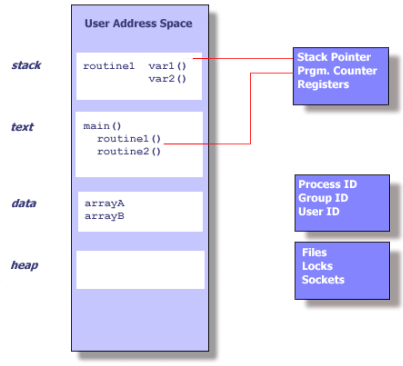
The Von Neuman Architecture



- **Control unit**: responsible for deciding which instruction in a program should be executed. (the boss)
- **ALU** (Arithmetic and logic unit): responsible for executing the actual instructions. (the worker)
- **Memory**:
  - A collection of locations, each of which is capable of storing instructions or data.
  - Every location has an address, which is used to access the location.

A Process (a task)

- An instance of a computer program that is being executed.
- Components of a process:
  - Memory space.
    - Program
    - data
  - Security information.
  - State of the process
    - Values of registers
    - Program counter, stack pointer, …
    - Allocated resources



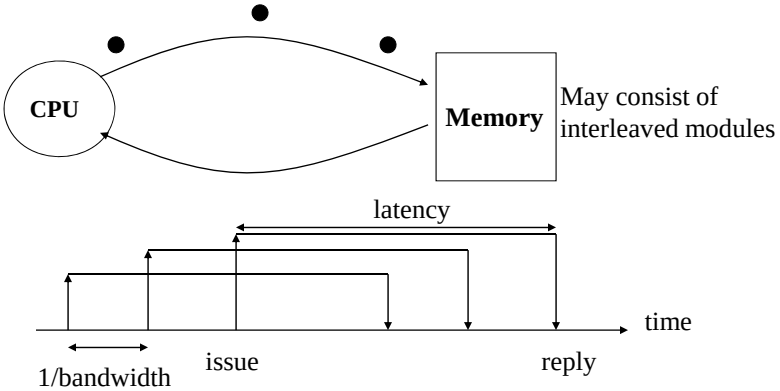
Multitasking (multiprogramming): Gives the illusion that multiple processes are running simultaneously.

- Each process takes turns running. (time slice)
- Switch context (process) when blocked or time is up

3

The Memory wall

- Memory system is often the bottleneck for many applications.
- Memory performance metrics:
  - latency
  - bandwidth.



4

Memory Latency: An Example

- 1 GHz processor (1 ns clock) with a multiply-add units capable of executing a multiply/add in each cycle
  - The peak processor rating is 2 GFLOPS ($2 * 10^9$ floating point operation per second).
- We want to compute the dot product of two vectors x[0],…,x[n-1] and y[0],…,y[n-1].

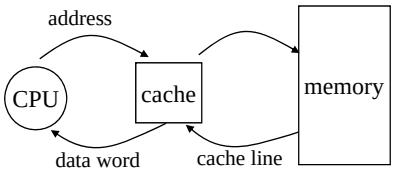
$$DP = \sum_{i=0}^{n-1} x[i] * y[i]$$

- Need two operands every ns for peak processor performance
- If DRAM has a latency of 100 ns, then it can supply only one operand every 100 cycle. Hence, can do a multiply/add (2 FLOPS) every 200 ns. That is one FLOP every 100 ns.
- Actual performance = 10 MFLOPS

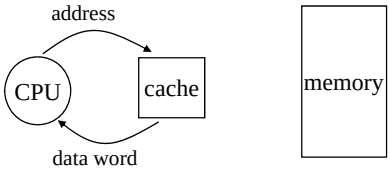
5

Overcoming the memory wall

- **Use caches:** take advantage of spatial and temporal locality



Cache miss



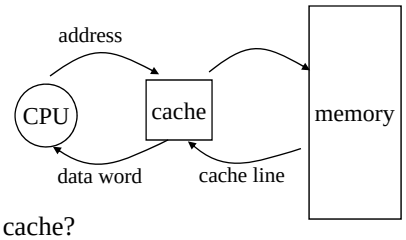
Cache hit

- **Use prefetching:** Can improve performance by pre-fetching. In this case, performance depends on memory bandwidth and not its latency. In the example above, if bandwidth is such that one operands can be fetched every 5ns, then can do a multiply/add every 10 ns. That is one FLOP every 5ns (200 MFLOP) – still not enough to match the 2GFLOP processor.
- **Use multithreading**

6

Review of cache operation

- **Issues to deal with:**
  - How do we know if a data item is in the cache?
  - If it is, how do we find it?
  - If it is not, what do we do?



- **It boils down to**
  - where do we put an item in the cache?
  - how do we identify the items in the cache?

- **Two solutions**
  - put item anywhere in cache (associative cache)
  - associate specific locations to specific items (direct mapped cache)

Cache mappings

- **Full associative** – a new line can be placed at any location in the cache.
- **Direct mapped** – each cache line has a unique location in the cache to which it will be assigned.
- ***n*-way set associative** – each cache line can be place in one of *n* different locations in the cache.

- When more than one line in memory can be mapped to several locations in cache we need to decide which line should be replaced or evicted.

Example

Memory Index	Cache Location		
	Fully Assoc	Direct Mapped	2-way
0	0, 1, 2, or 3	0	0 or 1
1	0, 1, 2, or 3	1	2 or 3
2	0, 1, 2, or 3	2	0 or 1
3	0, 1, 2, or 3	3	2 or 3
4	0, 1, 2, or 3	0	0 or 1
5	0, 1, 2, or 3	1	2 or 3
6	0, 1, 2, or 3	2	0 or 1
7	0, 1, 2, or 3	3	2 or 3
8	0, 1, 2, or 3	0	0 or 1
9	0, 1, 2, or 3	1	2 or 3
10	0, 1, 2, or 3	2	0 or 1
11	0, 1, 2, or 3	3	2 or 3
12	0, 1, 2, or 3	0	0 or 1
13	0, 1, 2, or 3	1	2 or 3
14	0, 1, 2, or 3	2	0 or 1
15	0, 1, 2, or 3	3	2 or 3

Table 2.1: Assignments of a 16-line main memory to a 4-line cache

Effect of cache line size

```
double A[MAX][MAX], x[MAX], y[MAX];
. . .
/* Initialize A and x, assign y = 0 */
. . .
/* First pair of loops */
for (i = 0; i < MAX; i++)
   for (j = 0; j < MAX; j++)
      y[i] += A[i][j]*x[j];
. . .
/* Assign y = 0 */
. . .
/* Second pair of loops */
for (j = 0; j < MAX; j++)
   for (i = 0; i < MAX; i++)
      y[i] += A[i][j]*x[j];
```

Cache Line	Elements of A			
0	A[0][0]	A[0][1]	A[0][2]	A[0][3]
1	A[1][0]	A[1][1]	A[1][2]	A[1][3]
2	A[2][0]	A[2][1]	A[2][2]	A[2][3]
3	A[3][0]	A[3][1]	A[3][2]	A[3][3]

Direct mapped cache with line size = 4 words

Performance of second pair of loops is bad if cache can hold only 2 lines

Effect of data layout: An example

Consider the following code fragment:

```
for (j = 0; j < 1000; j++)
    column_sum[j] = 0.0;
     for (i = 0; i < 1000; i++)
        column_sum[j] += b[i][j];
```

The code fragment sums columns of the matrix b into a vector column_sum.

- The vector column_sum is small and easily fits into the cache
- The matrix b is accessed in a column order
- With row major storage, strided access results in very poor performance.

What is row major storage?

$b_{1,1}\ b_{1,2}\ \cdots\ b_{1,10}$
$b_{2,1}\ b_{2,2}\ \cdots\ b_{2,10}$
$\cdots\ \cdots\ \cdots$
$b_{10,1} b_{10,2}\ \cdots\ b_{10,10}$

Stored as

$b_{1,1}\ b_{1,2}\ \cdots b_{1,10}\ b_{2,1}\ b_{2,2}\ \cdots\ b_{2,10}\ b_{3,1}\ b_{3,2}\cdots\ b_{3,10}\cdots$

11

Change the access pattern

We can fix the above code as follows:

```
for (j = 0; j < 1000; j++)
     column_sum[j] = 0.0;
for (i = 0; i < 1000; i++)
     for (j = 0; j < 1000; j++)
          column_sum[j] += b[i][j];
```

In this case, the matrix is traversed in a row-order and performance can be expected to be significantly better --- Why?

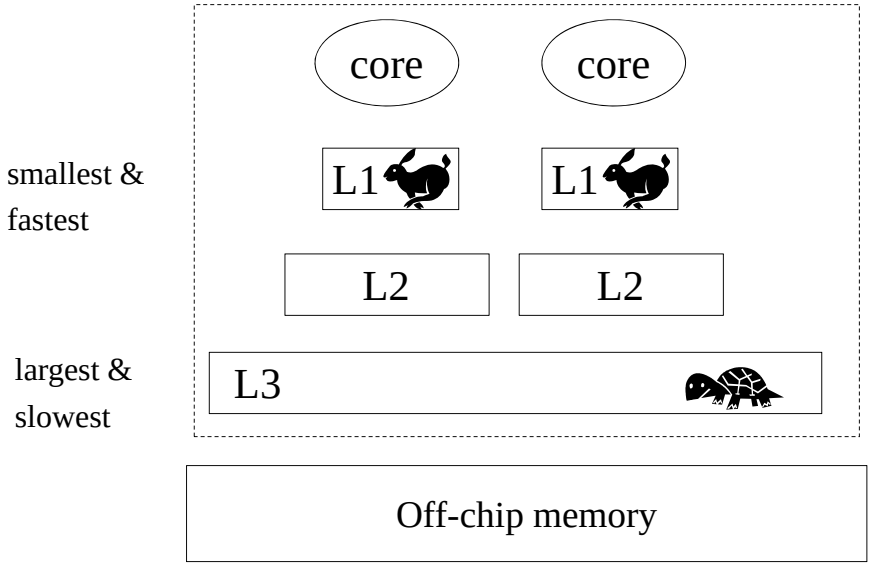
Memory layouts and order of computation can have a significant impact on the spatial and temporal locality.

12

Issues with cache

- When a CPU writes data to cache, the value in cache may be inconsistent with the value in main memory.
- **Write-through** caches handle this by updating the data in main memory at the time it is written to cache.
- **Write-back** caches mark data in the cache as dirty. When the cache line is replaced by a new cache line from memory, the dirty line is written to memory.

Multiple levels of caches in multi-core processors

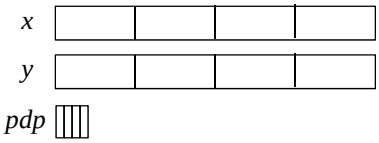


Multithreading: originally designed as an alternate Approach for Hiding Memory Latency

We illustrate threads with the simple example of dot product of two vectors, *x* and *y*.:

```
dp = 0 ;
for (i = 0 ;  i < n ;  i++)
    dp +=  x[i] * y[i]
```

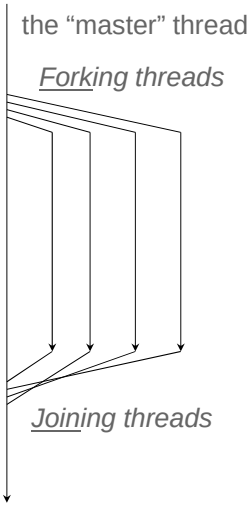
```
dp = 0 ;
for (k = 0; k < 4; k++)
    partial_product (k*n/4, n/4);
for (k = 0; k < 4; k++)
    dp += pdp[k] ;

void partial_product (a , b);
{  pdp[k] = 0 ;
  for (i = a ;  i < a+b ;  i++)
      pdp[k] += x[i] * y[i] ;
  return ;
}
```



15

Instead of calling the function *partial_product()*, create a new thread that will execute that function.

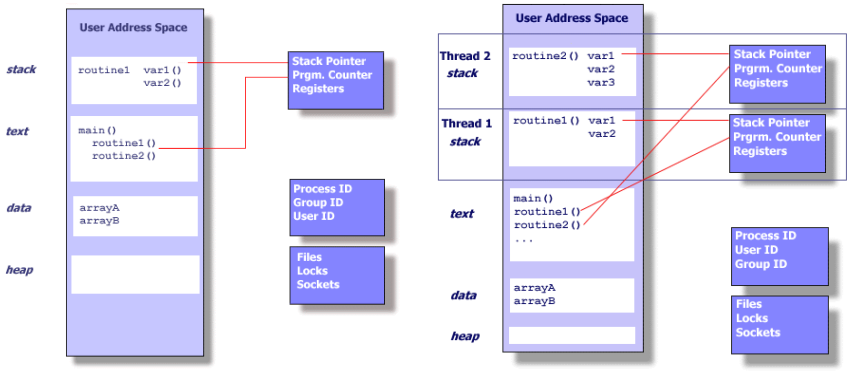
```
dp = 0 ;
for (k = 0; k < 4; k++)              /* fork threads */
    create_thread (partial_product, k*n/4, n/4);
Wait until all threads return ;      /* join threads */
for (k = 0; k < 4; k++)
    dp +=  pdp[k] ;

void partial_product (a , b);
{ pdp[k] = 0 ;
  for (i = a ;  i < a+b ;  i++)
      pdp[k] += x[i] * y[i] ;
  return ;
}
```



16

A Thread is a light weight process

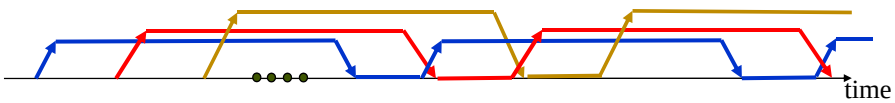
- Threads within the same process
  - Share the memory address space
  - Each has its own registers, program counter and stack pointer
  - The OS schedules processes but a thread library function schedules threads within a process



17

Multithreading for Latency Hiding: Example

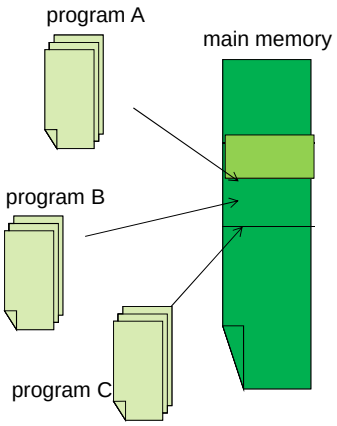
- In the example of slide 16, the first thread accesses a pair of vector elements and waits for them.
- In the meantime, the second thread can access two other vector elements in the next cycle, and so on.
- After *L cycles*, where *L* is the latency of the memory system, the first thread gets the requested data from memory and can perform *B* computations, where *B* is the number of words in a cache block size.
- When done, the data items for the next thread arrive, and so on. If there are *L/B* threads, then, we can perform a computation in every clock cycle.
- **Assumptions made**:
  - The memory system is capable of servicing multiple outstanding requests, and its bandwidth can keep up with the processing rate.
  - No time penalty for switching threads .



18

Virtual memory

- If we run a very large program or a program that accesses very large data sets, all of the instructions and data may not fit into main memory.
- Virtual memory is a cache for secondary storage (swap space).
- Exploits the principle of spatial and temporal locality.
- It only keeps the active parts of running programs in main memory (unit of storage = a page).



Virtual page numbers

- When a program is compiled its pages are assigned *virtual* page numbers.
- When the program is run, a table is created that maps the virtual page numbers to physical addresses.
- A **page table** is used to translate the virtual address into a physical address.

Virtual Address									
Virtual Page Number					Byte Offset				
31	30	···	13	12	11	10	···	1	0
1	0	···	1	1	0	0	···	1	1

bit address
bit value

Virtual Address Divided into Virtual Page Number and Byte Offset

Translation-lookaside buffer (TLB)

- Using a page table has the potential to significantly increase each program's overall run-time.
- A special address translation cache in the processor.

- It caches a small number of entries (typically 16–512) from the page table in very fast memory.
- **Page fault** – attempting to access a valid physical address for a page in the page table but the page is only stored on disk.

Instruction Level Parallelism (ILP)

- Attempts to improve processor performance by having multiple processor components or **functional units** simultaneously executing instructions.

- **Pipelining** - functional units are arranged in stages.

- **Multiple issue** - multiple instructions can be simultaneously initiated.

Pipelining

Time	Operation	Operand 1	Operand 2	Result
1	Fetch operands	9.87×10^4	6.54×10^3	
2	Compare exponents	9.87×10^4	6.54×10^3	
3	Shift one operand	9.87×10^4	0.654×10^4	
4	Add	9.87×10^4	0.654×10^4	10.524×10^4
5	Normalize result	9.87×10^4	0.654×10^4	1.0524×10^5
6	Round result	9.87×10^4	0.654×10^4	1.05×10^5
7	Store result	9.87×10^4	0.654×10^4	1.05×10^5

Add the floating point numbers 9.87×10^4 and 6.54×10^3

Pipelining example

```
float x[1000], y[1000], z[1000];
. . .
for (i = 0; i < 1000; i++)
   z[i] = x[i] + y[i];
```

- Assume each operation takes one nanosecond.
- This loop takes about 7000 nanoseconds.

- Divide the floating point adder into 7 separate pieces of hardware or functional units.
- Output of one functional unit is input to the next
- Loop takes 1006 nanoseconds.

Pipelining

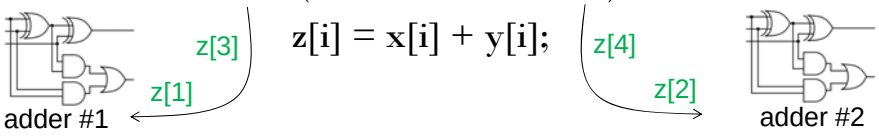
Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Pipelined Addition.
Numbers in the table are subscripts of operands/results.

Multiple Issue

- Multiple issue processors replicate functional units and try to simultaneously execute different instructions in a program.

```
for (i = 0; i < 1000; i++)
    z[i] = x[i] + y[i];
```



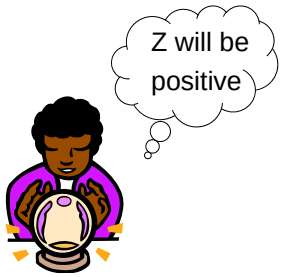
- **VLIW:** functional units are scheduled at compile time (static scheduling).
- **Superscalers:** functional units are scheduled at run-time (dynamic scheduling).

Speculation

- In order to make use of multiple issue, the system must find instructions that can be executed simultaneously.
- In speculation, the compiler or the processor makes a guess about an instruction, and then executes the instruction on the basis of the guess.

```
z = x + y ;
i f ( z > 0)  w = x - y ;
e l s e       w = x + y ;
```

If the system speculates incorrectly, it must go back and recalculate w.



Multi-threading

Provides a means to continue doing useful work when the currently executing task has stalled (ex. wait for long memory latency)

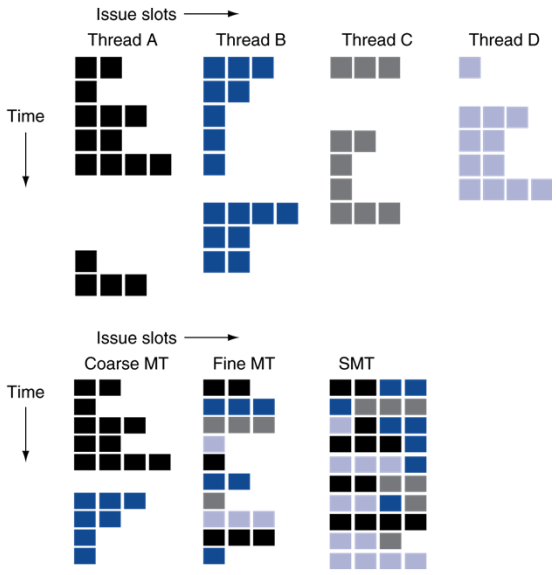
- **Software-based** multithreading (Posix Threads)
  - Hardware traps on a long-latency operation
  - Software then switches the running process to another (context switching)
  - Issues with the overhead
- **Hardware –based** multithreading
  - For user defined threads or compiler generated threads
  - Hardware support for context switching
  - Replicate registers (including PC and stack pointer)
- Example: IBM Power5 and Pentium-4 supports SMT

Hardware multi-threading

- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (such as the stalls resulting from data hazards)
- SMT in multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available

29

Hardware-based multi-threading



30