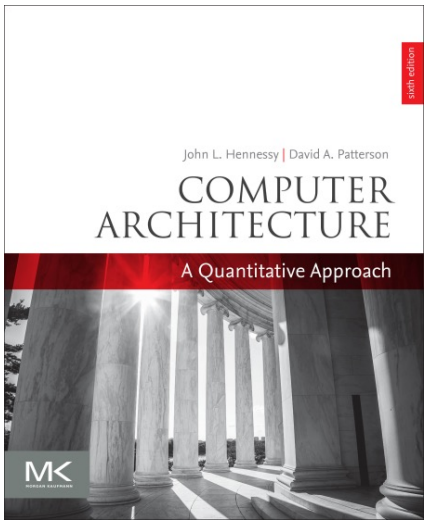
CESE 4085 Modern Computer Architecture

Lectures 7 & 8: Data-Level Parallelism



Chapter 4

Data-Level Parallelism in Vector, SIMD, and GPU Architectures

Lecture Overview

Lecture 7:

- Vector Processors
- SIMD Extensions
- Multi-threading (Chapter 3)

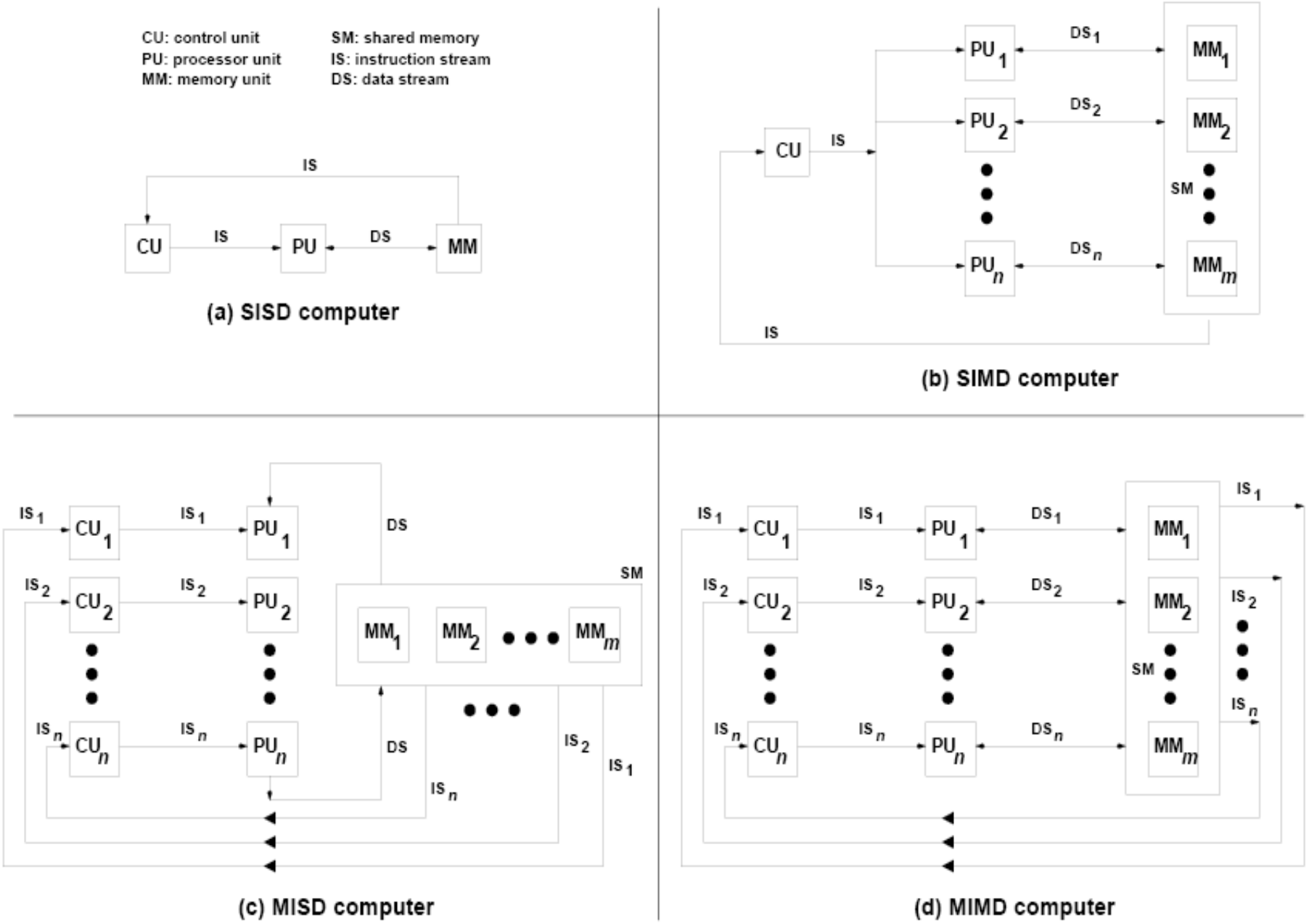
Combining topics of 6th and earlier editions of CA book

Lecture 8: (ambitious)

- GPUs
- Loop-level parallelism (*skipped, but is exam material*)
- Thread-Level Parallelism – Multi-processors (Chapter 5)
- Memory Coherence (Chapter 5)
  - Snooping vs. Directory Protocols

Flynn's Taxomony

- SISD (Single Instruction, Single Data)
  - Uniprocessors
- SIMD (Single Instruction, Multiple Data)
  - Exploits *data-level parallelism*
  - Vector architectures also belong to this class
    - Multimedia extensions (MMX, SSE, VIS, AltiVec, …)
  - Examples: Illiac-IV, CM-2, MasPar MP-1/2
- MISD (Multiple Instruction, Single Data)
  - ???
- MIMD (Multiple Instruction, Multiple Data)
  - Examples: Sun Enterprise 5000, Cray T3D/T3E, SGI Origin
    - exploits *thread-level parallelism*; flexible
  - Most widely used


CU: control unit
PU: processor unit
MM: memory unit
SM: shared memory
IS: instruction stream
DS: data stream
IS
CU
IS
PU
DS
MM
(a) SISD computer
CU
IS
PU 1
PU 2
PU n
DS 1
DS 2
DS n
MM 1
MM 2
MM m
SM
IS
(b) SIMD computer
IS 1
CU 1
PU 1
DS
SM
MM 1
MM 2
MM m
IS 2
CU 2
PU 2
IS n
CU n
PU n
DS
IS 2
IS 1
IS n
(c) MISD computer
IS 1
CU 1
PU 1
DS 1
MM 1
IS 1
IS 2
CU 2
PU 2
DS 2
MM 2
IS 2
SM
IS n
CU n
PU n
DS n
MM m
IS n
(d) MIMD computer

Introduction

SIMD architectures can exploit significant data-level parallelism for:

- Matrix-oriented scientific computing
- Media-oriented image and sound processors

SIMD is more energy efficient than MIMD

- Only needs to fetch one instruction per data operation
- Makes SIMD attractive for personal mobile devices

SIMD allows programmer to continue to think sequentially

SIMD Parallelism

Vector architectures

SIMD extensions

Graphics Processor Units (GPUs)

For x86 processors:

- Expect two additional cores per chip per year
- SIMD width to double every four years
- Potential speedup from SIMD to be twice that from MIMD!

Intermezzo – processing vectors

Question – what takes a lot time when processing vectors/arrays in a scalar processor?

For example:

```
for(i=0; i<N, i++)
  {Y(i)=a*X(i)+Y(i)}
```

Question: what do we need to do every iteration?

Answer:

-

Intermezzo – processing vectors

Question – what takes a lot time when processing vectors/arrays in a scalar processor?

For example:

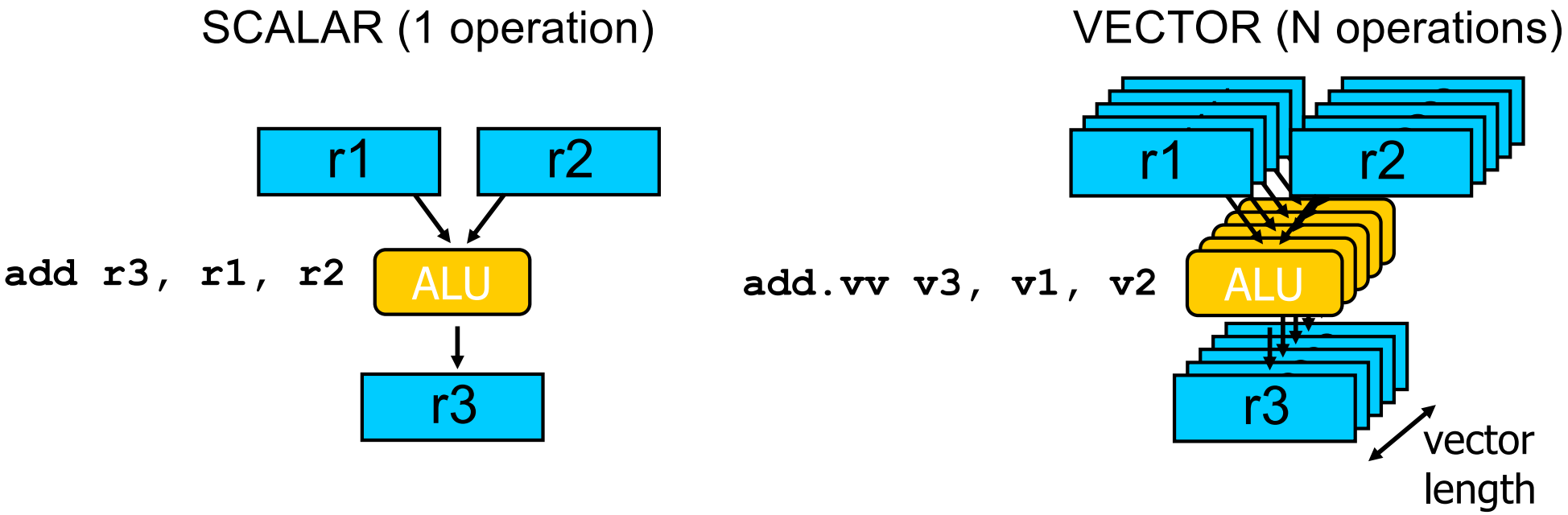
```
for(i=0; i<N, i++)
  {Y(i)=a*X(i)+Y(i)}
```

Question: what do we need to do every iteration?

Answer: check i (compare), increment i, branch

(Also: load/store per loop iteration resulting in potential (cache) misses, thus long memory access latencies)

Vector processors



- if N is the vector length, not necessarily N ALUs
- number of ALU's = lanes

Vector processors

- Single vector instruction specifies lots of work
  - equivalent to executing an entire loop
  - no control hazards for the loop branches
  - fewer instructions to fetch and decode (*also: sm. program size*)
- Computation of each result in the vector is independent of the computation of other results in the same vector
  - deep pipeline without data hazards; high clock rate
  - HW has to check for data hazards only between vector instructions (once per vector, not per vector element)
- Access memory with known pattern
  - elements are all adjacent in memory => highly interleaved memory banks provides high bandwidth
  - access is initiated for entire vector => high memory latency is amortised (no data caches are needed)

Vector processors

- Vector operations: arithmetic (add, sub, mul, div), memory accesses, effective address calculations
- Multiple vector instructions can be in progress at the same time => more parallelism
- The application has to be as such …
  - Regular loops
  - Large scientific and engineering applications (car crash simulations, whether forecasting, …)
  - Multimedia applications
- .. to benefit from using a vector processor

Basic Vector Architectures

- Vector processor: ordinary pipelined scalar unit + vector unit
- Types of vector processors
  - Memory-memory processors: all vector operations are memory-to-memory
  - Vector-register processors: all vector operations except load and store are among the vector registers

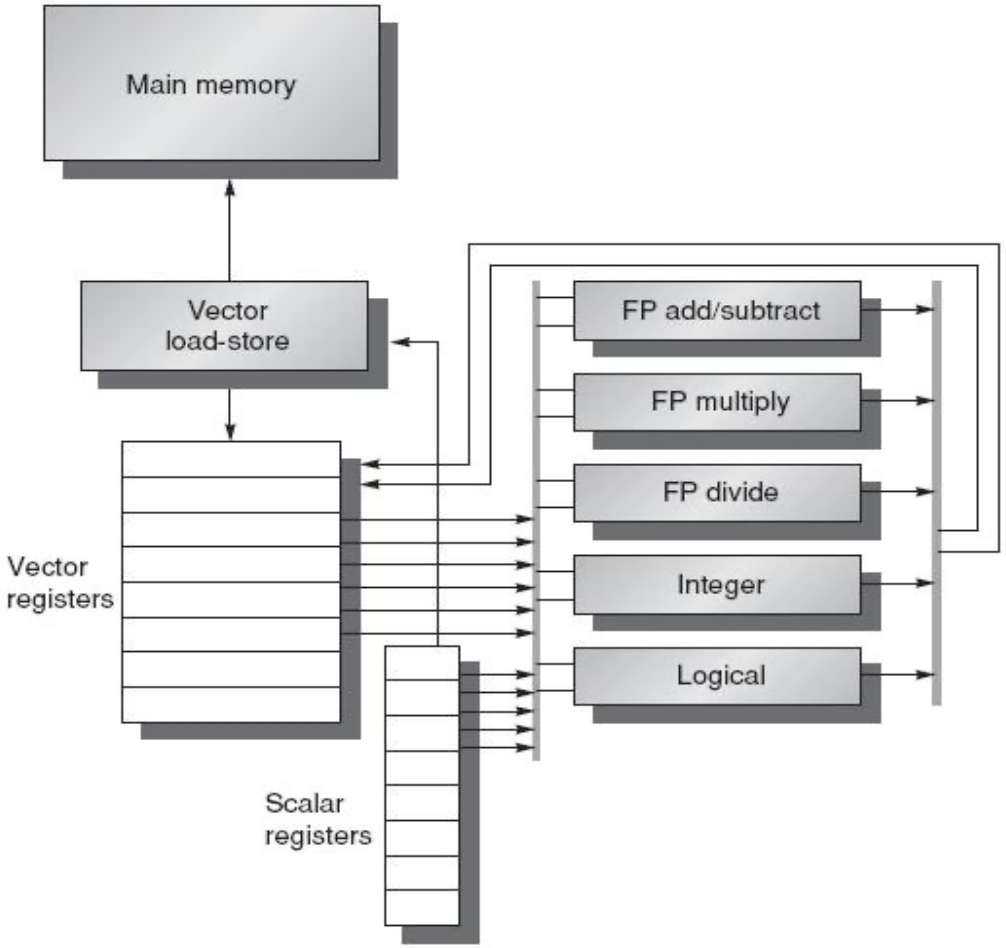
Vector-register processor

- Vector Registers: each vector register is a fixed length bank holding a single vector
  - has at least 2 read and 1 write ports (overlap operations)
  - typically 8-32 vector registers, each holding 64-128 elements (words)
  - VMIPS: 8 vector registers, each holding 64 elements (16 Rd ports, 8 Wr ports)
- Vector Functional Units (FUs): multiple resources, fully pipelined, start new operation every clock
  - typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift;
  - may have multiple of same unit
  - VMIPS: 5 FUs (FP add/sub, FP mul, FP div, FP integer, FP logical)

Vector-register processor

- Vector Load-Store Units (LSUs)
  - fully pipelined unit to load or store a vector; may have multiple LSUs
    - VMIPS: 1 VLSU, bandwidth is 1 word per cycle after initial delay
- Scalar registers
  - single element for FP scalar or address
  - VMIPS: 32 GPR, 32 FPRs they are read out and latched at one input of the FUs
- Cross-bar to connect FUs, LSUs, registers
  - cross-bar to connect Rd/Wr ports and FUs

VMIPS: Basic Structure



- 8 x 64-element vector registers
- 5 FUs – each fully pipelined
- Load/store unit – fully pipelined
- Scalar registers

VMIPS Vector Instructions

Instruction	Operands	Function
ADDV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2, i.e., R1+i × R2.
SVWS	(R1,R2),V1	Store V1 from address at R1 with stride in R2, i.e., R1+i × R2.
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--V.D S--VS.D	V1,V2 V1,F0	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in the vector-mask register and store count in R1.

Vector Execution Time

- *Execution time* is a function of vector length, data dependencies, struct. hazards)
- *Initiation rate*: rate at which a FU consumes vector elements
  - number of lanes = the number of parallel pipelines used to execute operations within each vector instruction; up to 8 (e.g. Cray X1)
  - the time for a single vector instruction = init.rate x vect.len.
- *Convoy*
  - set of vector instructions that can begin execution in same clock (no struct. or data hazards); assumption: convoys do not overlap in time (no forwarding).
- *Chime*: approx. time to execute a convoy
  - e.g. m convoys take m chimes; if each vector length is n, then they take approx. m x n clock cycles (ignores overhead; good approximation for long vectors)

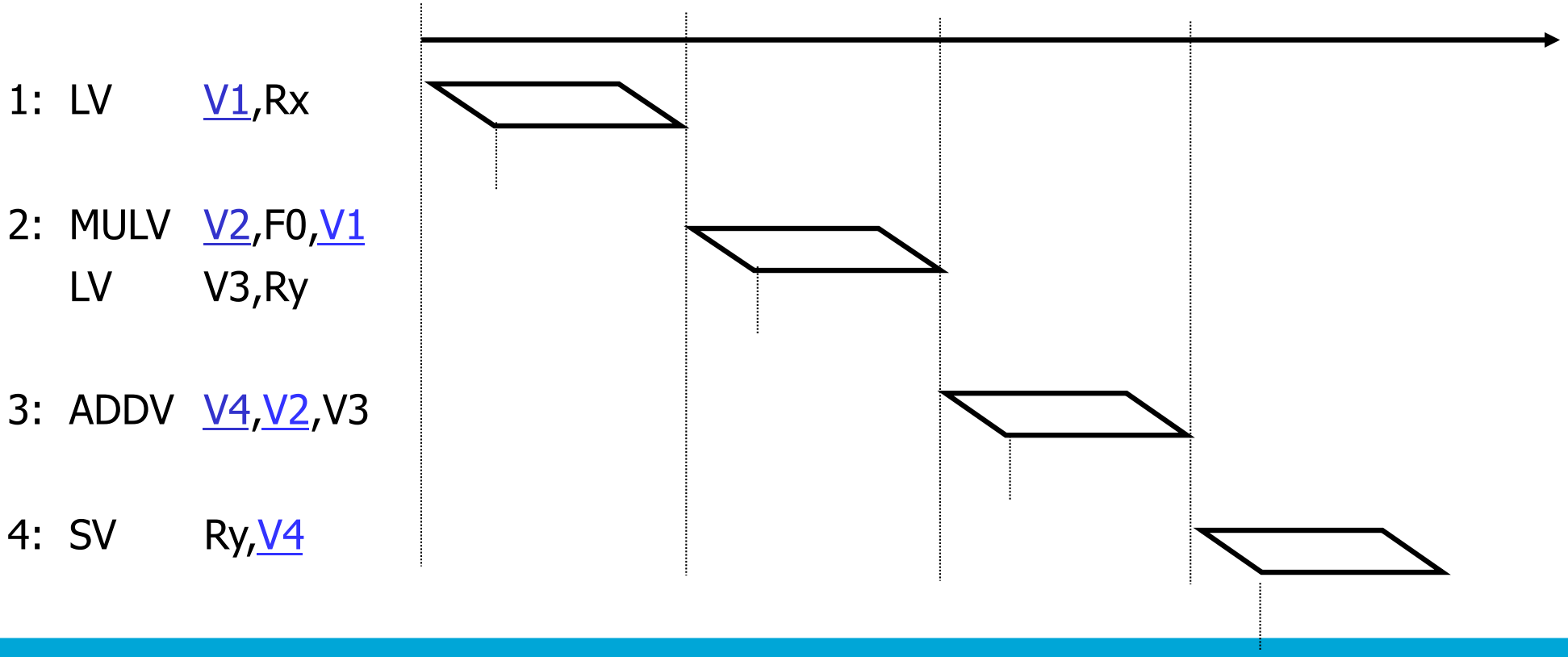
Vector Execution Time

- Example:
  - 4 convoys, 1 lane,
  - vector length=64 elements; 1 element/cycle (1 lane pipelined execution unit)
  - a convoy has to finish before another starts
    - 4 x 64 256 clocks

```
1: LV      V1,Rx  ;load vector X
2: MULVS.D V2, V1,F0  ;vector-scalar mult.
   LV          V3,Ry  ;load vector Y
3: ADDV.D  V4,V2,V3  ;add
4: SV       Ry,V4 ;store the result
```

Start-up time & execution time

- Instructions in a convoys may start at 1 cycle distance (pipelined instruction issue) – small overhead
- important source of overhead: s*tart-up time* = pipeline latency time (depth of FU pipeline, e.g. 12 for MULV, 6 for ADDV);



Vector Load/Store Units & Memories

- Start-up overheads usually longer for LSUs
- Memory system must sustain (number of lanes x word) /clock cycle
- Many Vector Procs. use banks (vs. simple interleaving):
  - support multiple loads/stores per cycle
    => multiple banks & address banks independently
  - support non-sequential accesses
- Note: Number of memory banks > memory latency to avoid stalls
  - m banks => m words per memory latency l clocks
  - if m < l, then gap in memory pipeline:
  - may have 1024 banks in SRAM

Variable Vector Length

- What to do when vector length is not exactly VL (64)?

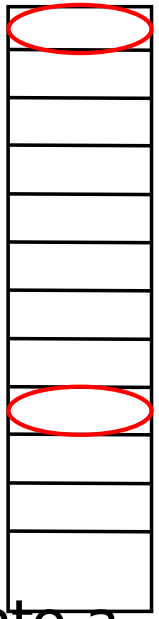
```
for(i=0; i<N, i++)
  {Y(i)=a*X(i)+Y(i)}
```

- N can be unknown at compile time
- Vector-Length Register (VLR): controls the length of any vector operation, including a vector load or store (cannot be > the length of vector registers)
- What if N > Max. Vector Length (MVL)? ⇒ Strip mining
  - 1st loop does short piece (N mod MVL), rest VL = MVL

Vector Stride

- Adjacent elements not sequential in memory (e.g. matrix multiplication)

```
for(i=0; i<100; i++)
  for(j=0; j<100; j++) {
    x(i,j)=0.0;
    for(k=0; k<100; k++)
      x(i,j)=y(i,j)+B(i,k)*C(k,j);
  }
```



- Matrix C accesses are not adjacent (800 bytes between)
- Stride: distance separating elements that are to be merged into a single vector ⇒ LVWS (load vector with stride) instruction
- Strides can cause bank conflicts (e.g., stride=32 and 16 banks)
  - e.g. x[i, stride], x[i, 2 x stride], x[i, 3 x stride], … map in the same bank

Vector Opt.1: Chaining (fwd. for vector regs.)

- MULV.D V1,V2,V3
  ADDV.D V4,V1,V5 ; separate convoy
- Chaining: if vector register (V1) is not a single entity but a group of individual registers ⇒ pipeline forwarding of individual elements of a vector
- Flexible chaining: allow vector to chain to any other active vector operation ⇒ simultaneous access to same register
  - more read/write ports
  - organize the registers in individual banks (⇒ simultaneous access to different banks)
- As long as enough HW, increases convoy size

Vector Opt.2: Conditional Execution

```
do i = 1, 64
   if (A(i) <> 0) then
      A(i) = A(i) - B(i)
   endif
```

- Loop is vectorizable, if only no "*if*"s (no control dependency)
- Conditional execution: turns a control dependency in a data dependency
- Vector-mask control:
  - vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1
  - vector-mask register loaded from vector test
- Some VP use vector mask only to disable the storing of the result (the operation still executes)

Vector Opt.3: Sparse Matrices

- Sparse matrix: elements of a vector are usually stored in some compacted form and then accessed indirectly, e.g.:

```
do 100 i = 1, n
   A(K(i))=A(K(i))+C(M(i))
```

Note: vectors A and C potentially stored using different compaction technique, therefore, K and M to unpack.

- Mechanism to support sparse matrices: scatter-gather operations
  - to support moving between a dense representation (0's not included) and normal matrix representation (0's included)
- Gather (LVI) fetches the vector whose elements are by:
  - address = a base address + offsets given in an index vector
  - sort of register indirect addressing; get elements in dense form
- Elements are operated on in dense form,
- (if needed) the sparse vector can be stored in expanded form: scatter store (SVI), using the same index vector

Summary

- fine-grained vs. coarse-grained multi-threading
- SMT – fine-grained based on OoO superscalar
  - single-thread performance should not be penalized
  - shares processor resources, except: PC, register renaming, ROB
  - reduces the impact of branch miss-prediction
- Vector processing
  - No control&data dependencies between elements of a vector
  - Vector instructions access memory with known pattern
  - Reduces branches and branch problems in pipelines
  - A single vector instruction does lots of work (loop)
  - Components of a vector processor: vector registers, functional units, load/store, crossbar....
  - VP optimisation: chaining, conditional execution, sparse matrices

Vector Architectures

Basic idea:

- Read sets of data elements into "vector registers"
- Operate on those registers
- Disperse the results back into memory

Registers are controlled by compiler

- Used to hide memory latency
- Leverage memory bandwidth

VMIPS

Example architecture: RV64V

- Loosely based on Cray-1
- 32 62-bit vector registers
  - Register file has 16 read ports and 8 write ports
- Vector functional units
  - Fully pipelined
  - Data and control hazards are detected
- Vector load-store unit
  - Fully pipelined
  - One word per clock cycle after initial latency
- Scalar registers
  - 31 general-purpose registers
  - 32 floating-point registers

VMIPS Instructions

.vv: two vector operands

.vs and .sv: vector and scalar operands

LV/SV: vector load and vector store from address

Example: DAXPY

```
    vsetdcfg   4*FP64# Enable 4 DP FP vregs
    fld        f0,a          # Load scalar a
    vld        v0,x5         # Load vector X
    vmul       v1,v0,f0      # Vector-scalar mult
    vld        v2,x6         # Load vector Y
    vadd       v3,v1,v2      # Vector-vector add
    vst        v3,x6         # Store the sum
    vdisable                 # Disable vector regs
```

8 instructions, 258 for RV64V (scalar code)

Vector Execution Time

Execution time depends on three factors:

- Length of operand vectors
- Structural hazards
- Data dependencies

RV64V functional units consume one element per clock cycle

- Execution time is approximately the vector length

Convey

- Set of vector instructions that could potentially execute together

Chimes

Sequences with read-after-write dependency hazards placed in same convey via *chaining*

Chaining

Allows a vector operation to start as soon as the individual elements of its vector source operand become available

Chime

Unit of time to execute one convey

m conveys executes in *m* chimes for vector length *n*

For vector length of *n*, requires *m* x *n* clock cycles

Example

```
vld     v0,x5       # Load vector X
vmul    v1,v0,f0    # Vector-scalar multiply
vld     v2,x6       # Load vector Y
vadd    v3,v1,v2    # Vector-vector add
vst     v3,x6       # Store the sum
```

Convoys:

```
1       vld     vmul
2       vld     vadd
3       vst
```

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires 32 x 3 = 96 clock cycles

Challenges

Start up time

- Latency of vector functional unit
- Assume the same as Cray-1
  - Floating-point add => 6 clock cycles
  - Floating-point multiply => 7 clock cycles
  - Floating-point divide => 20 clock cycles
  - Vector load => 12 clock cycles

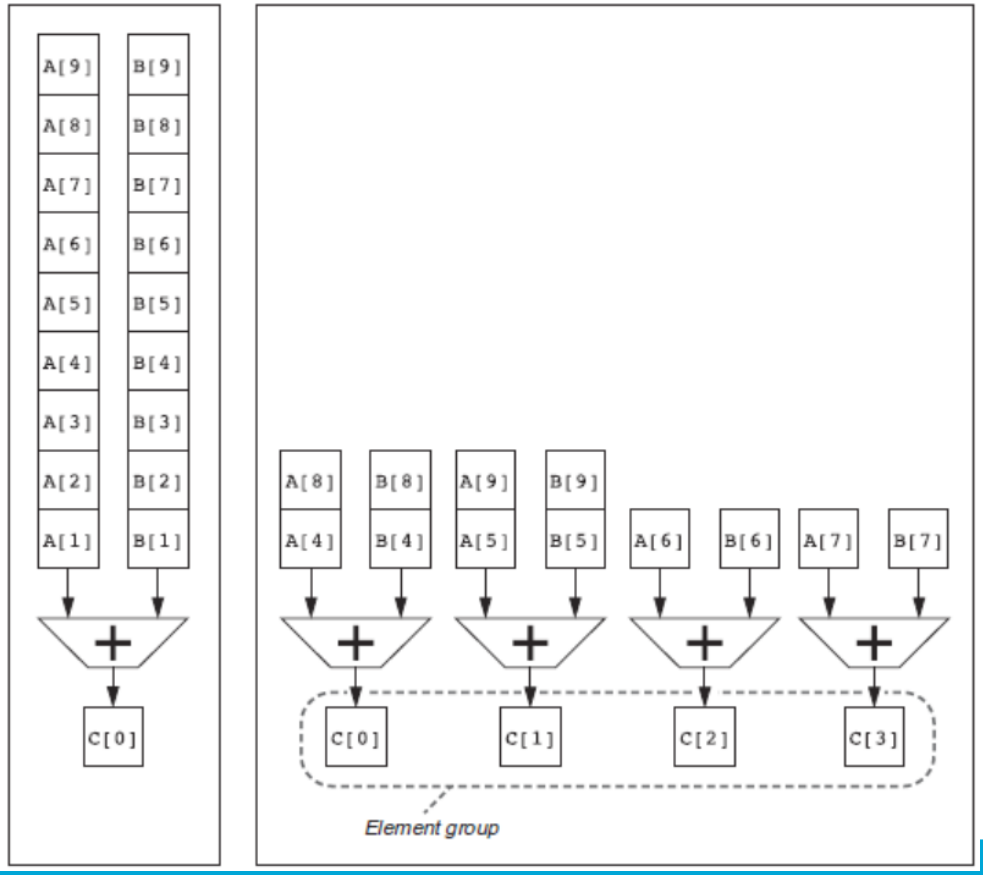
Improvements:

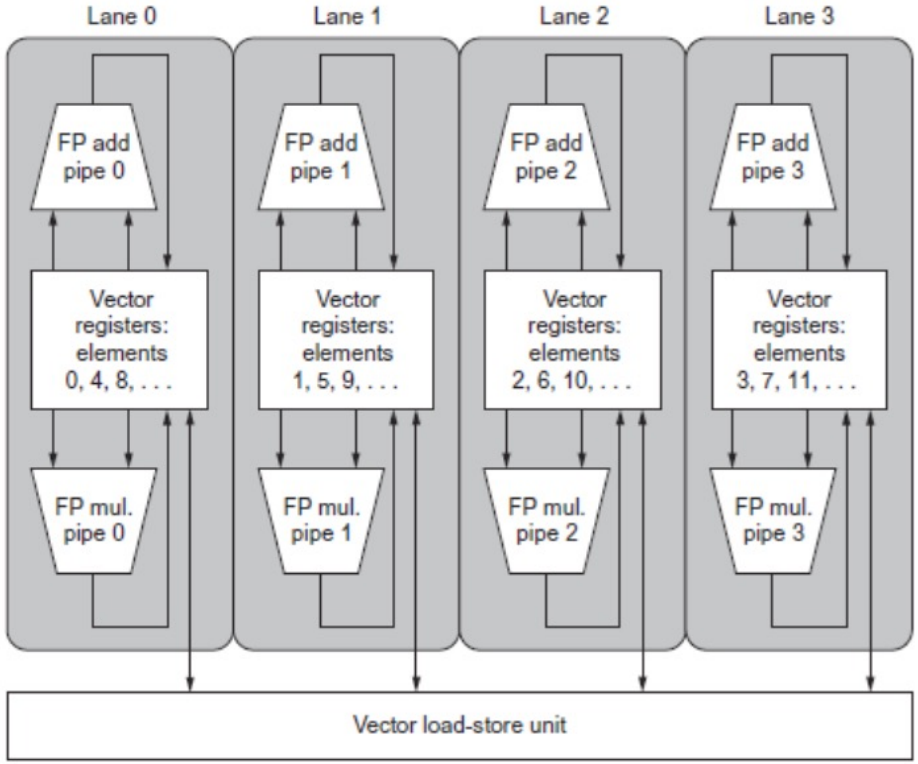
- > 1 element per clock cycle
- Non-64 wide vectors
- IF statements in vector code
- Memory system optimizations to support vector processors
- Multiple dimensional matrices
- Sparse matrices
- Programming a vector computer

Multiple Lanes

Element *n* of vector register *A* is "hardwired" to element *n* of vector register *B*

- Allows for multiple hardware lanes





Vector Length Register

```
for (i=0; i <n; i=i+1) Y[i] = a * X[i] + Y[i];
      vsetdcfg 2 DP FP   # Enable 2 64b Fl.Pt. registers
      fld f0,a           # Load scalar a
     loop: setvl t0,a0     # vl = t0 = min(mvl,n)
      vld v0,x5          # Load vector X
      slli t1,t0,3       # t1 = vl * 8 (in bytes)
      add x5,x5,t1       # Increment pointer to X by vl*8
      vmul v0,v0,f0      # Vector-scalar mult
      vld v1,x6          # Load vector Y
      vadd v1,v0,v1      # Vector-vector add
      sub a0,a0,t0       # n -= vl (t0)
      vst v1,x6          # Store the sum into Y
      add x6,x6,t1       # Increment pointer to Y by vl*8
      bnez a0,loop       # Repeat if n != 0
      vdisable           # Disable vector regs}
```

Vector Mask Registers

Consider:

```
for (i = 0; i < 64; i=i+1)
    if (X[i] != 0)
        X[i] = X[i] – Y[i];
```

Use predicate register to “disable” elements:

```
vsetdcfg    2*FP64      # Enable 2 64b FP vector regs
vsetpcfgi   1       # Enable 1 predicate register
vld             v0,x5 # Load vector X into v0
vld             v1,x6 # Load vector Y into v1
fmv.d.x     f0,x0 # Put (FP) zero into f0
vpne        p0,v0,f0    # Set p0(i) to 1 if v0(i)!=f0
vsub        v0,v0,v1    # Subtract under vector mask
vst             v0,x5 # Store the result in X
vdisable            # Disable vector registers
vpdisable           # Disable predicate registers
```

Memory Banks

Memory system must be designed to support high bandwidth for vector loads and stores

Spread accesses across multiple banks

- Control bank addresses independently
- Load or store non sequential words (need independent bank addressing)
- Support multiple vector processors sharing the same memory

Example:

- 32 processors, each generating 4 loads and 2 stores/cycle
- Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
- How many memory banks needed?
  - 32x(4+2)x15/2.167 = ~1330 banks

Stride

Consider:

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
        A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

Must vectorize multiplication of rows of B with columns of D

Use *non-unit stride*

Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:

#banks / LCM(stride,#banks) < bank busy time

Scatter-Gather

Consider:

```
for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

Use index vector:

```
vsetdcfg   4*FP64     # 4 64b FP vector registers
vld      v0, x7       # Load K[]
vldx       v1, x5, v0 # Load A[K[]]
vld      v2, x28      # Load M[]
vldi       v3, x6, v2 # Load C[M[]]
vadd       v1, v1, v3 # Add them
vstx       v1, x5, v0 # Store A[K[]]
vdisable              # Disable vector registers
```

Programming Vec. Architectures

Compilers can provide feedback to programmers

Programmers can provide hints to compiler

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

SIMD Extensions

Media applications operate on data types narrower than the native word size

- Example: disconnect carry chains to "partition" adder

Limitations, compared to vector instructions:

- Number of data operands encoded into op code
- No sophisticated addressing modes (strided, scatter-gather)
- No mask registers

SIMD Implementations

Implementations:

- Intel MMX (1996)
  - Eight 8-bit integer ops or four 16-bit integer ops
- Streaming SIMD Extensions (SSE) (1999)
  - Eight 16-bit integer ops
  - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
- Advanced Vector Extensions (2010)
  - Four 64-bit integer/fp ops
- AVX-512 (2017)
  - Eight 64-bit integer/fp ops
- Operands must be consecutive and aligned memory locations

Example SIMD Code

Example DXPY:

```
        fld       f0,a      # Load scalar a
        splat.4Df0,f0       # Make 4 copies of a
        addi      x28,x5,#256     # Last address to load
Loop: fld.4D      f1,0(x5)      # Load X[i] ... X[i+3]
        fmul.4D f1,f1,f0      # a x X[i] ... a x X[i+3]
        fld.4D    f2,0(x6)      # Load Y[i] ... Y[i+3]
        fadd.4D f2,f2,f1      # a x X[i]+Y[i]...
                              # a x X[i+3]+Y[i+3]
        fsd.4D    f2,0(x6)      # Store Y[i]... Y[i+3]
        addi      x5,x5,#32   # Increment index to X
        addi      x6,x6,#32   # Increment index to Y
        bne       x28,x5,Loop # Check if done
```

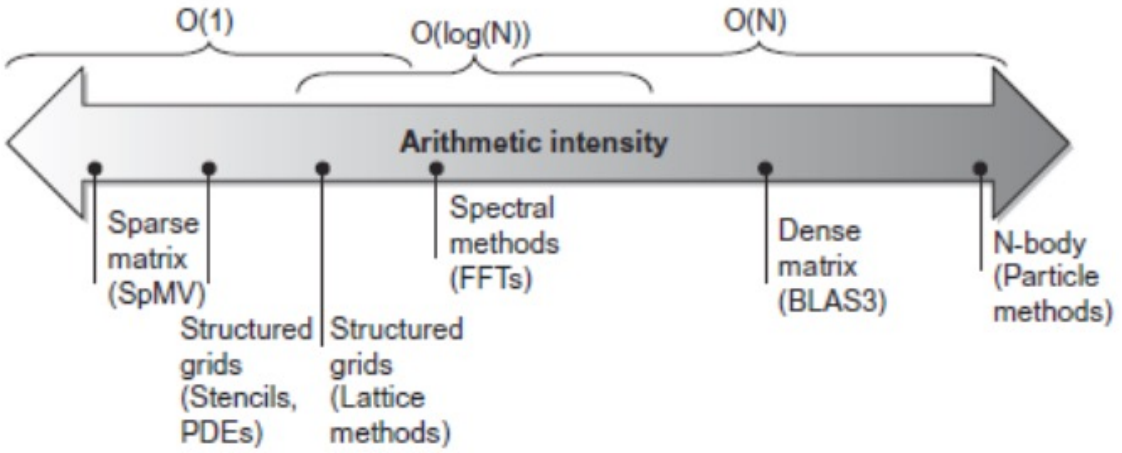
Roofline Performance Model

Basic idea:

- Plot peak floating-point throughput as a function of arithmetic intensity
- Ties together floating-point performance and memory performance for a target machine

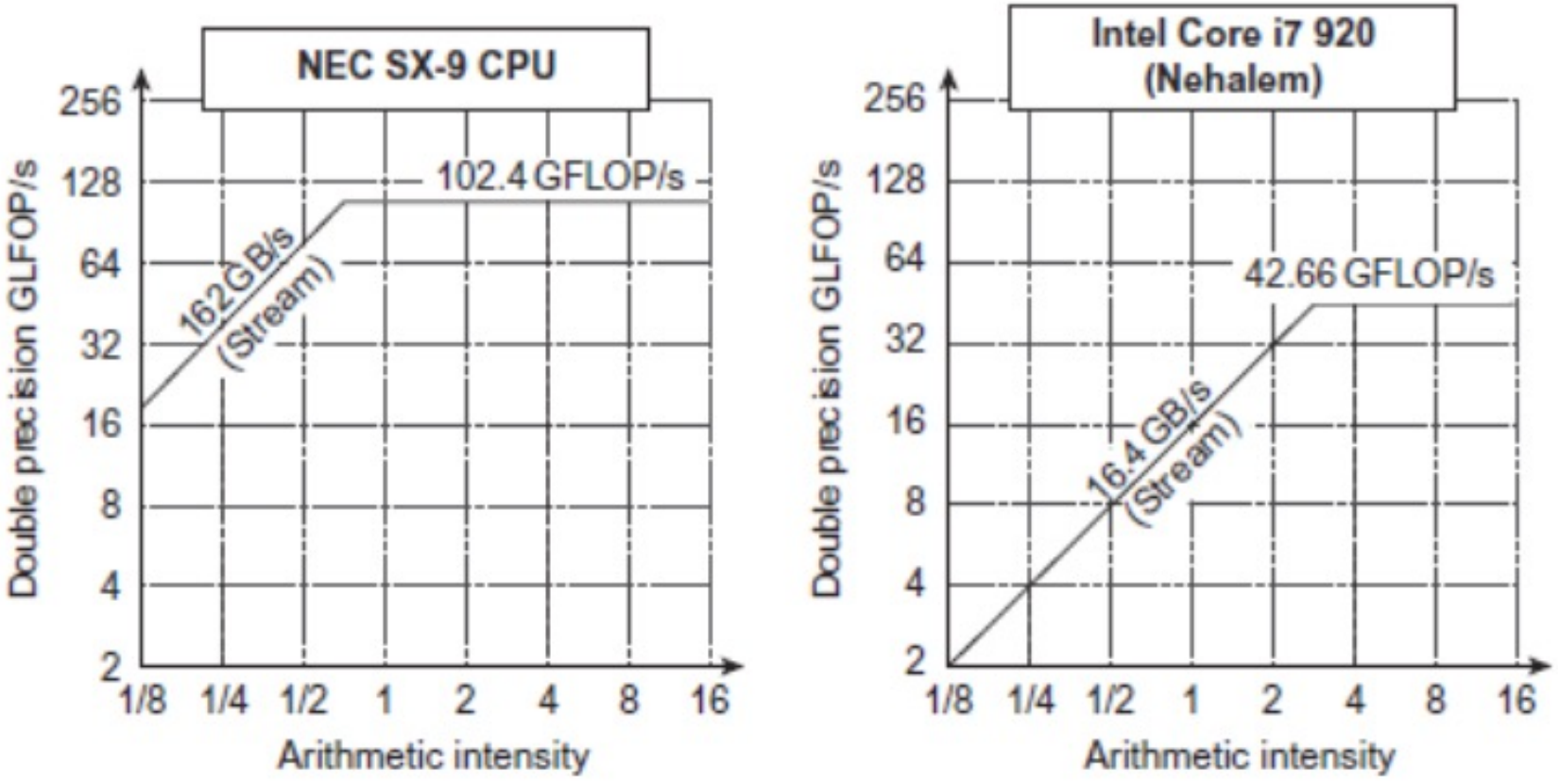
Arithmetic intensity

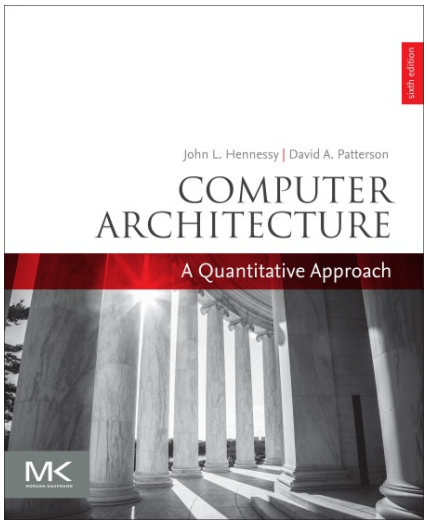
- Floating-point operations per byte read



Examples

Attainable GFLOPs/sec = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)





Chapter 5

Thread-Level Parallelism

Introduction

Thread-Level parallelism

- Have multiple program counters
- Uses MIMD model
- Targeted for tightly-coupled shared-memory multiprocessors

For *n* processors, need *n* threads

Amount of computation assigned to each thread = grain size

- Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

Start – multi-threading slides

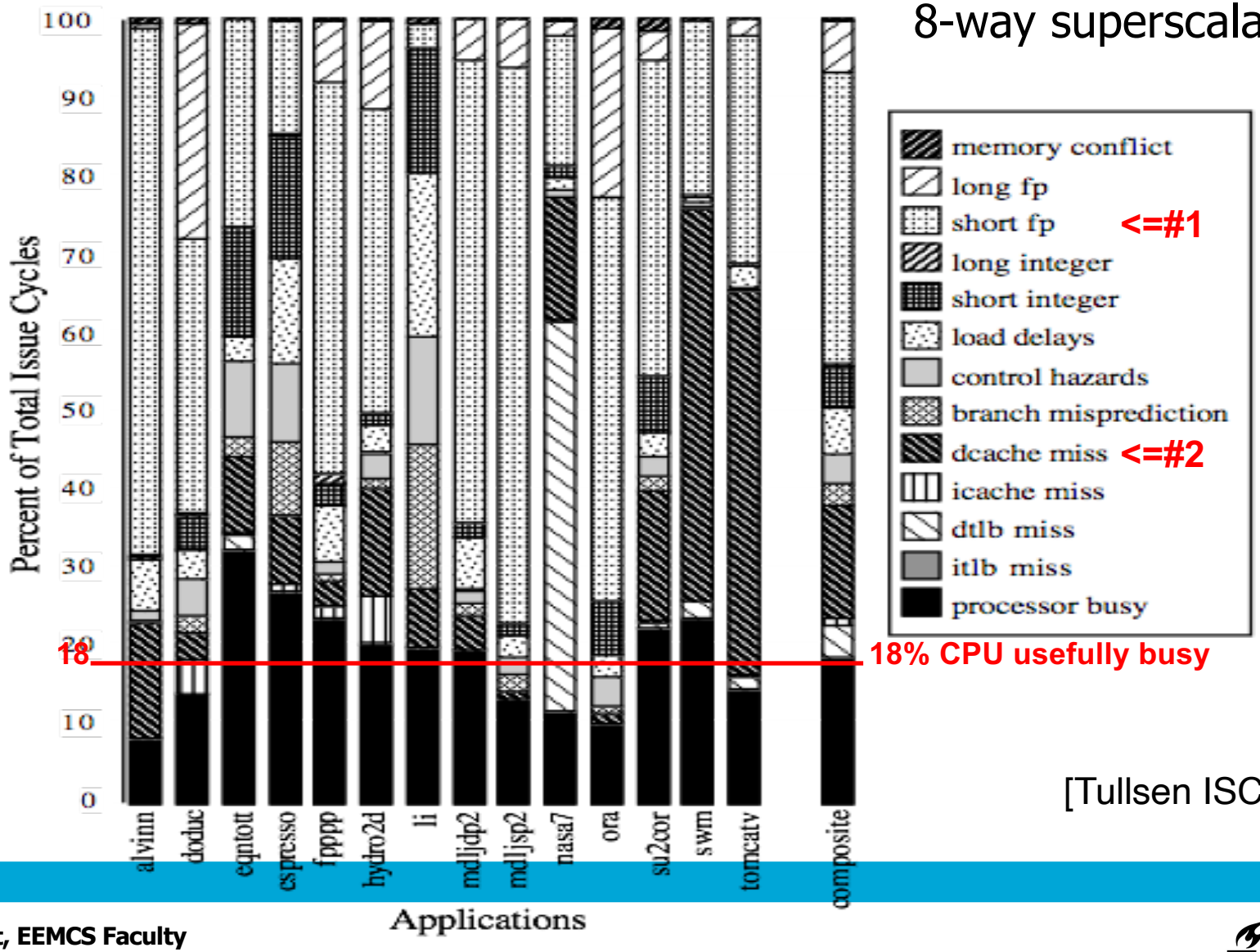
There are multithreading slides – check for usefulness!

Start – Multithreading slides

Trends in microarchitecture

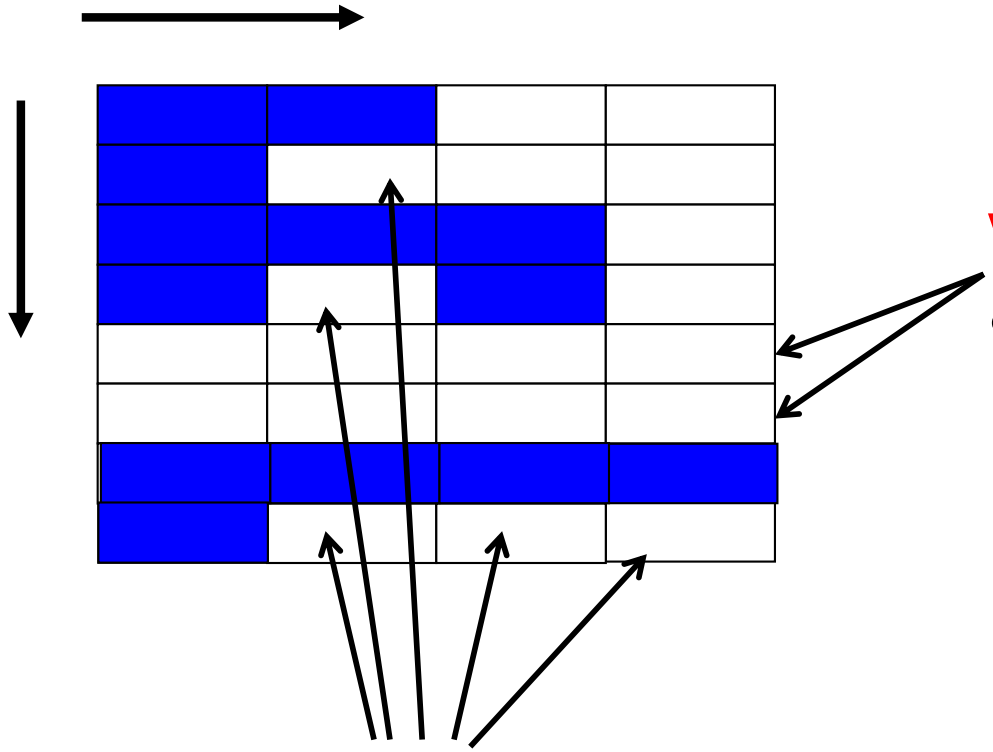
- Higher clock speeds
  - To achieve high clock frequency make pipeline deeper (super-pipelining)
  - Events that disrupt pipeline (branch mispredictions, cache misses, TLB misses, etc) become very expensive in terms of lost clock cycles
- ILP
  - Extract parallelism in a single program
  - Superscalar processors - multiple execution units working in parallel
  - Challenge to find enough instructions that can be executed concurrently (limit: instruction dependencies)
  - Out-of-order execution => instructions are sent to execution units based on instruction dependencies rather than program order

ILP limitations: most applications stall 80% or more of time during "execution"



[Tullsen ISCA95]

Resources waste



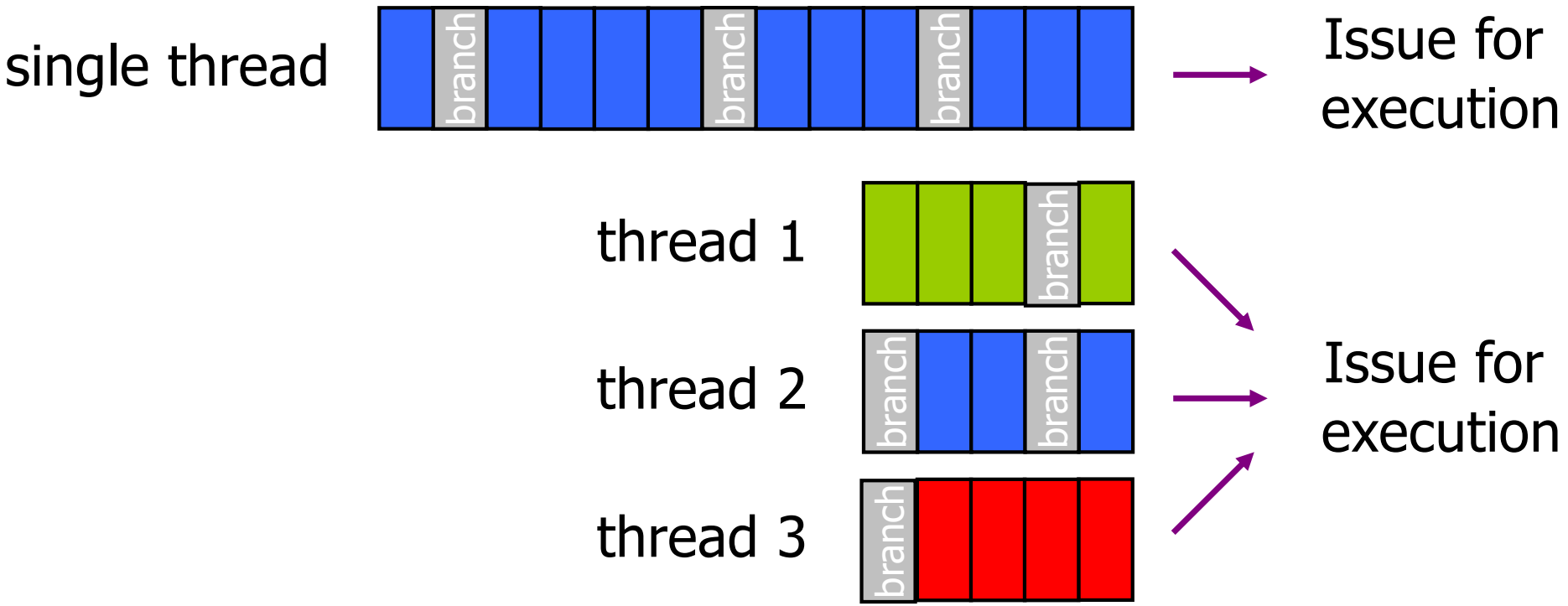
Vertical waste:

- due to stalls in the execution flow

Horizontal waste:

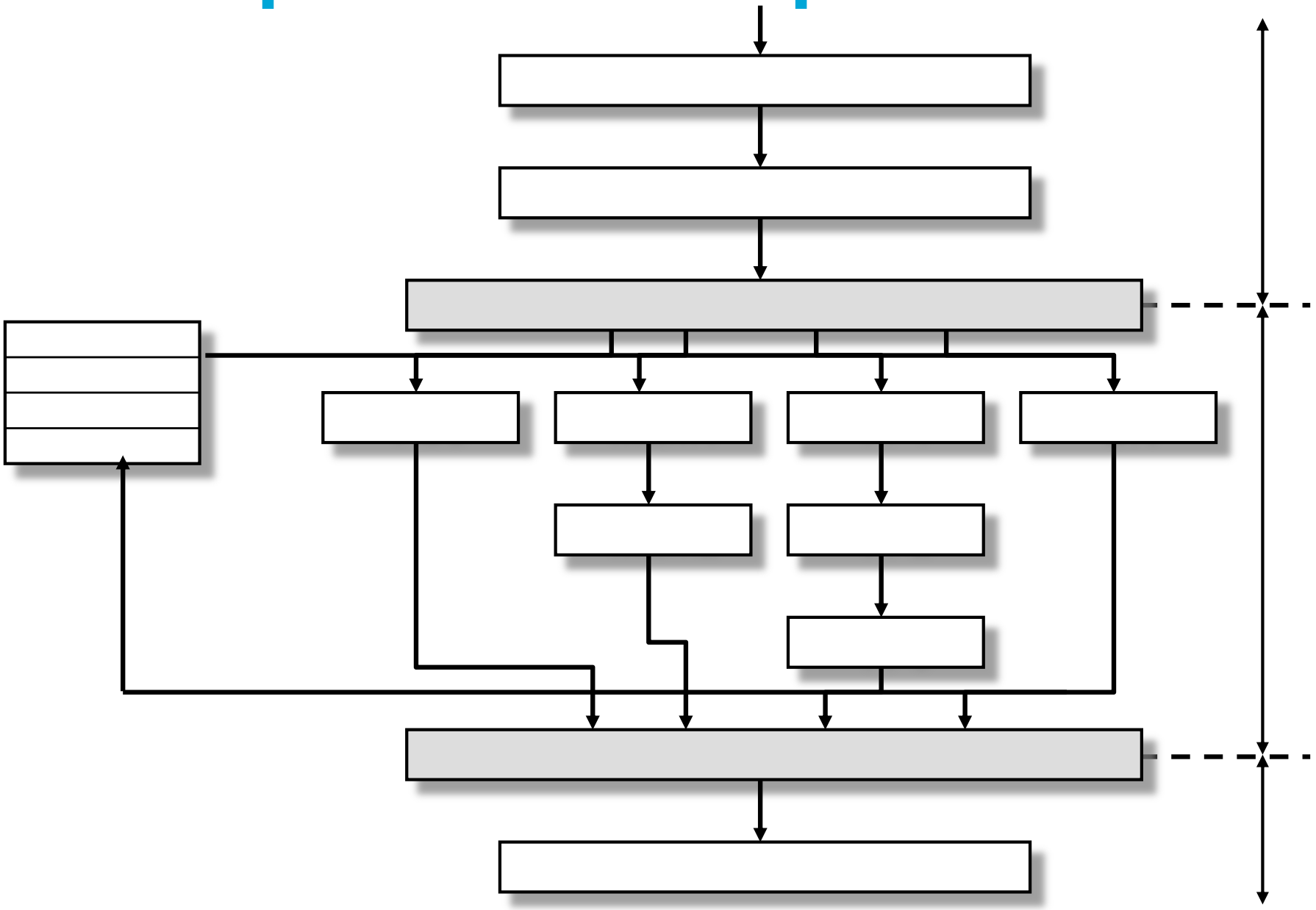
- due to low level ILP (not able to "use" all FUs)

Multi-threading idea



- Instead of enlarging the **depth** of the instruction window considered for issue (more speculation, lowering confidence),
- Enlarge its **"width"** by fetching from multiple threads
- Reduces the impact of Branch Prediction!

OoO superscalar – recap



Exploiting thread-level parallelism

- Multi-processors
  - Each with a full set of architectural resources

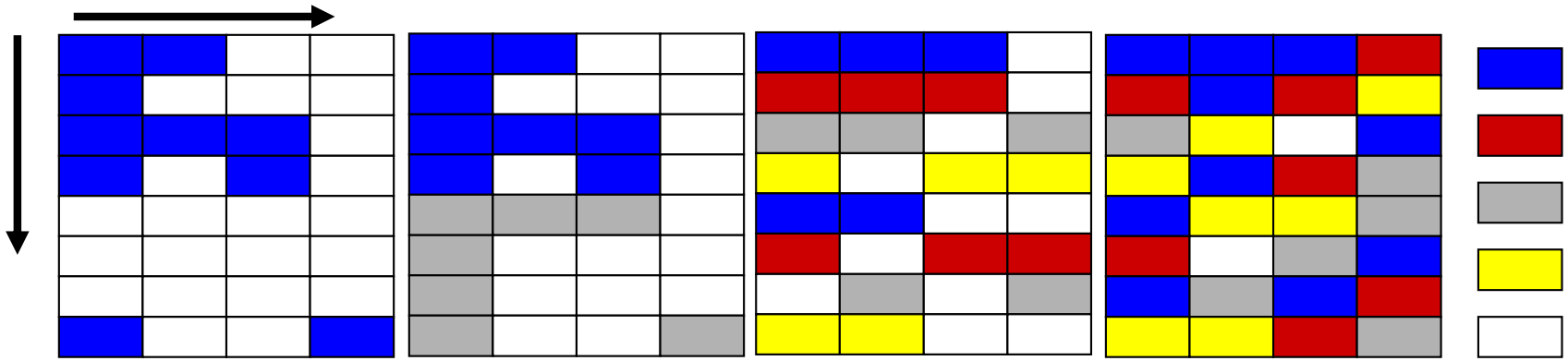
- Multiple threads – concurrently on the same processor Why?
  - Inexpensive – one CPU, no external interconnects
  - No remote or coherence misses (more capacity misses)
  - Threads can share resources $\Rightarrow$ we can increase threads without a corresponding linear increase in area

Multi-threading

When to switch (to **which**) thread?

- Cycle-by-cycle interleaving: fine-grained multi-threading
  - Processor switches between software threads after a predefined time slice (a cycle)
  - Round-robin among threads, skip when thread is stalled.
- Blocking interleaving: course-grained multi-threading
  - processor switches to another thread when:
    - a long latency operation stalls the current thread (L2 miss),
    - max number of cycles/thread exceeded
    - scheduling difficulties encountered
- Still, some execution slots are wasted
- Fundamental: single-thread performance is not penalized

Multi-threading

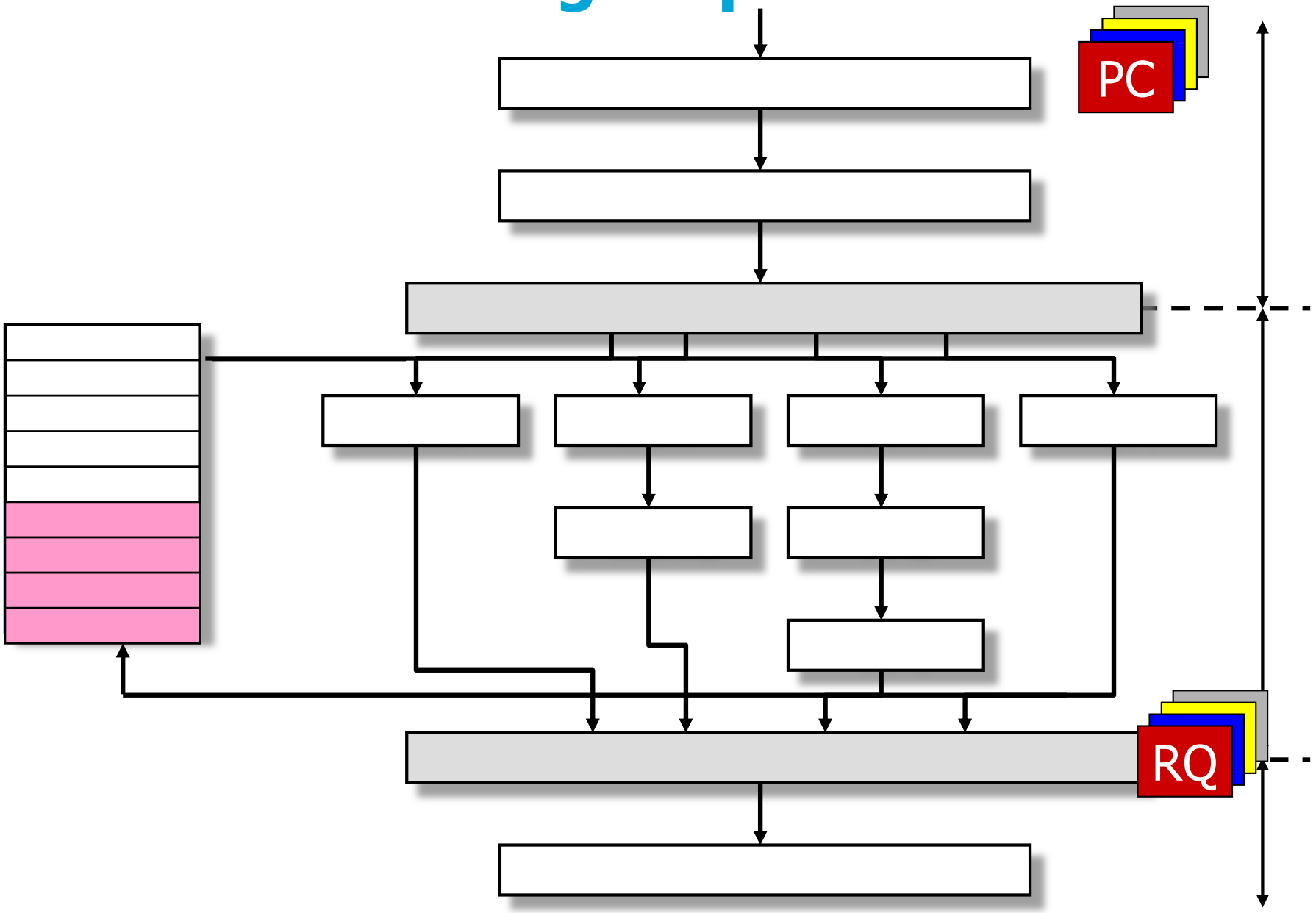


- Superscalar processor – high under-utilization
- Fine-grained multithreading – can only issue instructions from a single thread in a cycle
- Coarse-grained multithreading – same as fine-grain multithreading, but switches threads only at L2 miss
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot

What resources are shared?

- Multiple threads are simultaneously active
- For correctness, each thread needs its own
  - PC
  - logical registers (and its own mapping from logical to physical registers),
- For performance, each thread could have its own:
  - Reorder buffer (ROB) – so that a stall in one thread does not stall commit in other threads,
  - branch predictor,
  - I-cache, D-cache, TLB,
  - … replicate resources for low interference, although more sharing $\Rightarrow$ better resource utilization

Resource sharing&replication



SMT design – based on OoO superscalar

- Instruction issue
  - When to switch (to which) thread, such that the execution unit are fully used, and single-thread performance is not penalized?
    - Prioritized scheme
      - thread 1, is preferred; when thread 1 stalls, thread 2 is preferred
        …
    - Round-Robin
      - all threads compete for resources (fair)
- Execution
  - no changes in the execution path
- Retiring
  - same functionality per thread queues

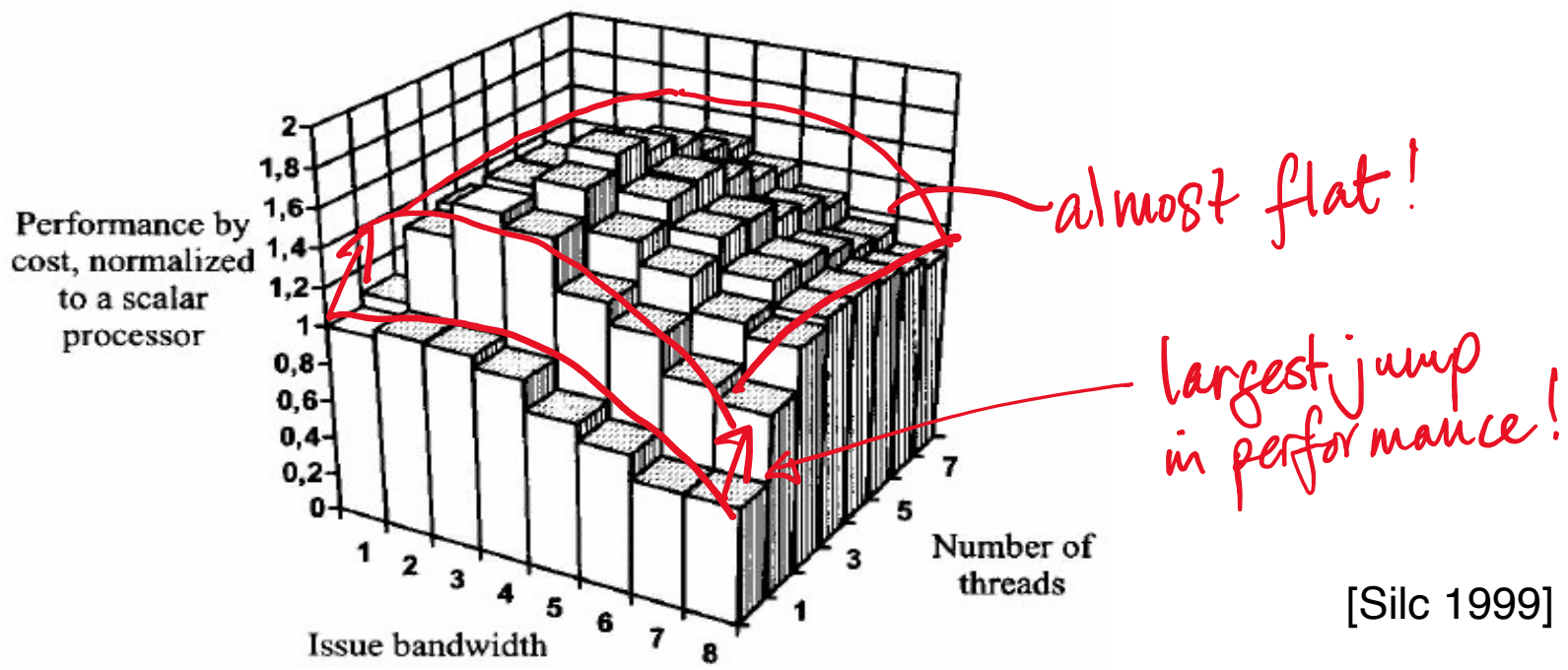
SMT design

- Instruction fetch – which thread to fetch from?
  - While fetching instructions, thread priority can dramatically influence total throughput
  - Static solutions: Round-robin; fetch such that each thread has an equal share of processor resources
    - Each cycle 8 instructions from 1 thread
    - Each cycle 4 instructions from 2 threads, 2 from 4,…
  - Dynamic solutions: check execution queues and favor some threads
    - with minimal # of in-flight branches
    - with minimal # of outstanding misses
    - with minimal # of in-flight instructions

Bottlenecks in SMT

- 8 threads SMT yields throughput improvements of 2x-4x
- Fetch and memory throughput remain bottlenecks
- Try not to affect clock cycle time, especially in
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging
- Larger register file needed to hold multiple contexts
  - register file access is likely to limit the number of threads
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance
  - increased cache associativity to reduce conflicts

SMT: performance vs. cost per unit



[Silc 1999]

- superscalar is cheap only when the number of slots is small
- SMT improves a lot on superscalar already with 2 threads
- with more than 4 threads, the SMT cost-performance decreases

SMT – what to do with it?

- We have HW threading support.
  - Who builds the threads?:
    - programmer, compiler,
    - software marks all potential parallelism, hardware dynamically modulates application parallelism (thread create/schedule/switch in hardware) [Chen2005]
    - just-in-time compilation with multithreading code generation?
  - Who schedules the threads?
    - at coarse grain the OS
    - fine-grain HW

Programmer control on HT

- pthreads library
  - create a thread that the OS will map on a (physical or logical) processor.
  - pthread_setschedparam() sets thread scheduling parameters (policy: SCHED_OTHER, SCHED_RR, or SCHED_FIFO, …).
  - sched_setaffinity() (defines a set of CPUs on which a process threads is eligible to run)
- Windows – similar function calls, e.g Get/SetProcessAffinityMask() ,Get/SetThreadAffinityMask(), Get/SetThreadIdealProcessor(), GetLogicalProcessorInformation()

Parallelism - revisited

- Data parallel
  - bit-level parallel: wider processor data-paths (8→16→32→64…)
  - word-level parallel: vector processors (SIMD)
- "Functional" parallel
  - ILP
    - pipelining, (OoO) superscalar, VLIW, EPIC
  - TLP
    - processes: multi-processors (centralized/distributed)
    - threads (lighter processes, same data space): hardware multi-threading (fine/coarse/SMT)

End – Multithreading slides

CESE 4085 Modern Computer Architecture

Lectures 7 & 8: Data-Level Parallelism

Lecture Overview

Lecture 7:

- Vector Processors
- SIMD Extensions
- Multi-threading (Chapter 3)

Combining topics of 6th and earlier editions of CA book

Lecture 8: (ambitious)

- GPUs
- Loop-level parallelism (*skipped, but is exam material*)
- Thread-Level Parallelism – Multi-processors (Chapter 5)
- Memory Coherence (Chapter 5)
  - Snooping vs. Directory Protocols

Some personal thoughts

These are personal observations about GPUs – before we delve more deeply into technical details:

- GPUs focused on graphics and still focus on graphics
- Floating-point data and processing (—> SIMD)
- (introduction of) CUDA allowed for (more) general-purpose application of GPUs – when certain conditions apply, e.g., embarrassingly parallel data processing
- "grouping" of data for processing — independent data at different levels —> need to learn new terminology and forcing programmers to break up their data in processable sizes
- Abstraction layer across different GPUs requiring recompilation to map operations to hardware (instructions)
- High bandwidth
- Pay attention to comparison with vector processors and SIMD

Graphical Processing Units

Basic idea:

- Heterogeneous execution model
  - CPU is the *host*, GPU is the *device*
- Develop a C-like programming language for GPU
- Unify all forms of GPU parallelism as *CUDA thread*
- Programming model is "Single Instruction Multiple Thread"

Threads and Blocks

A thread is associated with each data element

Threads are organized into blocks

Blocks are organized into a grid

GPU hardware handles thread management, not applications or OS

NVIDIA GPU Architecture

Similarities to vector machines:

- Works well with data-level parallel problems
- Scatter-gather transfers
- Mask registers
- Large register files

Differences:

- No scalar processor
- Uses multithreading to hide memory latency
- Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Example (Figure 4.13 in book)

Code that works over all elements is the grid

- e.g., 8192 elements

Thread blocks break this down into manageable sizes

- (up to) 512 threads per block

Thus grid size = 16 blocks (= 8192 / 512)

SIMD instruction executes 32 elements at a time

A thread block is analogous to a strip-mined vector loop with vector length of 32 (in this example: 16 (=512/32))

Block is assigned to a multithreaded SIMD processor by the thread block scheduler

Current-generation GPUs have 7-15 multithreaded SIMD processors

Terminology (specific to Pascal GPU)

Each thread is limited to 256 (vector) registers (page 319)

- 32 thread * 256 vector registers * 32 registers (per vec-reg) is too much and potentially unused, thus: Dynamic Hardware Register allocation (read page 320)

Groups of 32 threads combined into a SIMD thread or "warp"

- Mapped to 16 physical lanes

Up to 32 warps are scheduled on a single SIMD processor

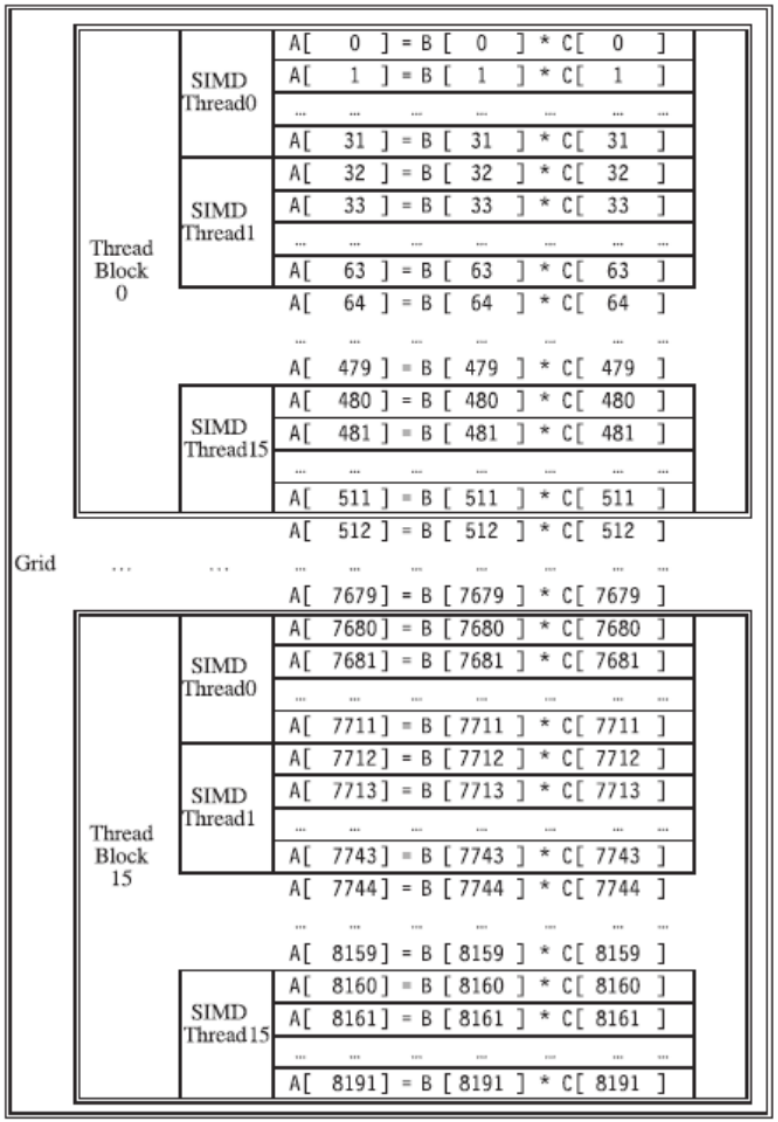
- Each warp has its own PC
- Thread scheduler uses scoreboard to dispatch warps
- By definition, no data dependencies between warps
- Dispatch warps into pipeline, hide memory latency

Thread block scheduler schedules blocks to SIMD processors

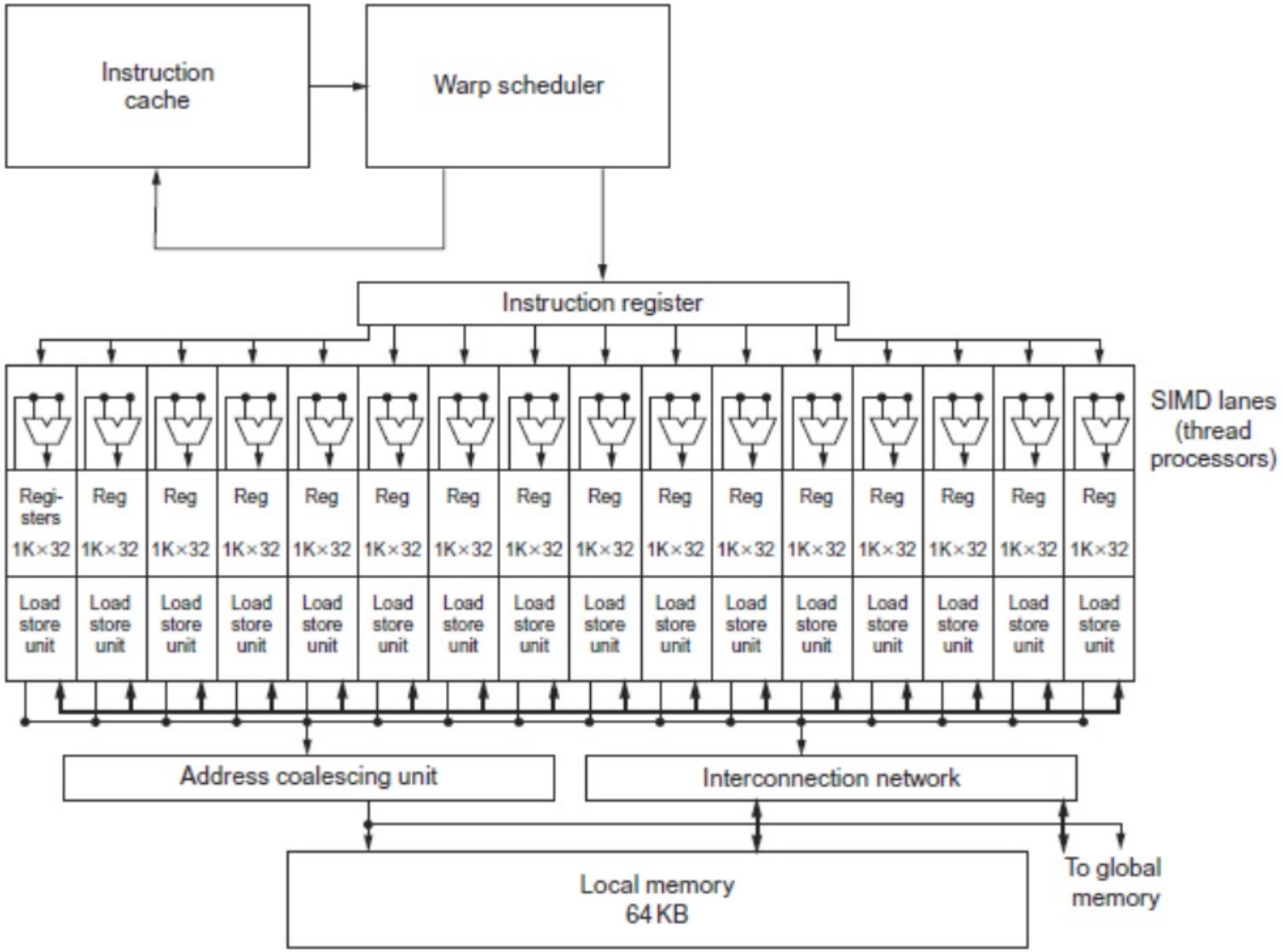
Within each SIMD processor:

- 32 SIMD lanes
- Wide and shallow compared to vector processors

Example (Figure 4.13 in book)



GPU Organization



NVIDIA Instruction Set Arch.

ISA is an abstraction of the hardware instruction set

"Parallel Thread Execution (PTX)"

opcode.type d,a,b,c;

Uses virtual registers —> compiler must allocate these to hw. regs.

Translation to machine code is performed in software

Example:

```
shl.s32     R8, blockIdx, 9; Thread Block ID * Block size (512 or 29)
add.s32     R8, R8, threadIdx    ; R8 = i = my CUDA thread ID
ld.global.f64     RD0, [X+R8]    ; RD0 = X[i]
ld.global.f64     RD2, [Y+R8]    ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4    ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2    ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0       ; Y[i] = sum (X[i]*a + Y[i])
```

Conditional Branching

Like vector architectures, GPU branch hardware uses internal masks

Also uses

- Branch synchronization stack
  - Entries consist of masks for each SIMD lane
  - I.e. which threads commit their results (all threads execute)
- Instruction markers to manage when a branch diverges into multiple execution paths
  - Push on divergent branch
- …and when paths converge
  - Act as barriers
  - Pops stack

Per-thread-lane 1-bit predicate register, specified by programmer

Example

Skip


```
if (X[i] != 0)
      X[i] = X[i] – Y[i];
else X[i] = Z[i];

ld.global.f64    RD0, [X+R8]          ; RD0 = X[i]
setp.neq.s32     P1, RD0, #0          ; P1 is predicate register 1
@!P1, braELSE1, *Push          ; Push old mask, set new mask bits
                                ; if P1 false, go to ELSE1
ld.global.f64    RD2, [Y+R8]          ; RD2 = Y[i]
sub.f64    RD0, RD0, RD2              ; Difference in RD0
st.global.f64    [X+R8], RD0          ; X[i] = RD0
@P1, bra ENDIF1, *Comp                ; complement mask bits
                                ; if P1 true, go to ENDIF1
ELSE1:           ld.global.f64 RD0, [Z+R8]      ; RD0 = Z[i]
            st.global.f64 [X+R8], RD0       ; X[i] = RD0
ENDIF1:   <next instruction>, *Pop ; pop to restore old mask
```

NVIDIA GPU Memory Structures

Each SIMD Lane has private section of off-chip DRAM

- "Private memory"
- Contains stack frame, spilling registers, and private variables

Each multithreaded SIMD processor also has local memory

- Shared by SIMD lanes / threads within a block

Memory shared by SIMD processors is GPU Memory

- Host can read and write GPU memory

Pascal Architecture Innovations

Each SIMD processor has

- Two or four SIMD thread schedulers, two instruction dispatch units
- 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
- Two threads of SIMD instructions are scheduled every two clock cycles

Fast single-, double-, and half-precision

High Bandwith Memory 2 (HBM2) at 732 GB/s

NVLink between multiple GPUs (20 GB/s in each direction)

Unified virtual memory and paging support

Pascal Multithreaded SIMD Proc.

Vector Architectures vs GPUs

SIMD processor analogous to vector processor, both have MIMD

Registers

- RV64V register file holds entire vectors
  - GPU distributes vectors across the registers of SIMD lanes
- RV64 has 32 vector registers of 32 elements (1024)
  - GPU has 256 registers with 32 elements each (8K)
- RV64 has 2 to 8 lanes with vector length of 32, chime is 4 to 16 cycles
  - SIMD processor chime is 2 to 4 cycles
- GPU vectorized loop is grid
- All GPU loads are gather instructions and all GPU stores are scatter instructions

SIMD Architectures vs GPUs

GPUs have more SIMD lanes

GPUs have hardware support for more threads

Both have 2:1 ratio between double- and single-precision performance

Both have 64-bit addresses, but GPUs have smaller memory

SIMD architectures have no scatter-gather support

Loop-Level Parallelism

Skip

Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations

Loop-carried dependence

Example 1:

```
for (i=999; i>=0; i=i-1)
    x[i] = x[i] + s;
```

No loop-carried dependence

Loop-Level Parallelism

Skip

Example 2:

```
for (i=0; i<100; i=i+1) {
      A[i+1] = A[i] + C[i]; /* S1 */
      B[i+1] = B[i] + A[i+1]; /* S2 */
}
```

S1 and S2 use values computed by S1 in previous iteration

S2 uses value computed by S1 in same iteration

Loop-Level Parallelism

Skip

Example 3:

```
for (i=0; i<100; i=i+1) {
      A[i] = A[i] + B[i]; /* S1 */
      B[i+1] = C[i] + D[i]; /* S2 */
}
```

S1 uses value computed by S2 in previous iteration but dependence is not circular so loop is parallel

Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

Loop-Level Parallelism

Skip

Example 4:

```
for (i=0;i<100;i=i+1)  {
    A[i] = B[i] + C[i];
    D[i] = A[i] * E[i];
}
```

Example 5:

```
for (i=1;i<100;i=i+1)  {
    Y[i] = Y[i-1] + Y[i];
}
```

Finding dependencies

Skip

Assume indices are affine:

$a \times i + b$ (i is loop index)

Assume:

Store to $a \times i + b$, then

Load from $c \times i + d$

i runs from m to n

Dependence exists if:

Given j, k such that $m \leq j \leq n$, $m \leq k \leq n$

Store to $a \times j + b$, load from $a \times k + d$, and $a \times j + b = c \times k + d$

Finding dependencies

Skip

Generally cannot determine at compile time

Test for absence of a dependence:

GCD test:

If a dependency exists, GCD(*c*,*a*) must evenly divide (*d*-*b*)

Example:

```
for (i=0; i<100; i=i+1) {
X[2*i+3] = X[2*i] * 5.0;
}
```

Finding dependencies

Skip

Example 2:

```
for (i=0; i<100; i=i+1) {
Y[i] = X[i] / c; /* S1 */
X[i] = X[i] + c; /* S2 */
Z[i] = Y[i] + c; /* S3 */
Y[i] = c - Y[i]; /* S4 */
}
```

Watch for antidependencies and output dependencies

Finding dependencies

Skip

Example 2:

```
for (i=0; i<100; i=i+1) {
Y[i] = X[i] / c; /* S1 */
X[i] = X[i] + c; /* S2 */
Z[i] = Y[i] + c; /* S3 */
Y[i] = c - Y[i]; /* S4 */
}
```

Watch for antidependencies and output dependencies

Reductions

Skip

Reduction Operation:

```
for (i=9999; i>=0; i=i-1)
      sum = sum + x[i] * y[i];
```

Transform to…

```
for (i=9999; i>=0; i=i-1)
      sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
      finalsum = finalsum + sum[i];
```

Do on p processors:

```
for (i=999; i>=0; i=i-1)
      finalsum[p] = finalsum[p] + sum[i+1000*p];
```

Note: assumes associativity!

Fallacies and Pitfalls

Skip

GPUs suffer from being coprocessors

- GPUs have flexibility to change ISA

Concentrating on peak performance in vector architectures and ignoring start-up overhead

- Overheads require long vector lengths to achieve speedup

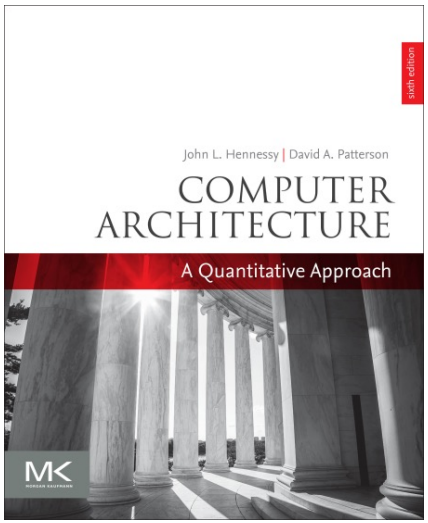
Increasing vector performance without comparable increases in scalar performance

You can get good vector performance without providing memory bandwidth

On GPUs, just add more threads if you don't have enough memory performance

END of NEW CH4

NEW CH5



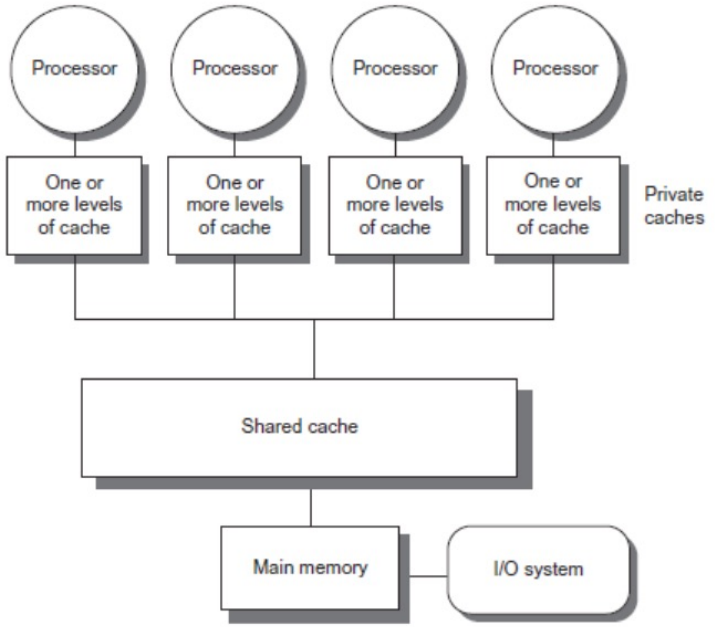
Chapter 5

Thread-Level Parallelism

Types

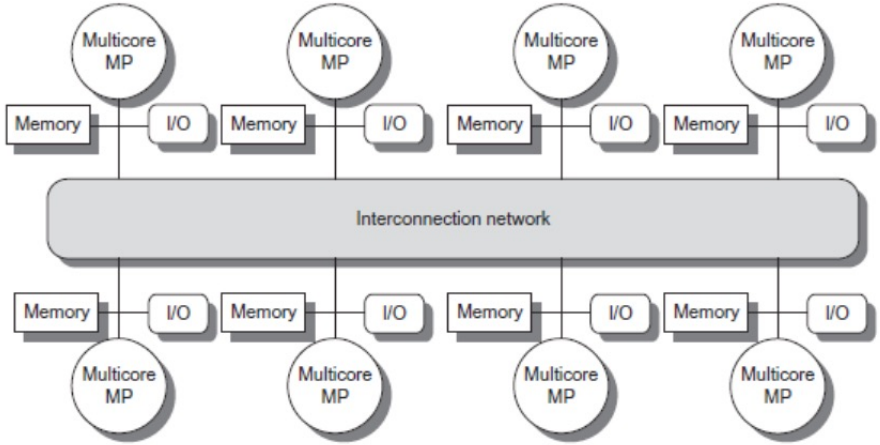
Symmetric multiprocessors (SMP)

- Small number of cores
- Share single memory with uniform memory access/latency (UMA)



Distributed shared memory (DSM)

- Memory distributed among processors
- Non-uniform memory access/latency (NUMA)
- Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks



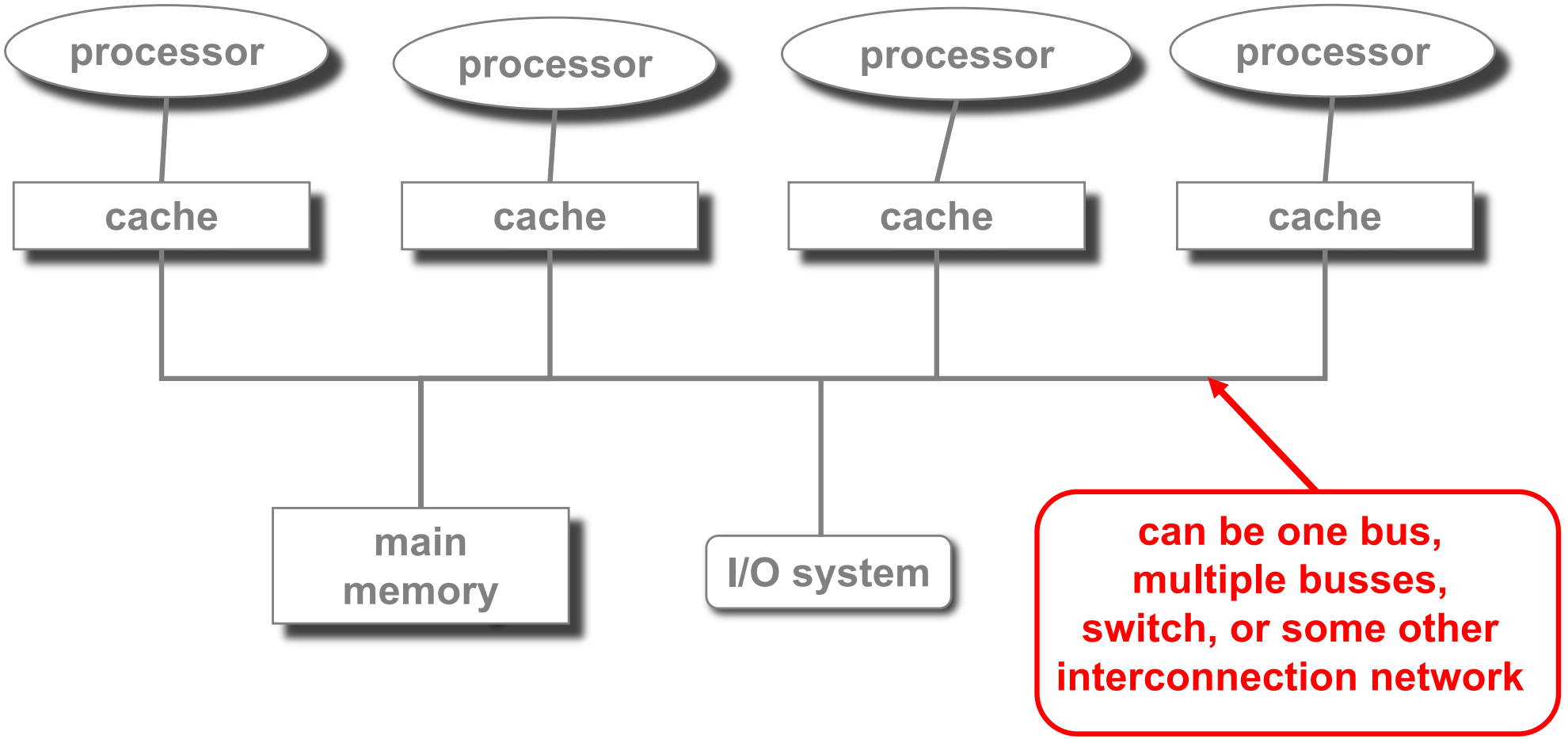
Shared vs. Distributed Memory

- Centralized shared-memory architectures
  - Main memory is shared between the processors
  - At most few dozen processors
  - Also called
    - Symmetric (shared-memory) Multiprocessors (SMPs)
      - single main memory that has symmetric relationship to all processors
    - Uniform Memory Access (UMA) architectures
      - memory has uniform access time from any processor (in absence of contention)
- Common belief: easier to program than distributed memory architectures

Shared vs. Distributed Memory

- Centralized shared-memory architectures
  - Main memory is shared between the processors
  - At most few dozen processors
  - Also called
    - Symmetric (shared-memory) Multiprocessors (SMPs)
      - single main memory that has symmetric relationship to all processors
    - Uniform Memory Access (UMA) architectures
      - memory has uniform access time from any processor (in absence of contention)
- Common belief: easier to program than distributed memory architectures

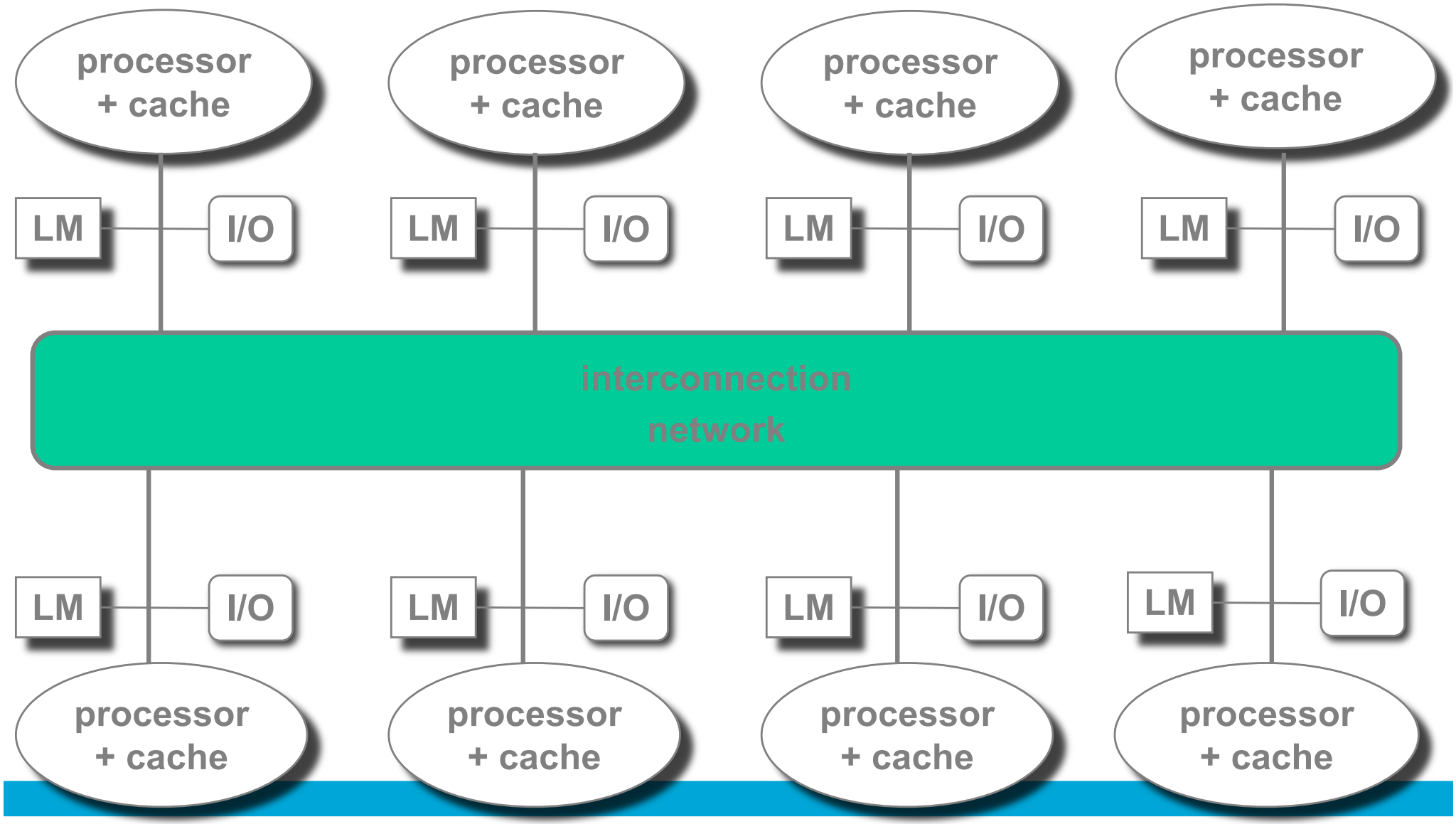
Centralized shared-memory MP



Distributed-Memory MPs

- Distributed-memory architectures
  - Memory is physically distributed among the processors
  - Typically have more processors than SMPs
  - More difficult to program than SMPs
  - Require some kind of interconnect
    - direct (switches)
    - indirect (2- or higher dimensional meshes, hypercubes, fat trees, etc.)
  - Also called Non-uniform Memory Access (NUMA) architectures

Distributed-Memory MPs



SMPs – memory architecture

- Private vs. shared data
  - private data: used by a single processor
    - when cached, in only one cache => no problem
  - shared data: used by multiple processors, allows processors to communicate
    - when cached, can be in multiple caches at the same time

- To ameliorate the problem of long memory access latency, cache both private and shared data
  - also reduce the interconnect bandwidth

Cache Coherence

Processors may see different values through their caches:

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

Coherence and Consistency

- Informally:
  - "Any read must return the most recent write"
  - Too difficult (and expensive) to enforce
  - Too vague and simplistic; it tells 2 things:
    - what values can be returned by a read (coherence)
    - when a written value will be returned by a read (consistency).
    - coherence defines behavior to same location, consistency defines behavior to other locations (order among accesses)

P_1	P_2
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

  - the intuition might not necessarily be true
  - a problem even without caches

Cache Coherence

Coherence

- All reads by any processor must return the most recently written value
- Writes to the same location by any two processors are seen in the same order by all processors

Consistency

- When a written value will be returned by a read
- If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

Enforcing Coherence

Coherent caches provide:

- *Migration*: movement of data
- *Replication*: multiple copies of data

Cache coherence protocols

- Directory based
  - Sharing status of each block kept in one location
- Snooping
  - Each core tracks sharing status of each block

Cache Coherence Protocols

- Snooping
  - Send address of all request for data to all processors
  - Processors snoop to see if they have copy (cache tags!) of requested data and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale machines (most of the market)
  - The data "sharing" status is kept in each cache
- Directory-based
  - Keep track of what is being shared in one centralized place
  - Distributed memory => distributed directory for scalability (avoids bottlenecks, hot spots)
  - Scales better than snooping
  - Actually existed *before* snooping

Snoopy Coherence Protocols

Write invalidate (most used approach, achieve exclusivity)

- On write, invalidate all other copies
- Use bus itself to serialize
  - Write cannot complete until bus access is obtained

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

Write update

- On write, update all copies – lots of bandwidth needed

Basics of Write Invalidate

- Use the bus to perform invalidates
  - To perform an invalidate, acquire bus access and broadcast the address to be invalidated
    - all processors (actually their cache controllers) snoop the bus, listening to addresses
    - if the address is in my cache, invalidate my copy
- Serialization of bus access enforces write serialization
- On a read miss (may also be generated by an invalidation), where is the most recent value?
  - Easy for write-through caches (it is in the memory)
  - For write-back caches, again use snooping

Snoopy Coherence Protocols

Locating an item when a read miss occurs

- In write-back cache, the updated value must be sent to the requesting processor

Cache lines marked as shared or exclusive/modified

- Only writes to shared lines need an invalidate broadcast
  - After this, the line is marked as exclusive

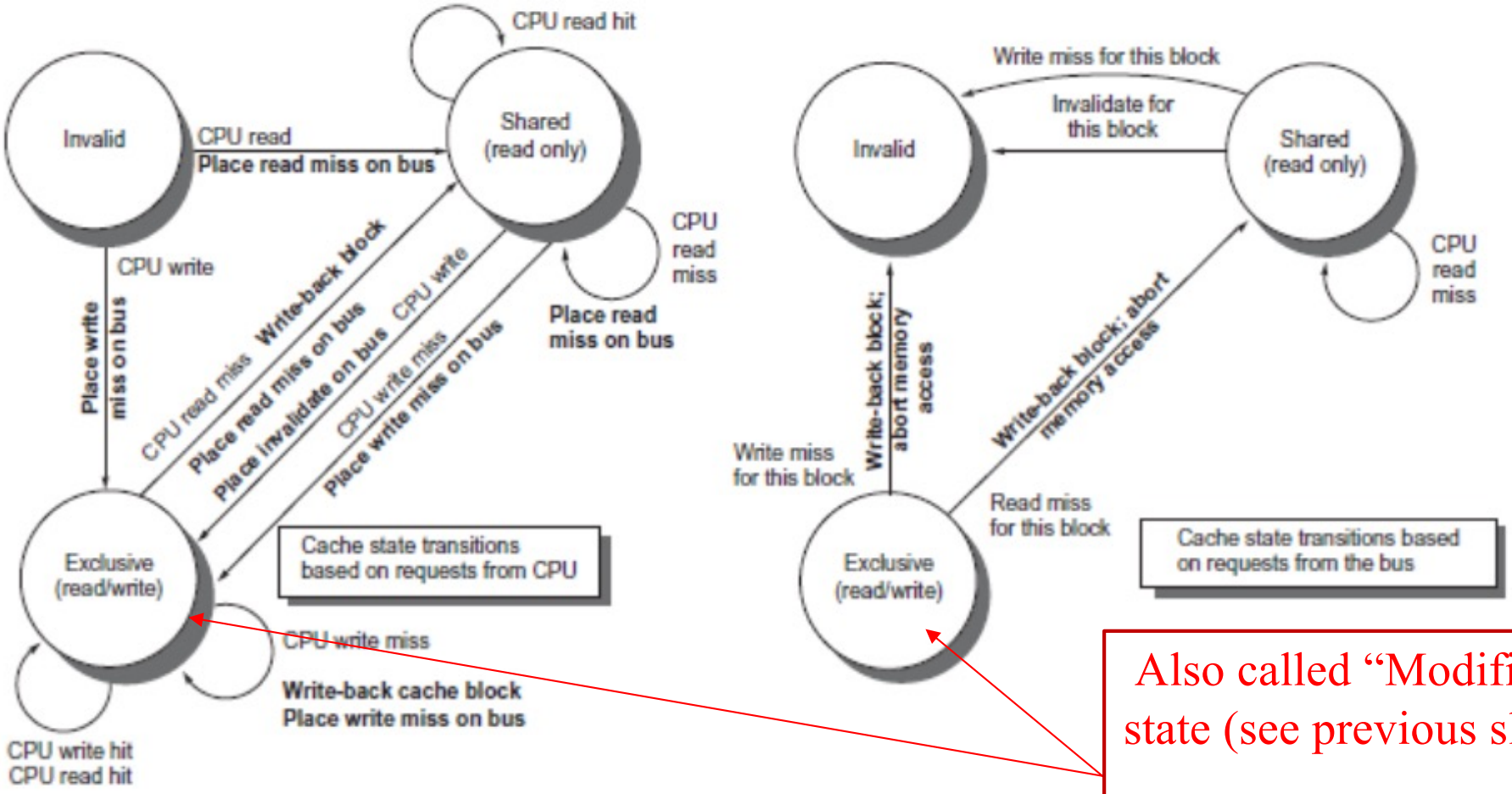
Snoopy Coherence Protocols

Self-study

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, because they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block; then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to read shared data: place cache block on bus, write-back block, and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Leads to FSMs on next slide = same information

Snoopy Coherence Protocols



Also called "Modified" state (see previous slide)

Thus: MSI protocol

Snoopy Coherence Protocols

Complications for the basic MSI protocol:

- Operations are not atomic
  - E.g. detect miss, acquire bus, receive a response
  - Creates possibility of deadlock and races
  - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate

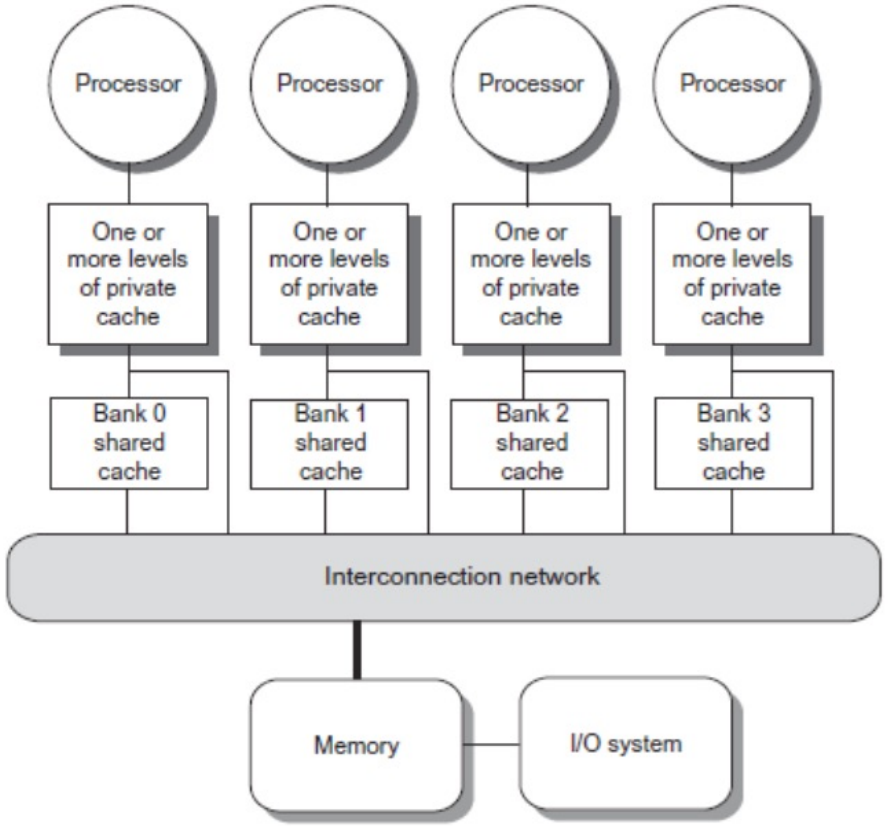
Extensions:

- Add exclusive state to indicate clean block in only one cache (MESI protocol)
  - Prevents needing to write invalidate on a write
- Owned state

Coherence Protocols: Extensions

Shared memory bus and snooping bandwidth is bottleneck for scaling symmetric multiprocessors

- Duplicating tags
- Place directory in outermost cache
- Use crossbars or point-to-point networks with banked memory



Coherence Protocols

Every multicore with >8 processors uses an interconnect other than bus

- Makes it difficult to serialize events
- Write and upgrade misses are not atomic
- How can the processor know when all invalidates are complete?
- How can we resolve races when two processors write at the same time?
- Solution: associate each block with a single bus

Limitations in SMP and snooping protocols

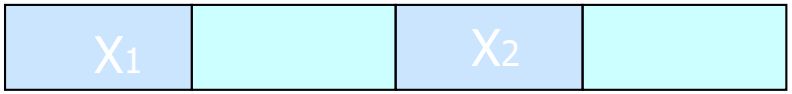
- Number of processors increase ⇒ bus traffic increases
- Processor speed increases ⇒ bus traffic increases
  - bus must support both coherence traffic & normal memory traffic
- ⇒ Multiple buses or interconnection networks (cross bar or small point-to-point)
  - however snooping still requires broadcast; caches have to respond to requests from all other caches ⇒ limits the scalability of SMT with snooping

- AMD Opteron example:
  - A memory connected directly to each dual-core chip
  - Point-to-point connections for up to 4 chips
  - coherence: b-cast to find shared copies, but ACKs to order operations
  - Remote memory and local memory latency are similar, allowing OS Opteron as UMA computer, though the memory is distributed

True Vs. False Sharing

* Write invalidate
* 1 valid bit per cache block

- True sharing: the word(s) being read is (are) the same as the word(s) being written
- False sharing: the word being read is different from the word being written, but they are in same cache block



Need to (write) invalidate to gain exclusivity (MSI) for the write operation (page 394)

Time	P1	P2	Comment
1	Write X1		True sharing miss (assume X1 was read by P2)- invalidation required
2		Read X2	False sharing miss, since X2 is invalidated by the write of X1 by P1
3	Write X1		False sharing miss, since X2 is shared again after P2 read it
4		Write X2	False sharing miss since writing to X2 while invalid for the X1 write
5	Read X2		True sharing miss since it involves a read of X2 which was invalidated

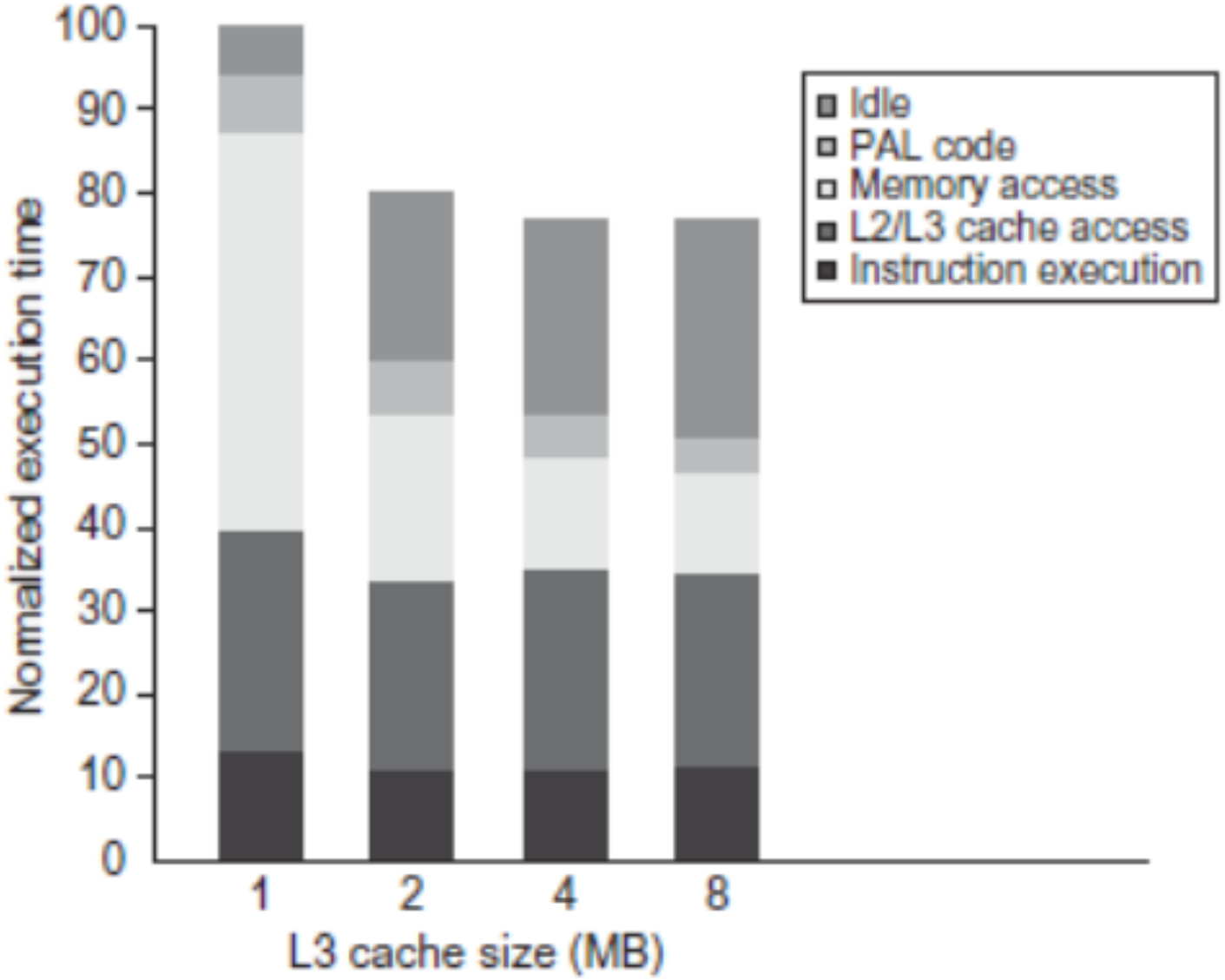
Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of
  - Uniprocessor cache miss traffic
  - Traffic caused by communication
    - Results in invalidations and subsequent cache misses
  - 4th C: coherence miss (more of those in tightly coupled applications that share large amounts of data)
    - Joins Compulsory, Capacity, Conflict
    - Programmers must prevent false sharing
      - if we distribute data between processors, make sure it's aligned at cache block boundaries

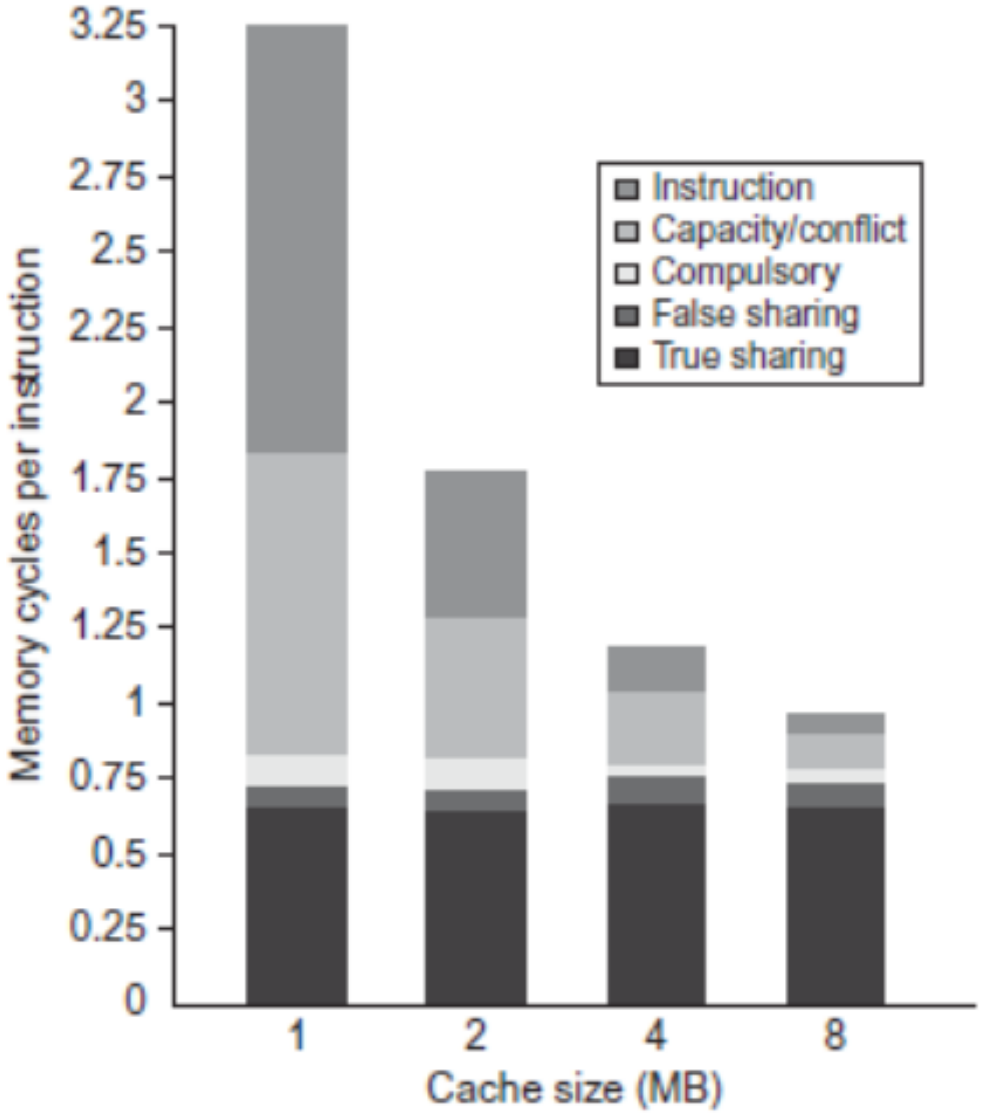
Performance

- Coherence influences cache miss rate
  - Coherence misses
    - True sharing misses
      - Write to shared block (transmission of invalidation)
      - Read an invalidated block
    - False sharing misses
      - Read an unmodified word in an invalidated block

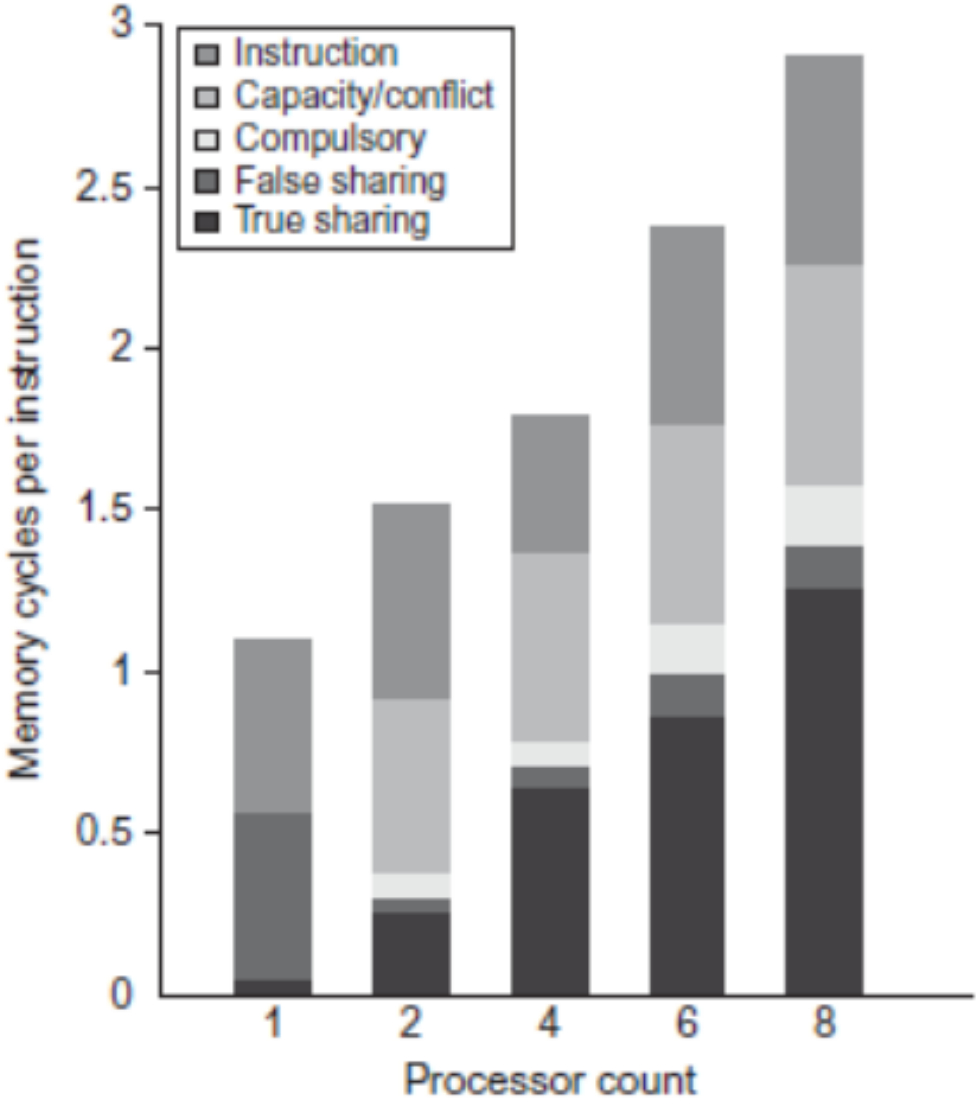
Performance Study: Commercial Workload *SKIP*



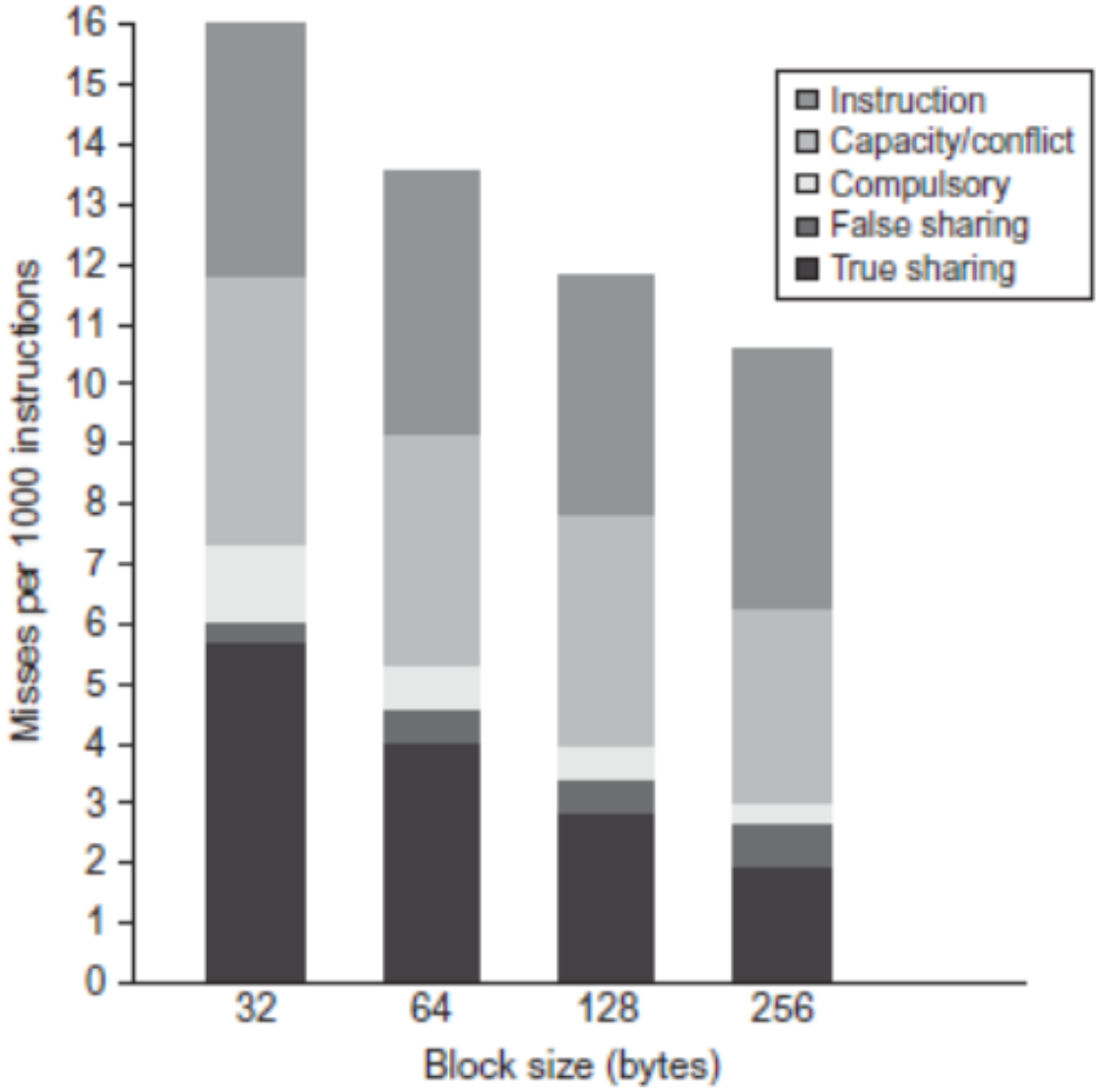
Performance Study: Commercial Workload *SKIP*



Performance Study: Commercial Workload *SKIP*



Performance Study: Commercial Workload *SKIP*



Qualitative Performance Differences *Self-study*

- Performance differences between write invalidate and write update:
  - Multiple writes to same word require
    - multiple write broadcasts in write update protocol
    - one invalidation in write invalidate
  - When a cache block consists of multiple words, each word written to a cache block requires
    - multiple write broadcasts in write update protocol
    - one invalidation in write invalidate
    - write invalidate works on cache blocks, write update on words/bytes
  - Delay between writing a word in one processor and reading the new value in another is less in write update
- And the winner is: write invalidate because bus bandwidth is most precious

Directory Protocols

Self-study

Snooping schemes require communication among all caches on every cache miss

- Limits scalability
- Another approach: Use centralized directory to keep track of every block
  - Which caches have each block
  - Dirty status of each block

Implement in shared L3 cache

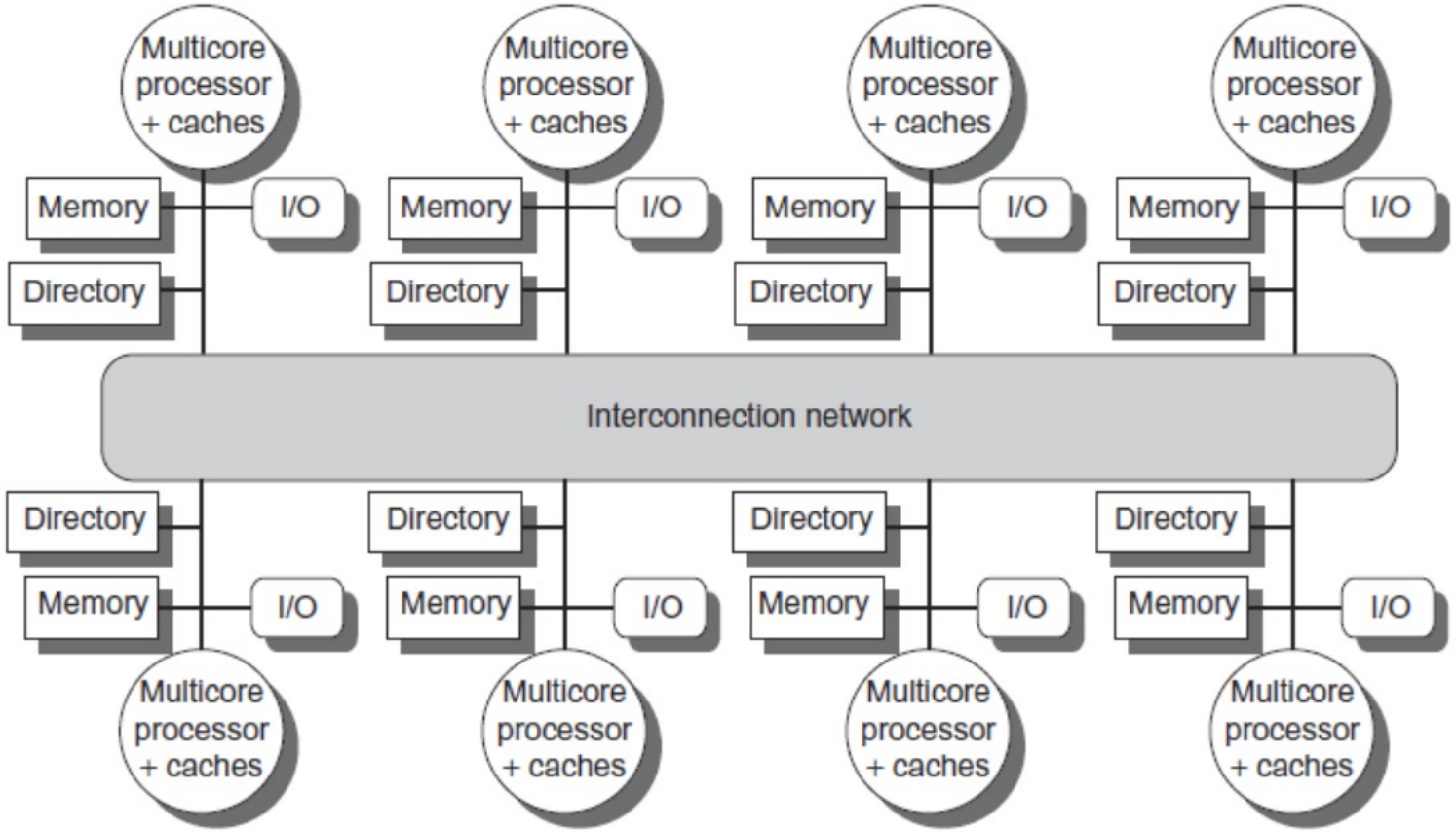
- Keep bit vector of size = # cores for each block in L3
- Not scalable beyond shared L3

Directory Protocols

Self-study

Alternative approach:

Distribute memory



Directory Protocols

Self-study

For each block, maintain state:

- Shared
  - One or more nodes have the block cached, value in memory is up-to-date
  - Set of node IDs
- Uncached
- Modified
  - Exactly one node has a copy of the cache block, value in memory is out-of-date
  - Owner node ID

Directory maintains block states and sends invalidation messages

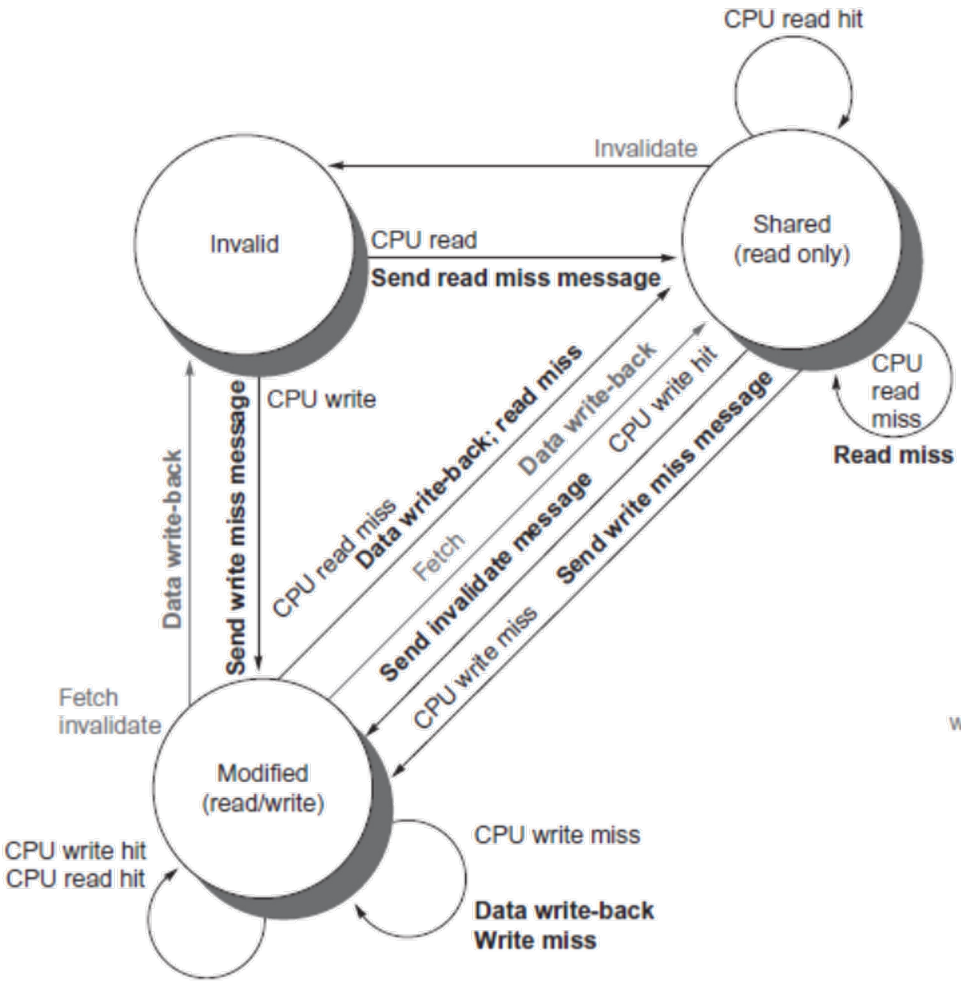
Messages

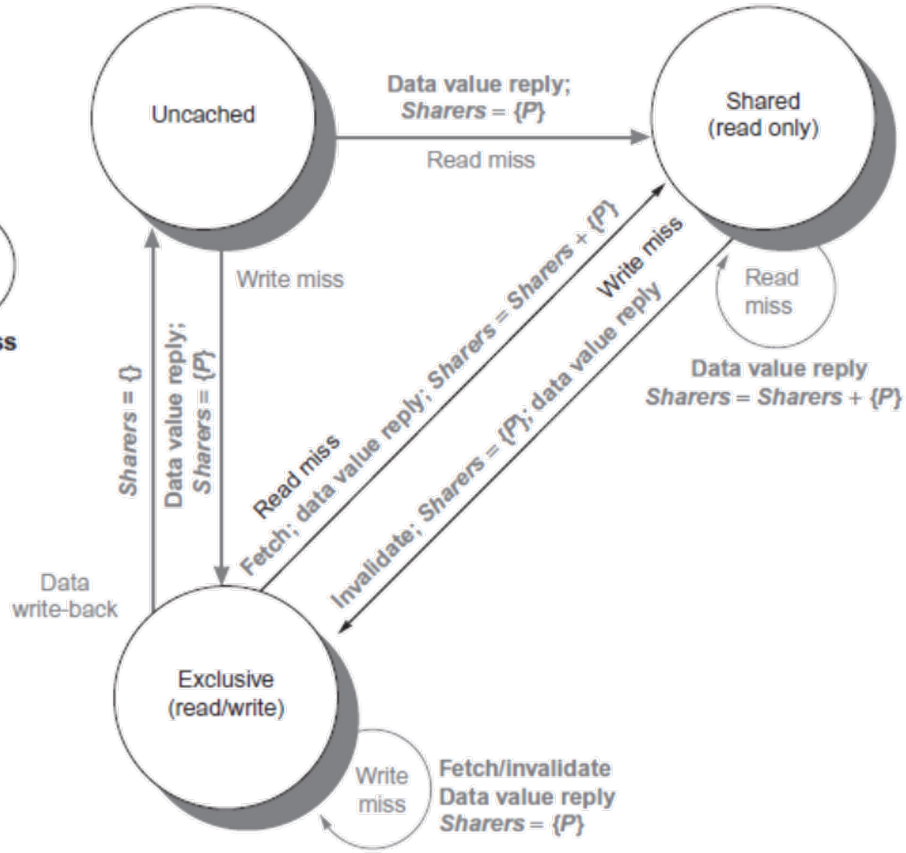
Self-study

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/ invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write back a data value for address A.

Directory Protocols

Self-study





Directory Protocols

Self-study

For uncached block:

- Read miss
  - Requesting node is sent the requested data and is made the only sharing node, block is now shared
- Write miss
  - The requesting node is sent the requested data and becomes the sharing node, block is now exclusive

For shared block:

- Read miss
  - The requesting node is sent the requested data from memory, node is added to sharing set
- Write miss
  - The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive

Directory Protocols

Self-study

For exclusive block:

- Read miss
  - The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
- Data write back
  - Block becomes uncached, sharer set is empty
- Write miss
  - Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

Synchronization

Skip

Basic building blocks:

- Atomic exchange
  - Swaps register with memory location
- Test-and-set
  - Sets under condition
- Fetch-and-increment
  - Reads original value from memory and increments it in memory
- Requires memory read and write in uninterruptable instruction

- RISC-V: load reserved/store conditional
  - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails

Implementing Locks

Skip

Atomic exchange (EXCH):

```
try: mov x3,x4   ;mov exchange value
     lr x2,x1    ;load reserved from
     sc x3,0(x1) ;store conditional
     bnez x3,try ;branch store fails
     mov x4,x2   ;put load value in x4?
```

Atomic increment:

```
try: lr x2,x1    ;load reserved 0(x1)
     addi x3,x2,1;increment
     sc x3,0(x1) ;store conditional
     bnez x3,try ;branch store fails
```

Implementing Locks

Skip

Lock (no cache coherence)

```
        addi x2,R0,#1
lockit: EXCH x2,0(x1)      ;atomic exchange
        bnez x2,locket     ;already locked?
```

Lock (cache coherence):

```
lockit: ld x2,0(x1)        ;load of lock
        bnez x2,locket     ;not available-spin
        addi x2,R0,#1      ;load locked value
        EXCH x2,0(x1)      ;swap
        bnez x2,locket     ;branch if lock wasn't 0
```

Implementing Locks

Skip

Advantage of this scheme: reduces memory traffic

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied.
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied.
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss; generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Models of Memory Consistency

Skip

Processor 1:	Processor 2:
A=0	B=0
...	...
A=1	B=1
if (B==0) ...	if (A==0) ...

- Should be impossible for both if-statements to be evaluated as true
  - Delayed write invalidate?
- Sequential consistency:
  - Result of execution should be the same as long as:
    - Accesses on each processor were kept in order
    - Accesses on different processors were arbitrarily interleaved

Implementing Locks

Skip

To implement, delay completion of all memory accesses until all invalidations caused by the access are completed

- Reduces performance!

Alternatives:

- Program-enforced synchronization to force write on processor to occur before read on the other processor
  - Requires synchronization object for A and another for B
    - "Unlock" after write
    - "Lock" after read

Relaxed Consistency Models

Skip

Rules:

X → Y

Operation X must complete before operation Y is done

Sequential consistency requires:

R → W, R → R, W → R, W → W

Relax W → R

"Total store ordering"

Relax W → W

"Partial store order"

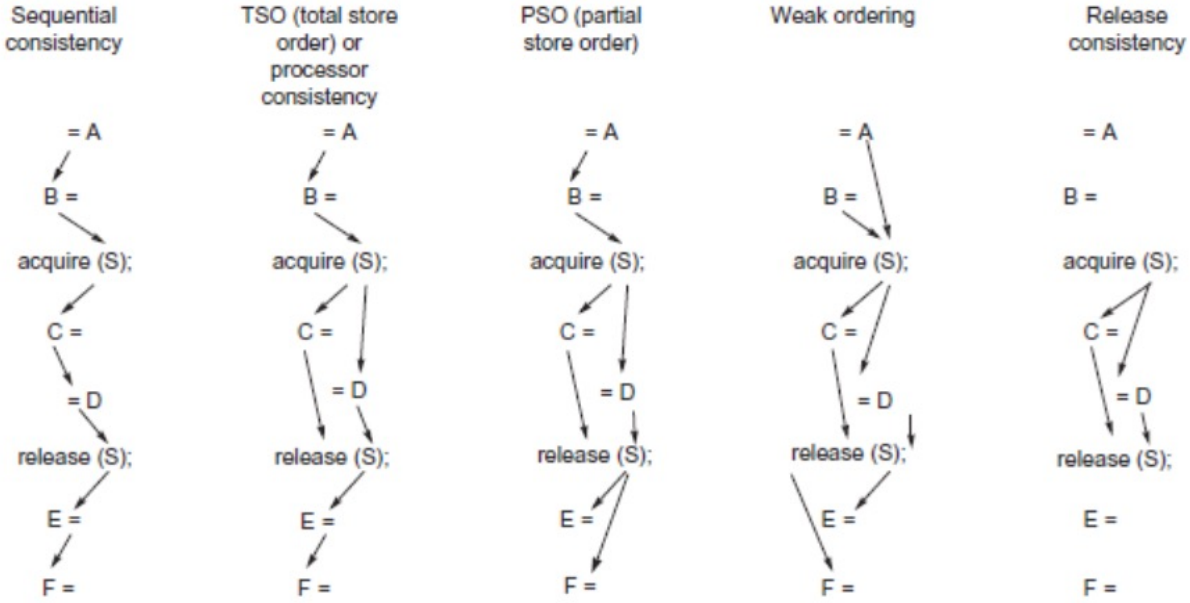
Relax R → W and R → R

"Weak ordering" and "release consistency"

Relaxed Consistency Models

Skip

Model	Used in	Ordinary orderings	Synchronization orderings
Sequential consistency	Most machines as an optional mode	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow R$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Total store order or processor consistency	IBMS/370, DEC VAX, SPARC	$R \rightarrow R$, $R \rightarrow W$, $W \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Partial store order	SPARC	$R \rightarrow R$, $R \rightarrow W$	$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Weak ordering	PowerPC		$S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
Release consistency	MIPS, RISC V, Armv8, C, and C++ specifications		$S_A \rightarrow W$, $S_A \rightarrow R$, $R \rightarrow S_R$, $W \rightarrow S_R$, $S_A \rightarrow S_A$, $S_A \rightarrow S_R$, $S_R \rightarrow S_A$, $S_R \rightarrow S_R$



Relaxed Consistency Models

Skip

Consistency model is multiprocessor specific

Programmers will often implement explicit synchronization

Speculation gives much of the performance advantage of relaxed models with sequential consistency

- Basic idea: if an invalidation arrives for a result that has not been committed, use speculation recovery

Fallacies and Pitfalls

SKIP

Fallacy = common misbelief

Pitfall = easily made mistake

P: Measuring performance of multiprocessors by linear speedup versus execution time

F: Amdahl's Law doesn't apply to parallel computers

F: Linear speedups are needed to make multiprocessors cost-effective

Doesn't consider cost of other system components

P: Not developing the software to take advantage of, or optimize for, a multiprocessor architecture

END of NEW CH5