Uniprocessors

HPC

Prof. Robert van Engelen

Overview

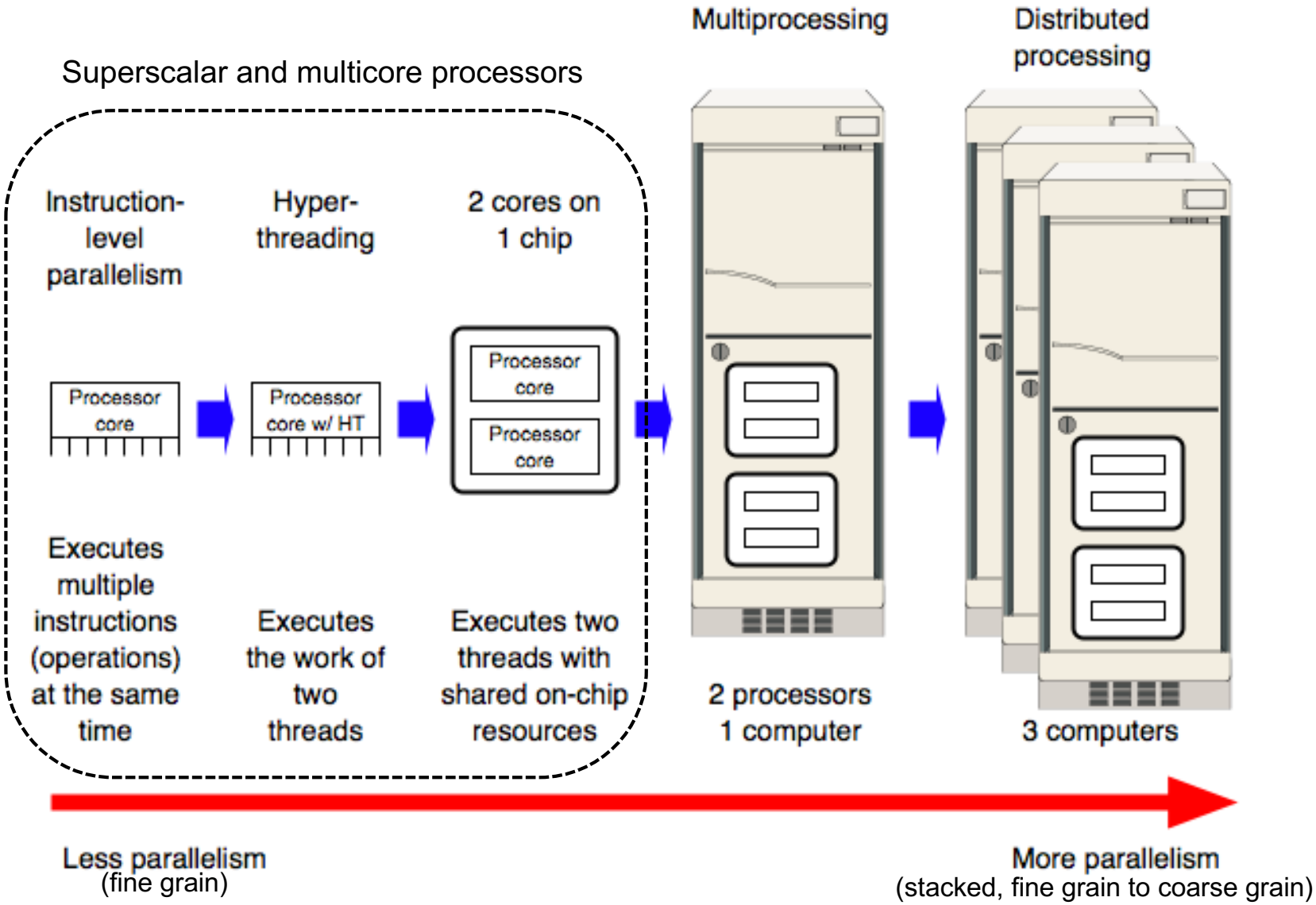
PART I: Uniprocessors and Compiler Optimizations

- Uniprocessors
  - Processor architectures
  - Instruction set architectures
  - Instruction scheduling and execution
- Data storage
  - Memory hierarchy
  - Caches, TLB
- Compiler optimizations

PART II: Multiprocessors and Parallel Programming Models

- Multiprocessors
  - Pipeline and vector machines
  - Shared memory
  - Distributed memory
  - Message passing
- Parallel programming models
  - Shared vs distributed memory
  - Hybrid
  - BSP

Levels of Parallelism



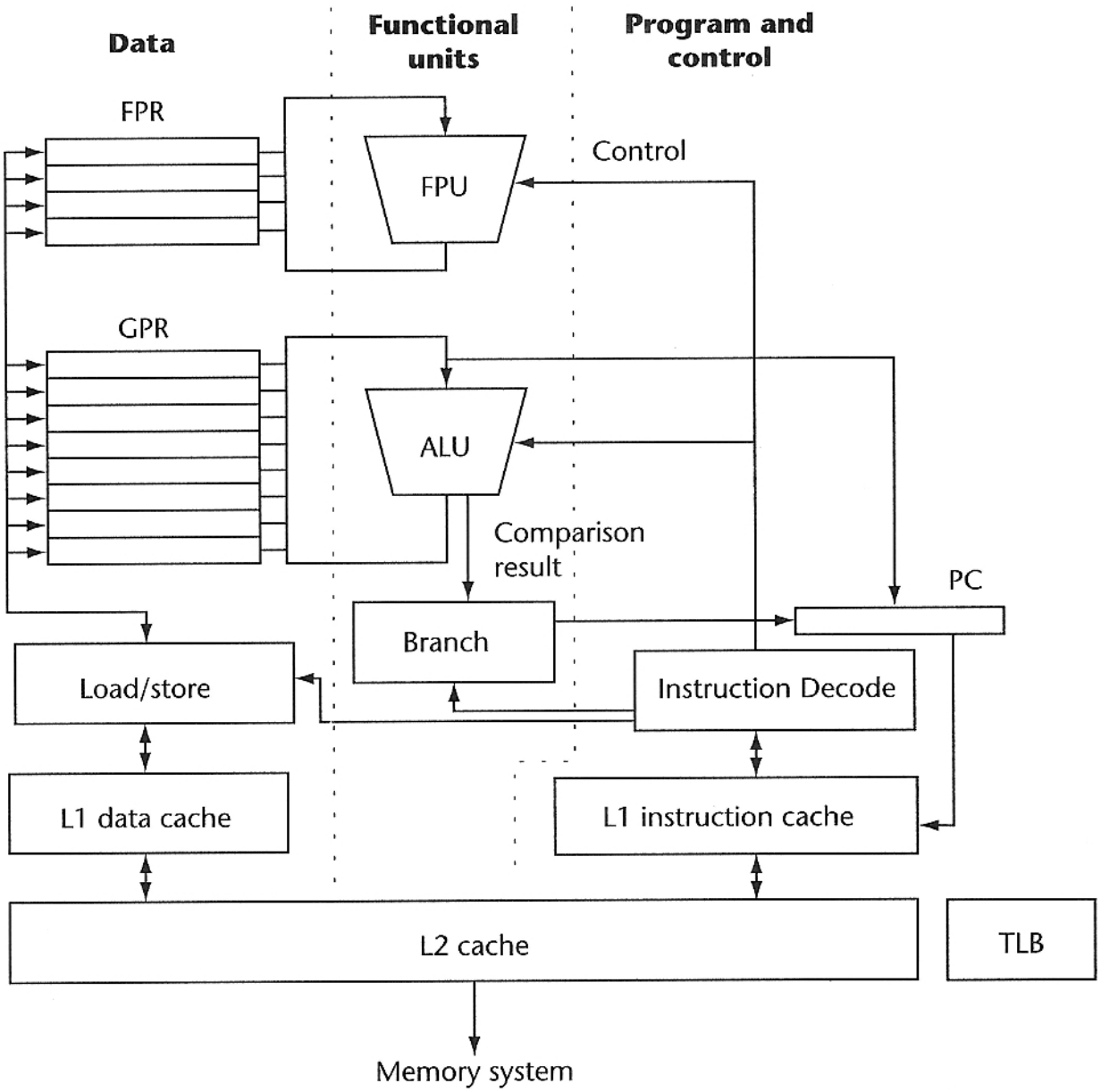
Processor Families

Family	*Instruction Set Architecture (ISA)*	*Processors*
CISC	DEC VAX Intel 80x86 (IA-32)	VAX-11/780 Intel Pentium Pro
Vector	CRAY Convex	CRAY T90 Convex C-4
RISC	HP PA-RISC SGI MIPS DEC Alpha Sun Sparc IBM PowerPC	PA-8600 MIPS R10000 Alpha21264 Sun UltraSparc-III IBM Power3
VLIW	Multiflow Cydrome Intel IA-64	Multiflow Trace Cydrome Cydra-5 Intel Itanium

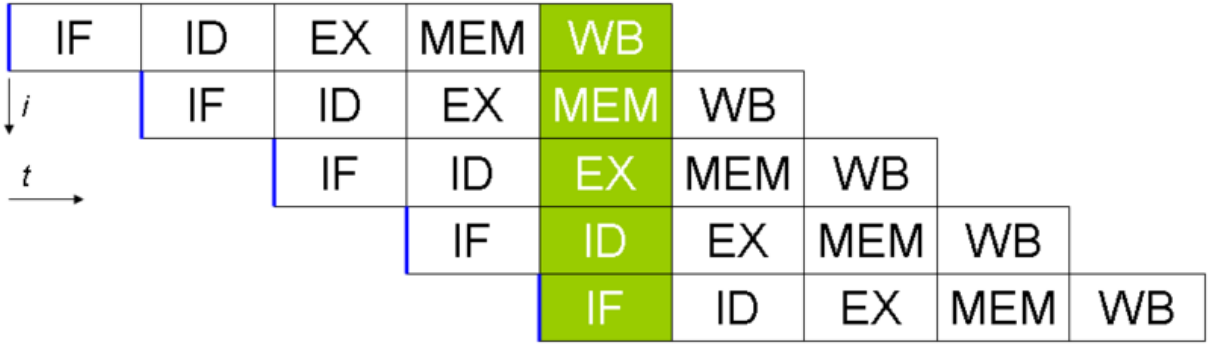
CISC, RISC, VLIW, and Vector Processor Families

- **CISC** (*complex instruction set computer*)
  - CISC ISAs offer specialized instructions of various (and variable) length
  - Instructions typically executed in *microcode*
- **RISC** (*reduced instruction set computer*)
  - No microcode and relatively few instructions
  - Many RISC processors are *superscalar* (for instruction-level parallelism)
  - Only load and store instructions access memory
  - Single common instruction word length
  - More registers than CISC
- **VLIW** (*very long instruction word*)
  - Bundles multiple instructions for parallel execution
  - Dependences between instructions in a bundle are prohibited
  - More registers than CISC and RISC
- Vector machines
  - Single instructions operate on vectors of data (not necessarily parallel)
  - Multiple vector instructions can be chained

Superscalar Architectures



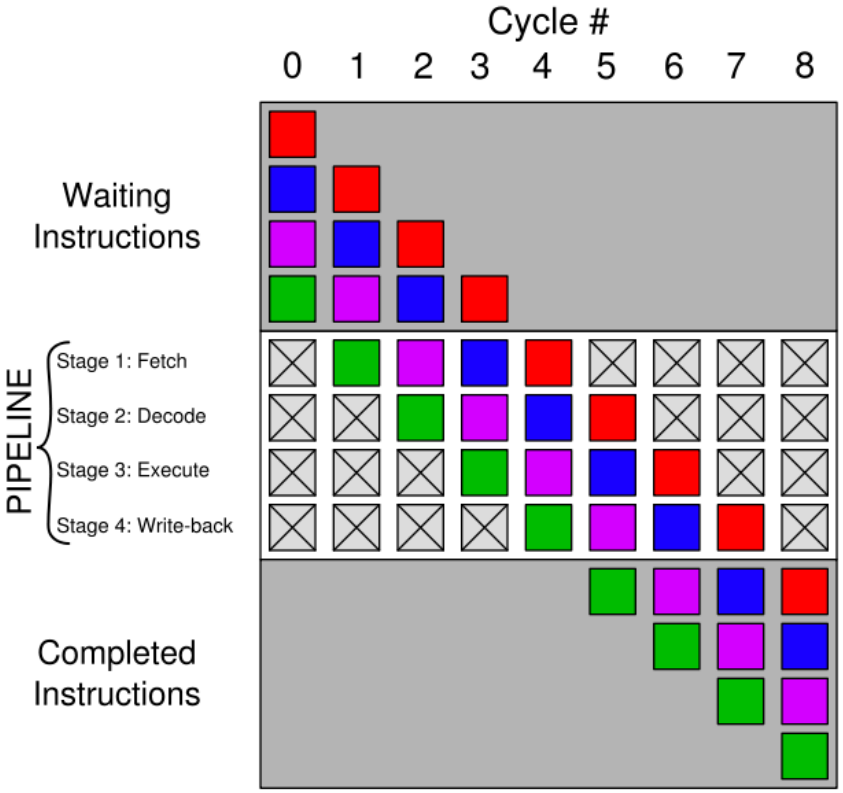
Instruction Pipeline



Five instructions are in the pipeline at different stages

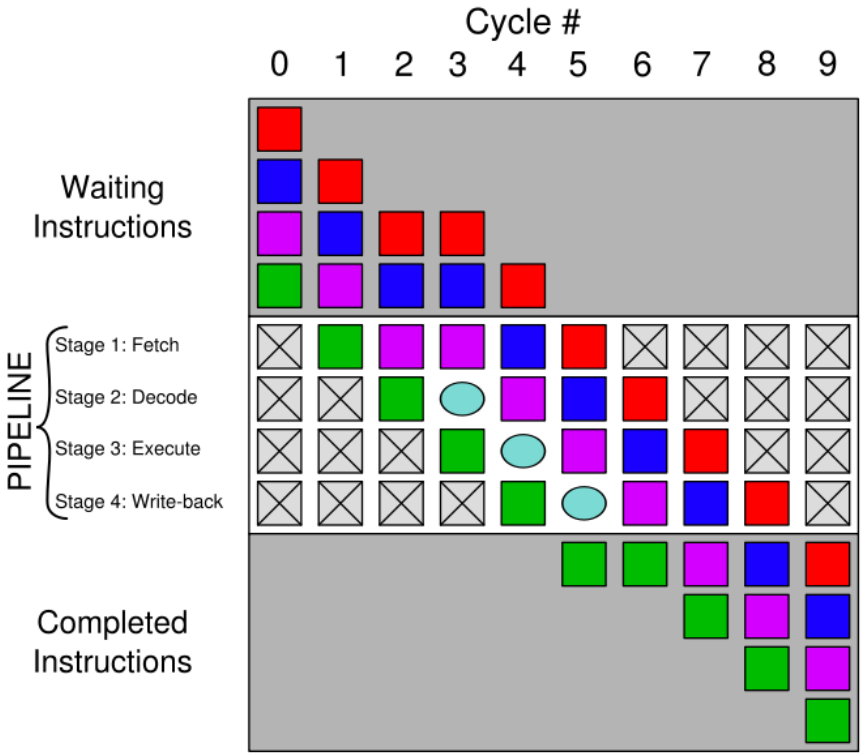
- An *instruction pipeline* increases the *instruction bandwidth*
- Classic 5-stage pipeline:
  - IF instruction fetch
  - ID instruction decode
  - EX execute (functional units)
  - MEM load/store
  - WB write back to registers and forward results into the pipeline when needed by another instruction

Instruction Pipeline Example



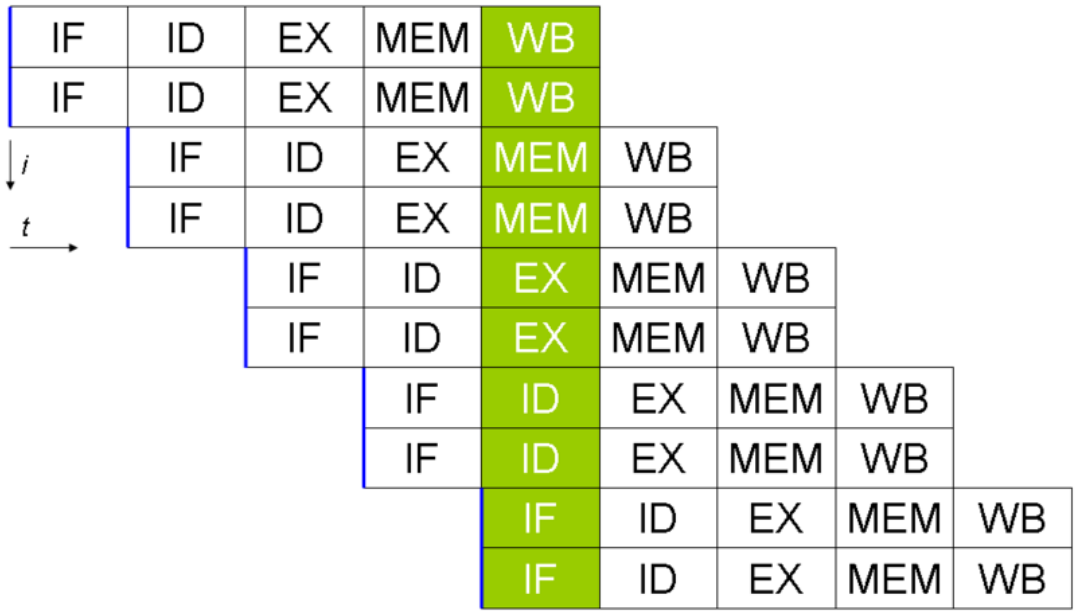
- Example 4-stage pipeline
- Four instructions (green, purple, blue, red) are processed in this order
  - Cycle 0: instructions are waiting
  - Cycle 1: Green is fetched
  - Cycle 2: Purple is fetched, green decoded
  - Cycle 3: Blue is fetched, purple decoded, green executed
  - Cycle 4: Red is fetched, blue decoded, purple executed, green in write-back
  - Etc.

Instruction Pipeline Hazards



- *Pipeline hazards*
  - From data dependences
  - Forwarding in the WB stage can eliminate some data dependence hazards
  - From instruction fetch latencies (e.g. I-cache miss)
  - From memory load latencies (e.g. D-cache miss)
- A hazard is resolved by *stalling the pipeline*, which causes a *bubble* of one ore more cycles
- Example
  - Suppose a stall of one cycle occurs in the IF stage of the purple instruction
  - Cycle 3: Purple cannot be decoded and a no-operation (NOP) is inserted

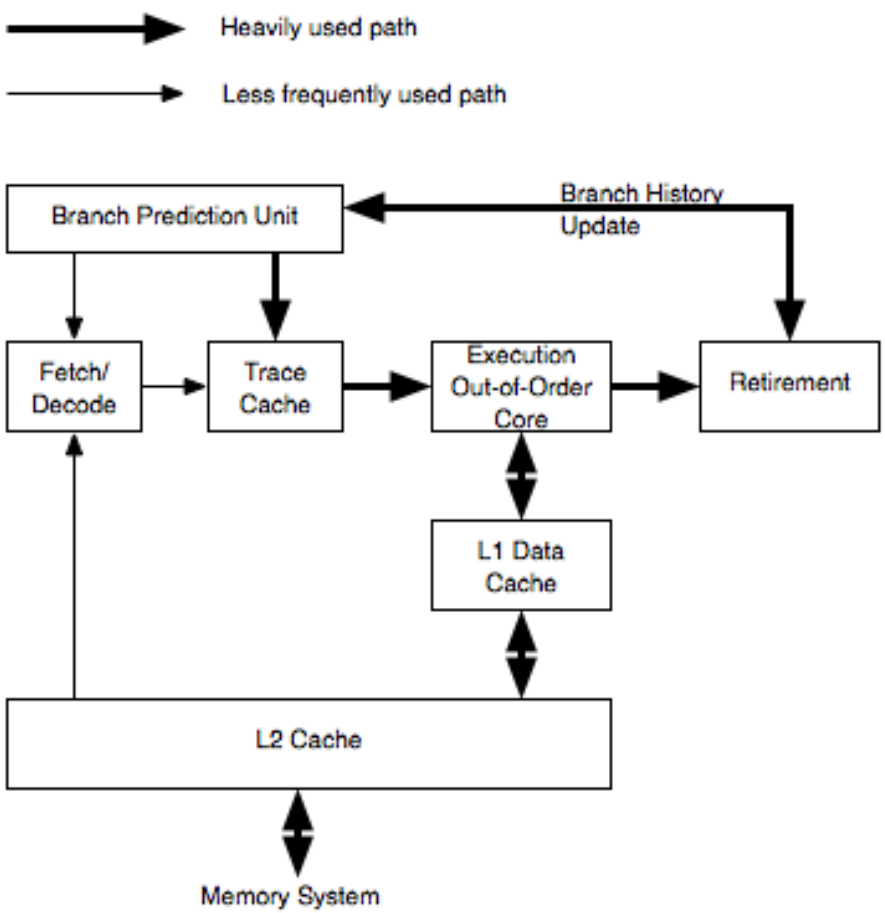
N-way Superscalar RISC



2-way superscalar RISC pipeline

- RISC instructions have the same word size
  - Can fetch multiple instructions without having to know the instruction content of each
- N-way superscalar RISC processors fetch N instructions each cycle
  - Increases the instruction-level parallelism

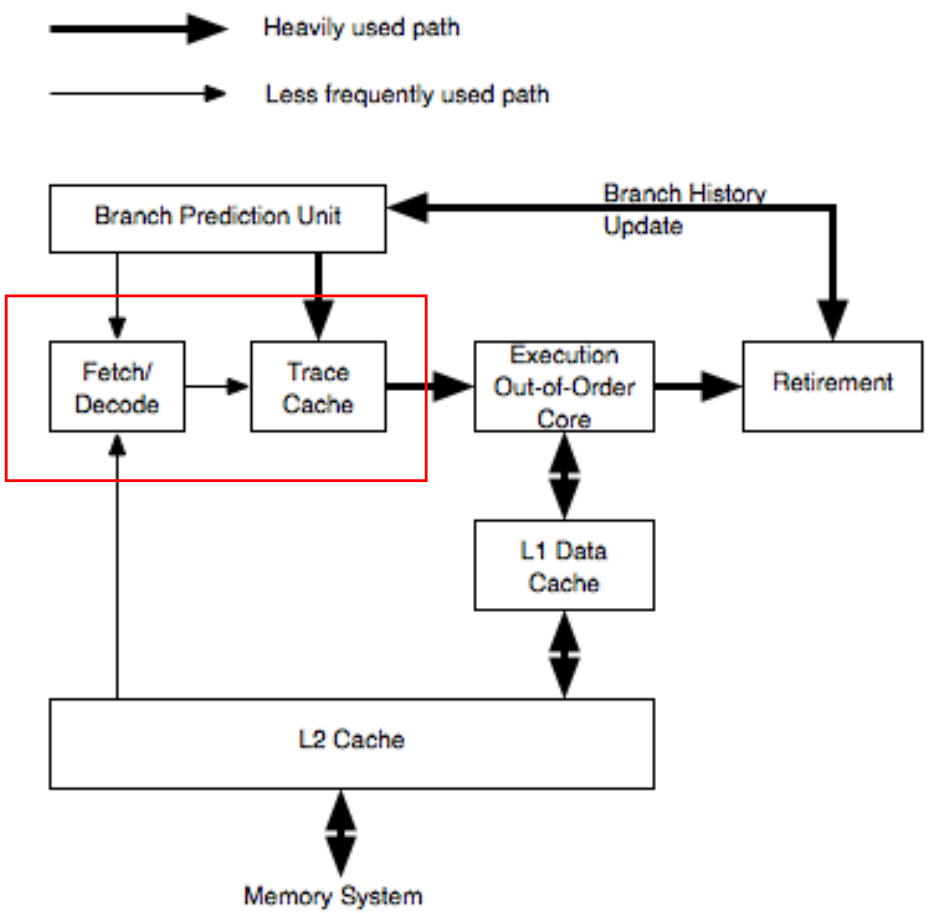
A CISC Instruction Set in a Superscalar Architecture



Simplified block diagram of the Intel Pentium 4

- Pentium processors translate CISC instructions to RISC-like μOps
- Advantages:
  - Higher instruction bandwidth
  - Maintains *instruction set architecture* (ISA) compatibility
- Pentium 4 has a 31 stage pipeline divided into three main stages:
  - Fetch and decode
  - Execution
  - Retirement

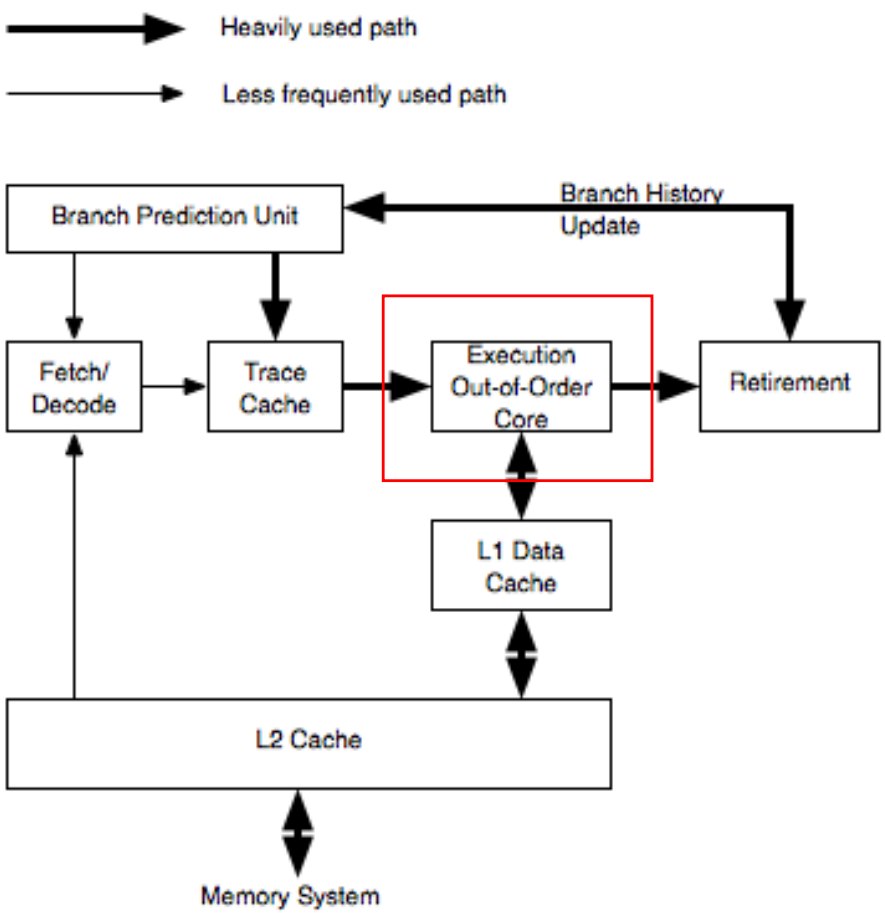
Instruction Fetch and Decode



Simplified block diagram of the Intel Pentium 4

- Pentium 4 decodes instructions into μOps and deposits the μOps in a *trace cache*
  - Allows the processor to fetch the μOps trace of an instruction that is executed again (e.g. in a loop)
- Instructions are fetched:
  - Normally in the same order as stored in memory
  - Or fetched from branch targets predicted by the *branch prediction unit*
- Pentium 4 only decodes one instruction per cycle, but can deliver up to three μOps per cycle to the execution stage
- RISC architectures typically fetch multiple instructions per cycle

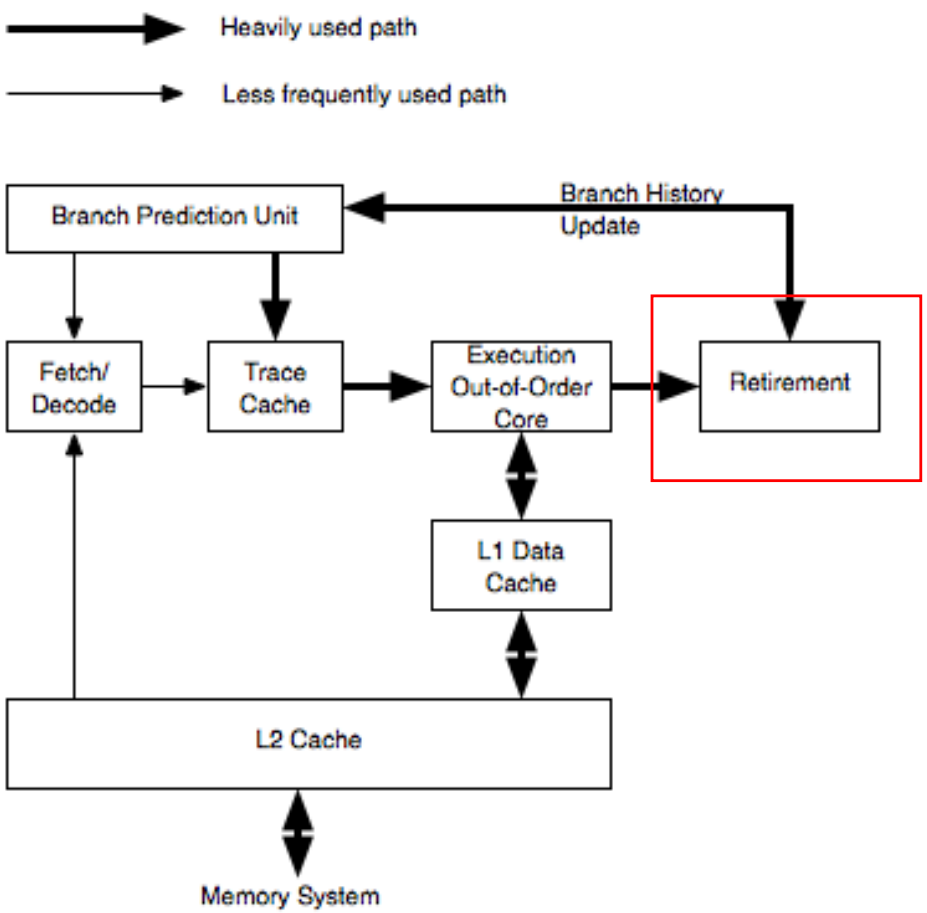
Instruction Execution Stage



Simplified block diagram of the Intel Pentium 4

- Executes multiple μOps in parallel
  - *Instruction-level parallelism* (ILP)
- The scheduler marks a μOp for execution when all operands of the a μOp are ready
  - The μOps on which a μOp depends must be executed first
  - A μOp can be *executed out-of-order* in which it appeared
- Pentium 4: a μOp is re-executed when its operands were not ready
- On Pentium 4 there are 4 *ports* to send a μOp into
- Each port has one or more fixed *execution units*
  - Port 0: ALU0, FPMOV
  - Port 1: ALU1, INT, FPEXE
  - Port 2: LOAD
  - Port 3: STORE

Retirement



Simplified block diagram of the Intel Pentium 4

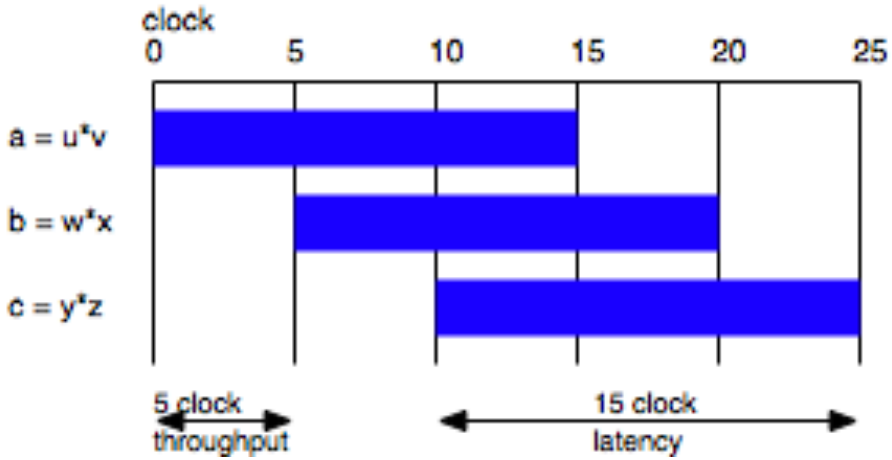
- Looks for instructions to mark completed
  - Are all μOps of the instruction executed?
  - Are all μOps of the preceding instruction retired? (putting instructions back in order)
- Notifies the *branch prediction unit* when a branch was incorrectly predicted
  - Processor stops executing the wrongly predicted instructions and discards them (takes ≈10 cycles)
- Pentium 4 retires up to 3 instructions per clock cycle

Software Optimization to Increase CPU Throughput

- Processors run at maximum speed (high instruction per cycle rate (IPC)) when
  1. There is a good *mix of instructions* (with *low latencies*) to keep the functional units busy
  2. Operands are available quickly from registers or D-cache
  3. The *FP to memory operation ratio* is high (FP : MEM > 1)
  4. Number of *data dependences* is low
  5. *Branches* are easy to predict
- The processor can only improve #1 to a certain level with *out-of-order scheduling* and partly #2 with *hardware prefetching*
- Compiler optimizations effectively target #1-3
- The programmer can help improve #1-5

Instruction Latency and Throughput

```
a=u*v; b=w*x; c=y*z
```



- *Latency*: the number of clocks to complete an instruction when all of its inputs are ready
- *Throughput*: the number of clocks to wait before starting an *identical* instruction
  - Identical instructions are those that use the same execution unit
- The example shows three multiply operations, assuming there is only one multiply execution unit
- Actual typical latencies (in cycles)
  - Integer add: 1
  - FP add: 3
  - FP multiplication: 3
  - FP division: 31

Instruction Latency Case Study

- Consider two versions of Euclid's algorithm
  1. Modulo version
  2. Repetitive subtraction version
- Which is faster?

```
int find_gcf1(int a, int b)
{ while (1)
  { a = a % b;
    if (a == 0) return b;
    if (a == 1) return 1;
    b = b % a;
    if (b == 0) return a;
    if (b == 1) return 1;
  }
}
```

Modulo version

```
int find_gcf2(int a, int b)
{ while (1)
  { if (a > b)
      a = a - b;
    else if (a < b)
      b = b - a;
    else
      return a;
  }
}
```

Repetitive subtraction

Instruction Latency Case Study

- Consider the cycles estimated for the case a=48 and b=40

Instruction	*#*	*Latency*	*Cycles*
Modulo	2	68	136
Compare	3	1	3
Branch	3	1	3
Other	6	1	6
Total	14		148

Modulo version

Instruction	*#*	*Latency*	*Cycles*
Subtract	5	1	5
Compare	5	1	5
Branch	14	1	14
Other	0		
Total	24		24

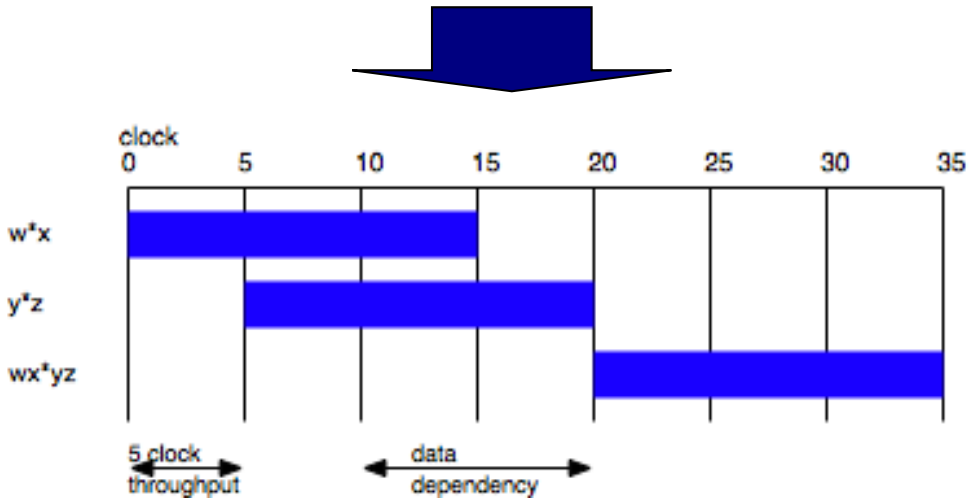
Repetitive subtraction

- Execution time for all values of a and b in [1..9999]

Modulo version	*Repetitive subtraction*	*Blended version*
18.55 sec	14.56 sec	12.14 sec

Data Dependences

```
(w*x)*(y*z)
```



- Instruction level parallelism is limited by *data dependences*
- Types of dependences:
  - RAW: *read-after-write* also called *flow dependence*
  - WAR: *write-after-read* also called *anti dependence*
  - WAW: *write-after-write* also called *output dependence*
- The example shows a RAW dependence
- WAR and WAW dependences exist because of *storage location reuse* (overwrite with new value)
  - WAR and WAW are sometimes called *false dependences*
  - RAW is a *true dependence*

Data Dependence Case Study

- Removing redundant operations by (re)using (temporary) space may increase the number of dependences
- Example: two versions to initialize a finite difference matrix
  1. Recurrent version with lower FP operation count
  2. Non-recurrent version with fewer dependences
- Which is fastest depends on effectiveness of loop optimization and *instruction scheduling* by compiler (and processor) to *hide latencies* and the number of *distinct memory loads*

```
dxi=1.0/h(1)
do i=1,n
  dxo=dxi
  dxi=1.0/h(i+1)
  diag(i)=dxo+dxi
  offdiag(i)=-dxi
enddo
```

With recurrence

(cross iter dep not shown)

WAR RAW

```
do i=1,n
  dxo=1.0/h(i)
  dxi=1.0/h(i+1)
  diag(i)=dxo+dxi
  offdiag(i)=-dxi
enddo
```

Without recurrence

(cross iter dep not shown)

Case Study (1)

Intel Core 2 Duo 2.33 GHz

```
dxi = 1.0/h[0];
for (i=1; i<n; i++)
{ dxo = dxi;
  dxi = 1.0/h[i+1];
  diag[i] = dxo+dxi;
  offdiag[i] = -dxi;
}
```

```
gcc -O3 fdinit.c
time ./a.out
```

0.135 sec

One cross-iteration flow dependence but fewer memory loads

Dependence spans multiple instructions, so has no big impact

```
for (i=1; i<n; i++)
{ dxo = 1.0/h[i];
  dxi = 1.0/h[i+1];
  diag[i] = dxo+dxi;
  offdiag[i] = -dxi;
}
```

```
gcc -O3 fdinit.c
time ./a.out
```

0.270 sec

One more memory load but fewer deps

Case Study (2)

Timing comparison of non-optimized kernels
recurrent version: 1.856 sec
non-recurrent version: 7.315 sec

Sun Studio Analyzer [fdinit.er]

File View Timeline Help

Functions | Callers-Callees | Source | Disassembly | Timeline | Experim

```
User CPU (sec.)   Source File: ./fdinit.f
                  Object File: ./fdinit
                  Load Object: <fdinit>
0.        21.         SUBROUTINE initarray1(h, n)
              <Function: initarray1_>
          22.         REAL*8 h(n), diag(n), offdiag(n), dxi, dxo
0.        23.         dxi = 1.0
0.931     24.         DO i = 1,n
0.100     25.           dxo = dxi
2.672     26.           dxi = 1.0/h(i+1)
1.311     27.           diag(i) = dxo + dxi
1.151     28.           offdiag(i) = -dxi
1.371     29.         ENDDO
0.        30.         END
0.        31.         SUBROUTINE initarray2(h, n)
              <Function: initarray2_>
          32.         REAL*8 h(n), diag(n), offdiag(n), dxi, dxo
0.460     33.         DO i = 1,n
1.671     34.           dxo = 1.0/h(i)
2.802     35.           dxi = 1.0/h(i+1)
1.011     36.           diag(i) = dxo + dxi
0.871     37.           offdiag(i) = -dxi
1.401     38.         ENDDO
0.        39.         END
```

```
f77 -g fdinit.f -o fdinit
collect -o fdinit.er ./fdinit
```

UltraSparc IIIi 1.2 GHz

```
f77 -fast -g fdinit.f -o fdinit
collect -o fdinit.er ./fdinit
```

*Timing comparison of **compiler optimized** kernels*
recurrent version: 0.841 sec
non-recurrent version: 0.841 sec

Sun Studio Analyzer [fdinit.er]

File View Timeline Help Find Text:

Functions | Callers-Callees | Source | Disassembly | Timeline | Experiments

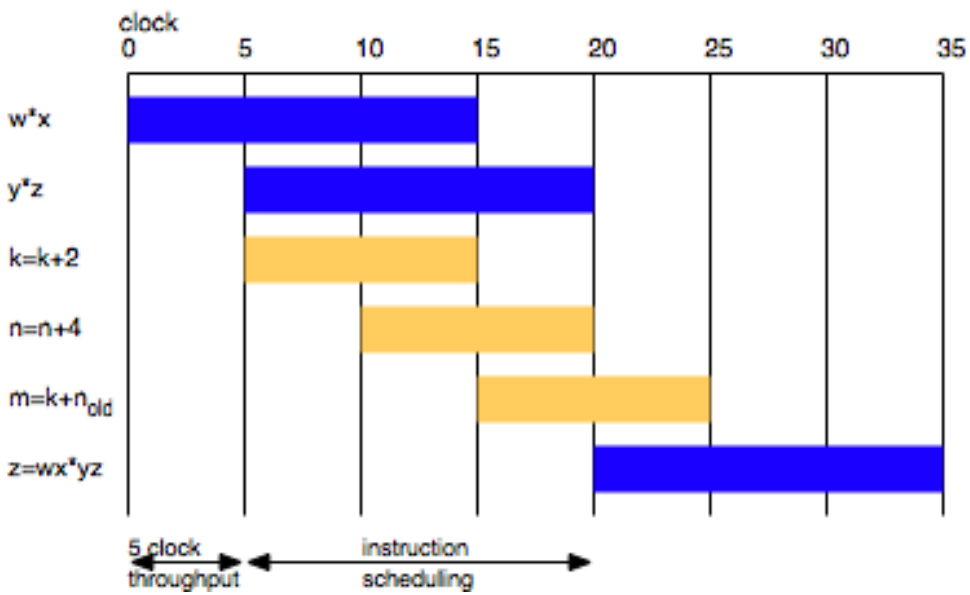
```
User CPU (sec.)   Source File: ./fdinit.f
                  Object File: ./fdinit
                  Load Object: <fdinit>
0.        21.         SUBROUTINE initarray1(h, n)
              <Function: initarray1_>
          22.         REAL*8 h(n), diag(n), offdiag(n), dxi, dxo
          23.         dxi = 1.0

          Source loop below has tag L3
          L3 scheduled with steady-state cycle count = 17
          L3 unrolled 4 times
          L3 has 1 loads, 1 stores, 2 prefetches, 1 FPadds, 0 FPmuls, and 1 FPdivs per iteration
          L3 has 0 int-loads, 0 int-stores, 4 alu-ops, 0 muls, 0 int-divs and 0 shifts per iteration
0.060     24.         DO i = 1,n
          25.           dxo = dxi
0.        26.           dxi = 1.0/h(i+1)
          27.           diag(i) = dxo + dxi
0.751     28.           offdiag(i) = -dxi
0.030     29.         ENDDO
0.        30.         END
0.        31.         SUBROUTINE initarray2(h, n)
              <Function: initarray2_>
          32.         REAL*8 h(n), diag(n), offdiag(n), dxi, dxo

          Source loop below has tag L4
          L4 scheduled with steady-state cycle count = 17
          L4 unrolled 4 times
          L4 has 1 loads, 1 stores, 2 prefetches, 1 FPadds, 0 FPmuls, and 1 FPdivs per iteration
          L4 has 0 int-loads, 0 int-stores, 4 alu-ops, 0 muls, 0 int-divs and 0 shifts per iteration
0.080     33.         DO i = 1,n
0.        34.           dxo = 1.0/h(i)
          35.           dxi = 1.0/h(i+1)
          36.           diag(i) = dxo + dxi
0.741     37.           offdiag(i) = -dxi
0.020     38.         ENDDO
0.        39.         END
```

Instruction Scheduling

```
z = (w*x)*(y*z);
k = k+2;
m = k+n
n = n+4;
```



- Instruction scheduling can hide data dependence latencies
- With *static scheduling* the compiler moves independent instructions up/down to fixed positions
- With *dynamic scheduling* the processor executes instructions *out of order* when they are ready
- The example shows an optimal instruction schedule, assuming there is one multiply and one add execution unit
  - Note that **`m=k+n`** uses the old value of **`n`** (WAR dependence)
  - Advanced processors remove WAR dependences from the execution pipeline using *register renaming*

Hardware Register Renaming

```
r0 = r1 + r2
r2 = r7 + r4
```

Sequential

```
r0 = r1 + r2; r2′= r7 + r4
…
r2 = r2′
```

Parallel

Pitfall: it does not help to manually remove WAR dependences in program source code by introducing extra scalar variables, because the compiler's register allocator reuses registers assigned to these variables and thereby reintroduces the WAR dependences at the register level

- Register renaming performed by a processor removes unnecessary WAR dependences caused by register reuse
  - Fixed number of registers are assigned by compiler to hold temporary values
- A processor's register file includes a set of *hidden registers*
- A hidden register is used in place of an actual register to eliminate a WAR hazard
- The WB stage stores the result in the destination register

Data Speculation

```
double *p, a[10];
…
*p = 0;
s += a[i];
```

```
r0 = 0
r1 = p
…
r2 = a+i
store r0 in M[r1]
load M[r2] into r3
```

```
r2 = a+i
adv load M[r2] into r3
…
r0 = 0
r1 = p
store r0 in M[r1]
check adv load address: reload r3
```

- A load should be initiated as far in advance as possible to hide memory latency
- When a store to address A1 is followed by a load from address A2 then there is a RAW dependence when A1=A2
  - A compiler assumes there is a RAW dependence if it cannot disprove A1=A2
- The *advanced load instruction* allows a process to ignore a potential RAW dependence and sort out the conflict at run time when the store address is known and is the same

Control Speculation

```
if (i<n)
  x = a[i]
```

```
  if (i≥n) jump to skip
  load a[i] into r0
skip:
  …
```

```
  speculative load a[i] into r0
  if (i≥n) jump to skip
    check spec load exceptions
skip:
  …
```

- *Control speculation* allows conditional instructions to be executed before the conditional branch in which the instruction occurs
  - Hide memory latencies
- A *speculative load instruction* performs a load operation
  - Enables loading early
  - Exceptions are ignored
- A *check operation* verifies that whether the load trigged an exception (e.g. bus error)
  - Reraise the exception

Hardware Branch Prediction

```
for (a=0; a<100; a++)
{ if (a % 2 == 0)
    do_even();
  else
    do_odd();
}
```

Simple branch pattern that is predicted correctly by processor

```
for (a=0; a<100; a++)
{ if (flip_coin() == HEADS)
    do_heads();
  else
    do_tails();
}
```

Random branch pattern that is difficult to predict

- *Branch prediction* is a architectural feature that enables a processor to fetch instructions of a target branch
  - When predicted correctly there is no branch penalty
  - When not predicted correctly, the penalty is typically >10 cycles
- Branch prediction uses a history mechanism per *branch instruction* by storing a compressed form of the past branching pattern

Improving Branch Prediction

- In a complicated branch test in C/C++ move the simplest to predict condition to the front of the conjunction

```
if (i == 0 && a[i] > 0)
```

  this example also has the added benefit of testing the more costly **`a[i]>0`** less frequently

- Rewrite conjunctions to logical expressions

```
if (t1==0 && t2==0 && t3==0)
⇒ if ( (t1|t2|t3) == 0 )
```

- Use max/min or arithmetic to avoid branches

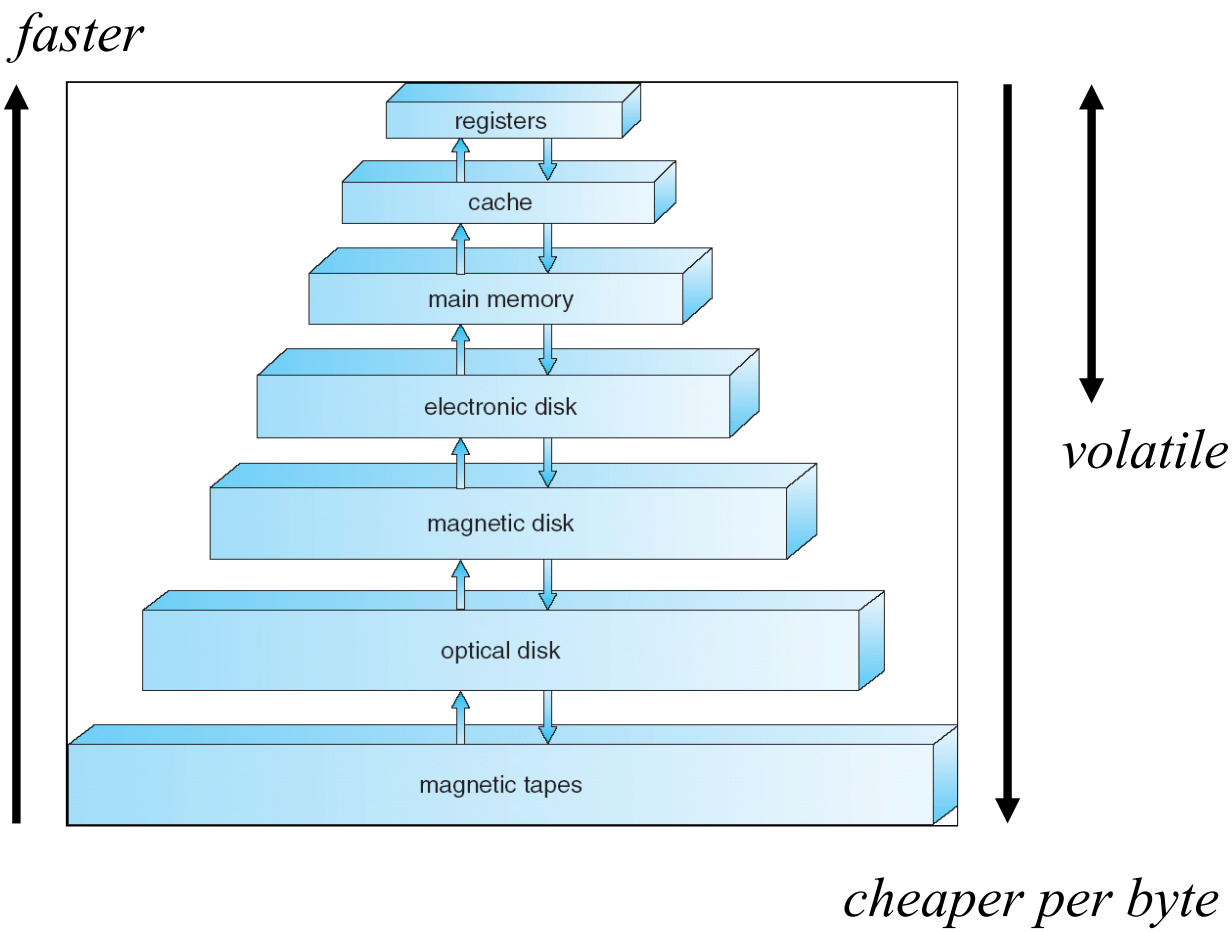
```
if (a >= 255) a = 255;
⇒ a = min(a, 255);
```

- Note that in C/C++ the *cond*`?`*then*`:`*else* operator and the **`&&`** and **`||`** operators result in branches!

Data Storage

- Memory hierarchy
- Performance of storage
- CPU and memory
- Virtual memory, TLB, and paging
- Cache

Memory Hierarchy



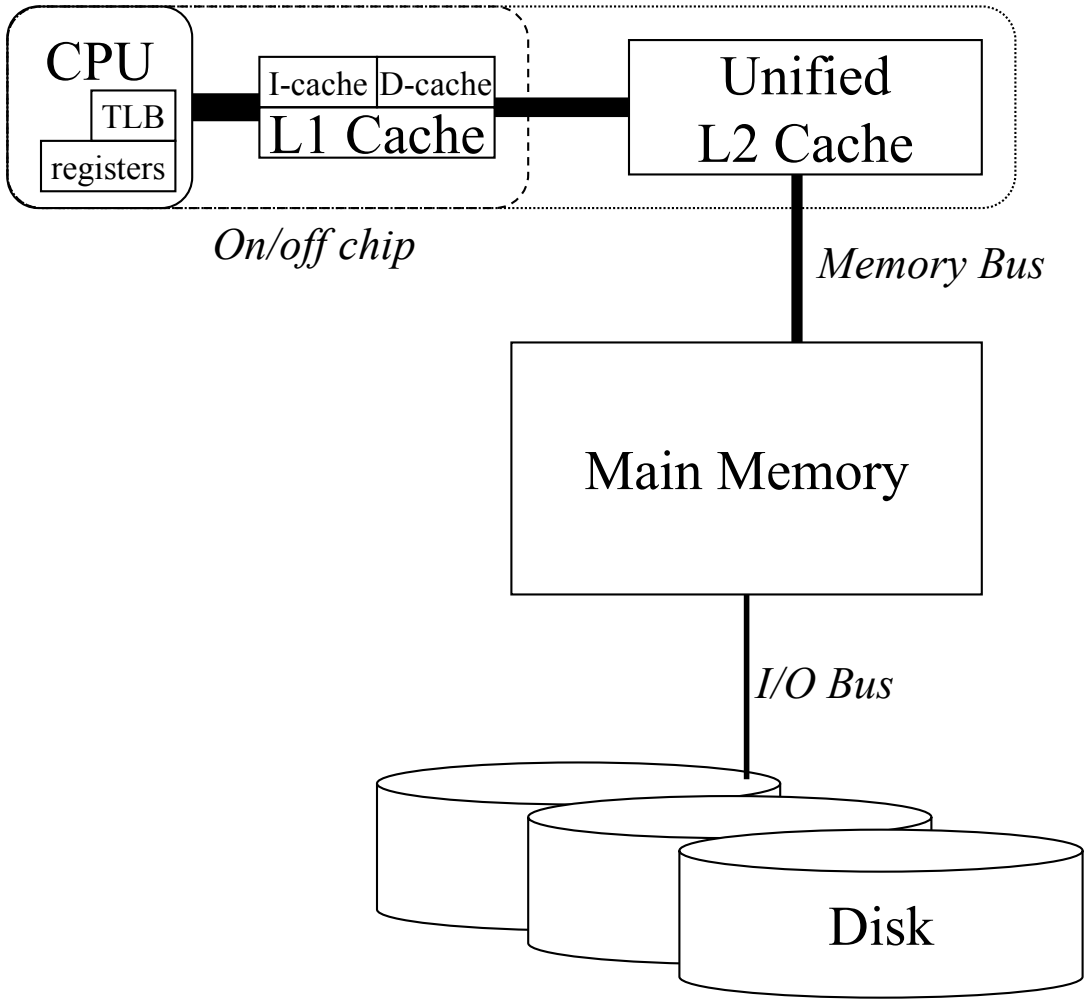
- Storage systems are organized in a hierarchy:
  - Speed
  - Cost
  - Volatility

Performance of Storage

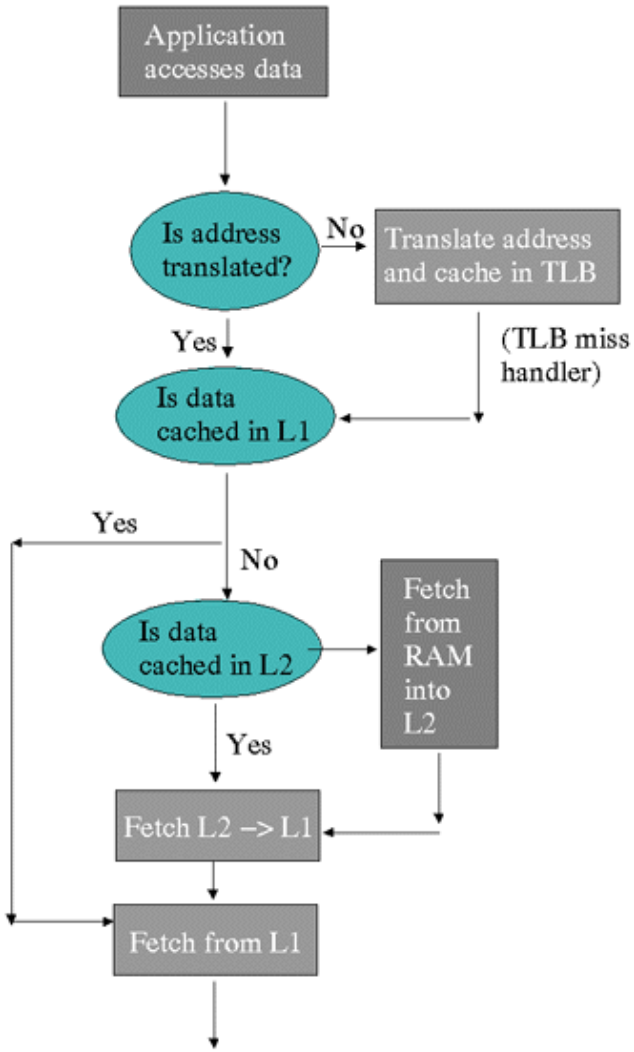
- Registers are fast, typically one clock cycle to access data
- Cache access takes tens of cycles
- Memory access takes hundreds of cycles
- Movement between levels of storage hierarchy can be explicit or implicit

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

CPU and Memory

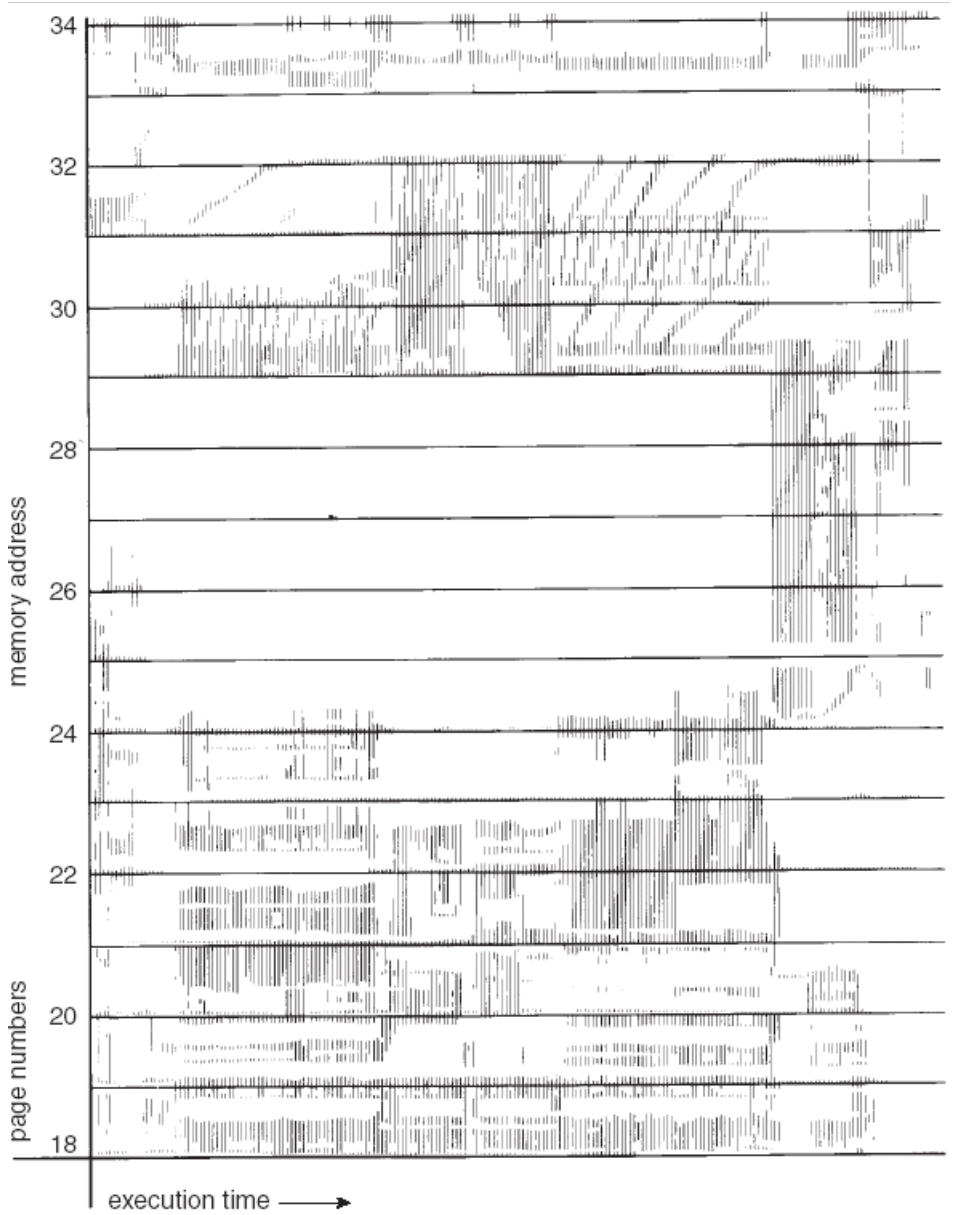


Memory Access



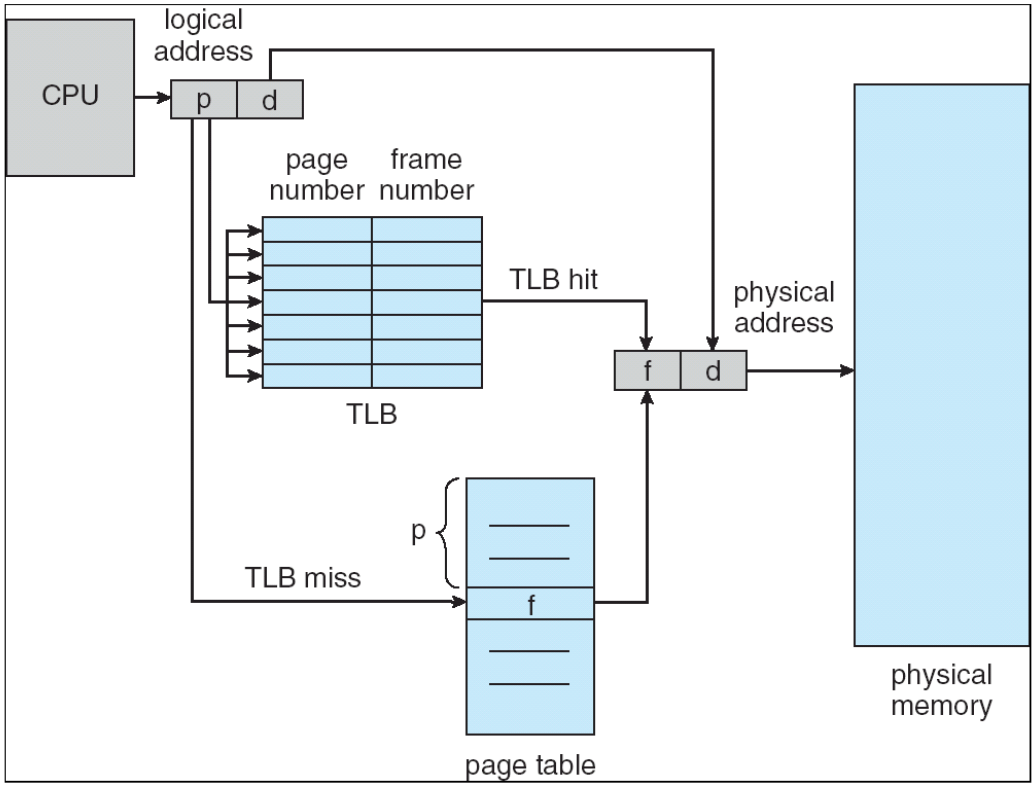
- A *logical address* is translated into a *physical address* in *virtual memory* using a *page table*
  - The *translation lookaside buffer* (TLB) is an efficient on-chip address translation cache
  - Memory is divided into *pages*
  - *Virtual memory systems* store pages in memory (RAM) and on disk
  - *Page fault*: page is fetched from disk
- L1 caches (on chip):
  - I-cache stores instructions
  - D-cache stores data
- L2 cache (E-cache, on/off chip)
  - Is typically *unified*: stores instructions and data

Locality in a Memory Reference Pattern

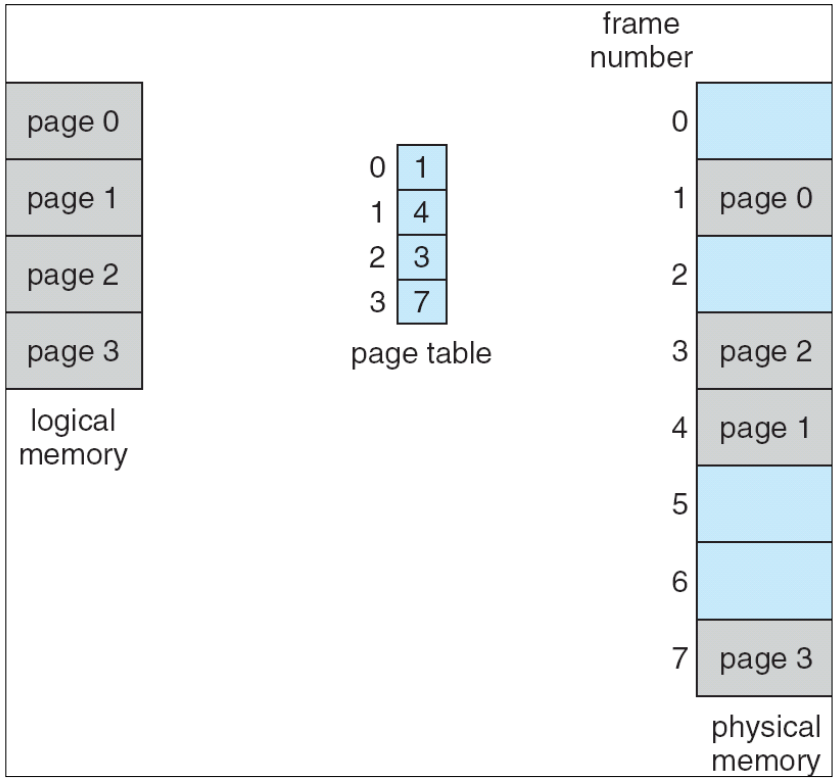


- The *working set model*
  - A process can be in physical memory if and only if all of the pages that it is currently using (the most recently used pages) can be in physical memory
  - *Page thrashing* occurs if pages are moved excessively between physical memory and disk
- Operating system picks and adjusts working set size
  - OS attempts to minimize page faults

Translation Lookaside Buffer



Logical address to physical address translation with TLB lookup to limit page table reads (page table is stored in physical memory)



Logical pages are mapped to physical pages in memory (typical page size is 4KB)

Finding TLB Misses

Performance Analyzer [mcf.er, ...]

File View Timeline Help

Find Text:

Functions | Callers-Callees | Source | Lines | Disassembly | PCs | DataLayout | DataObjects | MemObj | Timeline

User CPU (sec.)	E$ Stall Cycles (sec.)	DTLB Misses	E$ Refs	E$ Read Misses	Name
549.404	297.569	256 265 124	24 873 218 652	1 580 927 631	<Total>
280.706	184.224	225 606 768	9 539 061 477	984 629 538	refresh_potential
127.319	90.267	24 200 726	3 434 018 061	468 714 061	primal_bea_mpp
120.134	11.389	2 000 060	10 380 054 737	63 801 942	price_out_impl
6.154	5.533	500 015	80 000 387	31 000 930	flow_cost
4.933	0.011	0	856 005 682	0	sort_basket
3.733	2.711	300 009	105 000 554	16 300 489	dual_feasible
2.682	1.656	300 009	199 001 534	8 800 264	update_tree
1.861	0.933	700 021	108 000 616	3 700 111	primal_iminus
0.690	0.411	1 800 054	4 000 015	2 100 063	write_circulations
0.280	0.156	0	34 000 322	200 006	refresh_neighbour_lists
0.220	0.	600 018	11 000 082	100 003	primal_net_simplex
0.190	0.	0	25 000 140	0	number
0.180	0.	0	3 000 011	0	memset
0.090	0.	0	23 000 149	0	__doscan_u
0.060	0.033	0	1 000 005	200 006	primal_start_artificial
0.050	0.044	0	0	200 006	primal_feasible
0.040	0.	0	4 000 025	0	read_min
0.030	0.	0	0	0	sscanf
0.020	0.	0	0	0	vsscanf

Summary | Event | Legend | Leak

Data for Selected Object:

Name:	refresh_potential
PC Address:	2:0x000030E0
Size:	412
Source File:	mcf.er/archives/mcfutil.c
Object File:	src/mcf
Load Object:	<mcf>
Mangled Name:	
Aliases:	

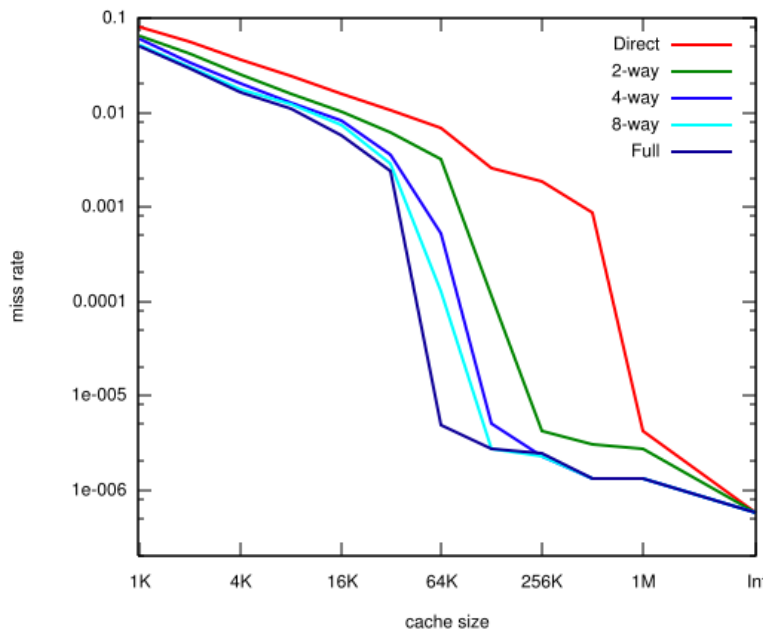
Process Times (sec.) / Counts

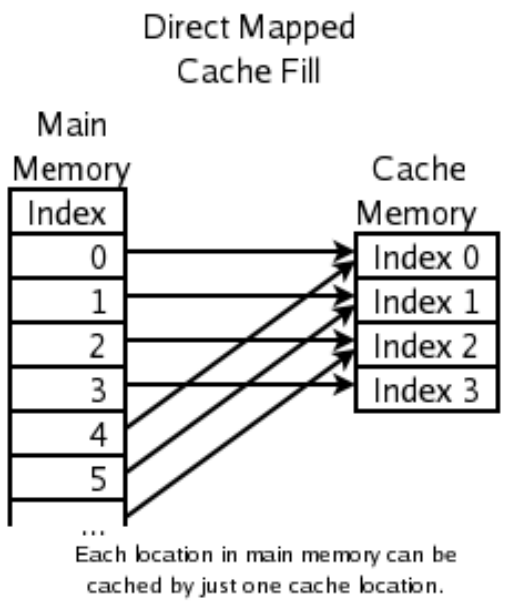
	Exclusive	Inc
User CPU:	280.706 (51.09%)	280.7
Wall:	282.488 (51.11%)	282.4
Total LWP:	282.488 (51.11%)	282.4
System CPU:	1.731 (56.17%)	1.7
Wait CPU:	0.050 (27.78%)	0.0
User Lock:	0. (0. %)	0.
Text Page Fault:	0. (0. %)	0.
Data Page Fault:	0. (0. %)	0.

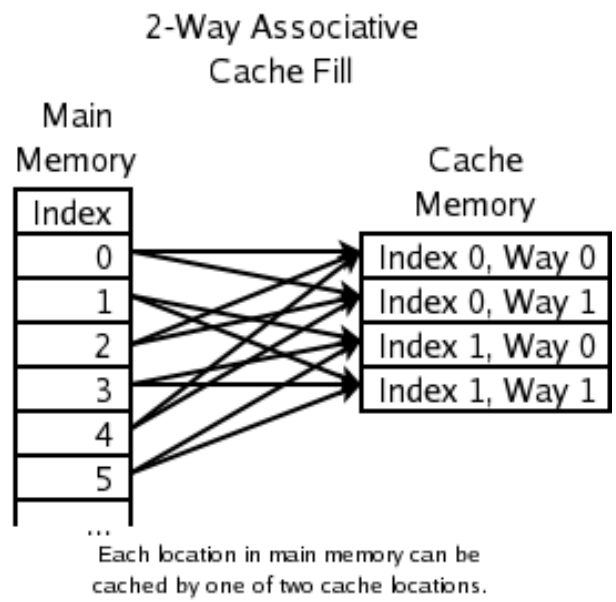
```
collect -o testprog.er -h DTLB_miss,on ./testprog
```

Caches

- *Direct mapped cache*: each location in main memory can be cached by just one cache location
- *N-way associative cache*: each location in main memory can be cached by one of *N* cache locations
- *Fully associative*: each location in main memory can be cached by any cache location
- Experiment shows SPEC CPU2000 benchmark cache miss rates

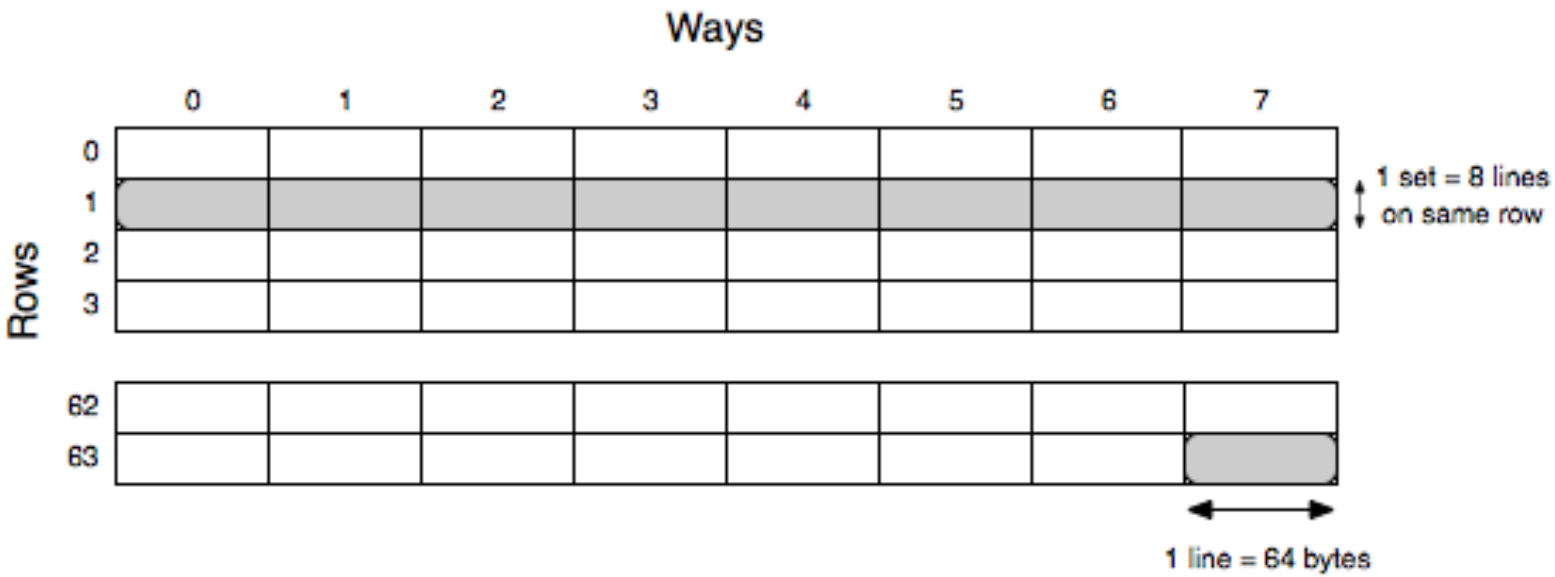






Cache Details

- An *N*-way associative cache has a *set* of *N cache lines* per *row*
- A cache line can be 8 to 512 bytes
  - Longer lines increase memory bandwidth performance, but space can be wasted when applications access data in a random order
- A typical 8-way L1 cache (on-chip) has 64 rows with 64 byte cache lines
  - Cache size = 64 rows x 8 ways x 64 bytes = 32768 bytes
- A typical 8-way L2 cache (on/off chip) has 1024 rows with 128 byte cache lines



Cache Misses

Compulsory miss:
reading **some_static_data[0]**

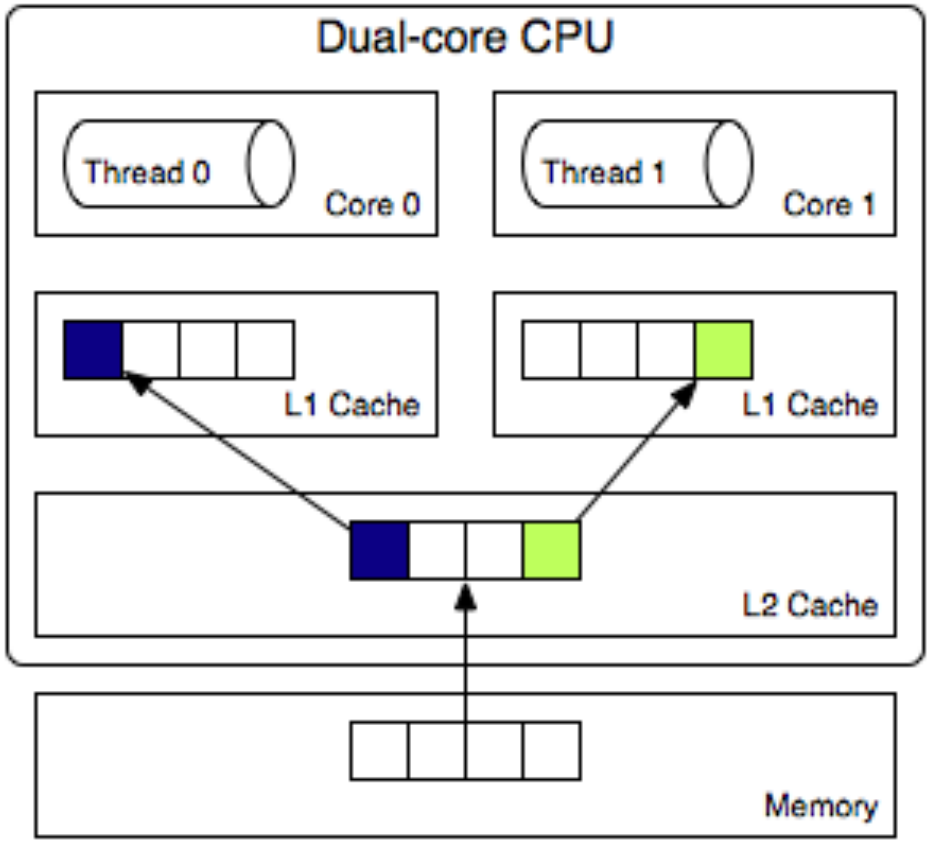
```
for (i = 0; i < 100000; i++)
  X[i] = some_static_data[i];
for (i = 0; i < 100000; i++)
  X[i] = X[i] + Y[i];
```

Capacity miss:
X[i] *no longer in cache*

Conflict miss:
X[i] *and* **Y[i]** *are mapped to the same cache line (e.g. when cache is direct mapped)*

- *Compulsory misses*
  - Caused by the first reference to a datum
  - Misses are effectively reduced by *prefetching*
- *Capacity misses*
  - Cache size is finite
  - Misses are reduced by limiting the *working set size* of the application
- *Conflict misses*
  - *Replacement misses* are caused by the choice of *victim* cache line to *evict* by the *replacement policy*
  - *Mapping misses* are caused by level of associativity

False Sharing



- On a multi-core processor each core has an L1 cache and shares the L2 cache with other cores
- *False sharing* occurs when two caches of processors (or cores) cache two different *non-shared data items* that reside on the *same cache line*
- *Cache coherency protocol* marks the cache line (on all cores) dirty to force reload
- To avoid false sharing:
  - Allocate non-shared data on different cache lines (using malloc)
  - Limit the use of global variables

Finding Cache Misses

Performance Analyzer [test.memory.1.er, ...]

File View Timeline Help

Find Text:

Functions | Callers-Callees | Source | Lines | Disassembly | PCs | Timeline | LeakList | Statistics | Experiments

User CPU (sec.)	CPU Cycles (sec.)	I$ Stall Cycles (sec.)	D$ and E$ Stall Cycles (sec.)	E$ Stall Cycles (sec.)	Name
41.539	31.950	0.066	13.671	11.499	<Total>
11.728	7.233	0.016	3.976	3.411	dgemv_gl_
9.947	5.433	0.012	4.045	3.525	dgemv_opt1_
8.236	7.900	0.009	0.002	0.001	load_arrays_
3.853	3.789	0.001	0.990	0.757	dgemv_g2_
1.481	1.456	0.002	0.945	0.741	dgemv_opt2_
1.111	1.089	0.002	0.	0.	__mt_WaitForWork_
1.091	1.022	0.001	0.949	0.807	dgemv_p1_ -- MP doall from line 12 [_$d1A12.dgemv_p1_]
1.061	1.022	0.001	0.951	0.801	dgemv_p2_ -- MP doall from line 31 [_$d1B31.dgemv_p2_]
0.971	0.933	0.001	0.896	0.729	dgemv_hi2_
0.951	0.922	0.001	0.896	0.718	dgemv_hi1_
0.380	0.378	0.	0.	0.	__mt_EndOfTask_Barrier_
0.360	0.356	0.	0.	0.001	barrier_ -- MP doall from line 20 [_$d1A20.barrier_]
0.360	0.356	0.	0.	0.001	barrier_ -- MP doall from line 45 [_$d1B45.barrier_]
0.010	0.	0.	0.001	0.	_private_close
0.	0.	0.	0.	0.	@plt
0.	0.	0.	0.001	0.	MAIN_
0.	0.	0.	0.	0.	__mt_MasterFunction_
0.	0.	0.	0.	0.	__mt_SlaveFunction_
0.	0.	0.	0.	0.	__mt_run_my_job_

Summary | Event | Legend | Leak

Data for Selected Object:

Name:	dgemv_gl_
PC Address:	2:0x00008538
Size:	1276
Source File:	/home/martyi/workarea/demos/cachetest
Object File:	/home/martyi/workarea/demos/cachetest
Load Object:	<cachetest>
Mangled Name:	
Aliases:	

Process Times (sec.) / Counts

	Exclusive	
User CPU:	11.728 (28.2%)	11
Wall:	11.958 (29.0%)	11
Total LWP:	11.958 (27.2%)	11
System CPU:	0.210 (17.4%)	0
Wait CPU:	0.020 (33.3%)	0
User Lock:	0. (0. %)	0
Text Page Fault:	0. (0. %)	0
Data Page Fault:	0. (0. %)	0
Other Wait:	0. (0. %)	0

```
collect -o test.memory.1.er -S off -p on -h ecstall,on,cycles,on ./cachetest -g memory.erg
collect -o test.memory.2.er -S off -p on -h icstall,on,dcstall,on ./cachetest -g memory.erg
```

Further Reading

- [HPC] pages 7-56
- Optional: [OPT] pages 51-141