More on Instruction-Level Parallelism

Professor Hugh C. Lauer
Professor Thérèse Smith

CS-4515, System Programming Concepts

Instruction-Level Parallelism

- **When exploiting instruction-level parallelism, goal is to minimize CPI**
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- **Parallelism within basic blocks is limited**
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches — i.e.,
    - Loops
    - Speculation

Basic Block (definition):–
A straight-line code sequence, with no branches *in* except entry and no branches *out* except exit. (p. 149)

Topics

- **Today**
  - Dependences
  - Loop Unrolling
  - Multi-threading
- **Next time**
  - Tomasulo's Algorithm
  - Core i7 pipeline

Dependences

- **Types of dependences:**
  1. Data dependence
  2. Name dependence
  3. Control dependence

Data Dependence

- **Instruction *j* is data dependent on instruction *i* if:–**
  - *i* produces a result that *j* may need, or
  - *j* is data dependent on *k* and *k* is data dependent on *i*

- **For example:–**

```
ADD     R3, R2, R1
SUB     R4, R3, 1       // SUB is data dependent on ADD
```

- **Also:**

```
DIV     R6, R4, R1      // DIV is data dependent on SUB
                        //    and therefore on ADD
```

Data Dependence (Implications)

- **Possibility of a data hazard**
- **Restricted order of execution**
  - i.e., no overlap, limited re-ordering
- **Establish upper bound of amount of parallelism**

Note:–
- **Data dependences in registers *can be detected* at runtime**
- **Data dependences in memory are very difficult to detect**

Name Dependence

- **Two instructions use the same name but no flow of information**
  - Not a true data dependence, *but a problem when reordering instructions*
- ***Antidependence*: instruction *j* writes a register or memory location that earlier instruction *i* reads**
  - Initial ordering (*i* before *j*) must be preserved
- ***Output dependence*: instruction *i* and instruction *j* write the same register or memory location**
  - Ordering must be preserved
- **To resolve, use renaming techniques**

Control Dependency

- **Determines ordering of instructions when branching**
  - To ensure instructions executed in correct program order
- **For example:**

```
      DADDU    R2, R3, R4
      BEQZ     R2, L1
      LW       R1, 0(R2)
L1:   ...
```

No data dependence between BEQZ & LW

However, cannot re-order because LW might dereference null pointer!

Control Dependences (continued)

- **Ordering of instruction *i* with respect to a branch instruction**
  - Instruction control dependent on a branch cannot be moved before the branch so that its execution is *no longer controlled* by the branch
  - An instruction not control dependent on a branch cannot be moved after the branch so that its execution *is controlled* by the branch

Compiler Techniques for Dependences

- **Scheduling - finding sequences of unrelated instructions that can be overlapped**
  - Rearrange to fill stall slots

- **Loop unrolling – replicate the loop body multiple times**
  - Eliminates branches
  - Allows instructions across iterations to be scheduled together

Scheduling Example

```
for (i = 99; i >= 0; i = i - 1)
   x[i] = x[i] + s;
```

Unscheduled code

```
Loop:  L.D     F0, 0(R1)
       stall
       ADD.D   F4, F0, F2
       stall
       stall
       S.D     F4, 0(R1)
       DADDUI  R1, R1, #-8
       stall                    (assume integer load latency = 1)
       BNE     R1, R2, Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Scheduling Example

```
for (i = 99; i >= 0; i = i - 1)
   x[i] = x[i] + s;
```

Scheduled code

```
Loop:  L.D     F0, 0(R1)
       DADDUI  R1, R1, #-8
       ADD.D   F4, F0, F2
       stall
       stall
       S.D     F4, 0(R1)
       BNE     R1, R2, Loop
```

Seven cycles per iteration!

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling

- Unroll by a factor of 4 (assume # elements is divisible by 4)
- Eliminate unnecessary instructions

```
Loop:  L.D      F0,0(R1)
       ADD.D    F4,F0,F2
       S.D      F4,0(R1)      // drop DADDUI & BNE
       L.D      F6,-8(R1)
       ADD.D    F8,F6,F2
       S.D      F8,-8(R1)     // drop DADDUI & BNE
       L.D      F10,-16(R1)
       ADD.D    F12,F10,F2
       S.D      F12,-16(R1)   // drop DADDUI & BNE
       L.D      F14,-24(R1)
       ADD.D    F16,F14,F2
       S.D      F16,-24(R1)
       DADDUI   R1,R1,#-32    // correct increment of R1
       BNE      R1,R2,Loop
```

- Note number of live registers *vs.* original loop
- Each iteration has separate registers

- Not shown:–
  Two pipeline stall cycles between each ADD.D and S.D (prev slide)

Loop Unrolling/Pipeline Scheduling

- **Pipeline schedule the unrolled loop:**

```
Loop:  L.D      F0,0(R1)
       L.D      F6,-8(R1)
       L.D      F10,-16(R1)
       L.D      F14,-24(R1)
       ADD.D    F4,F0,F2
       ADD.D    F8,F6,F2
       ADD.D    F12,F10,F2
       ADD.D    F16,F14,F2
       S.D      F4,0(R1)
       S.D      F8,-8(R1)
       DADDUI   R1,R1,#-32
       S.D      F12,16(R1)    // displacement adjusted for DADDUI
       S.D      F16,8(R1)     // displacement adjusted for DADDUI
       BNE      R1,R2,Loop
```

Fourteen cycles per 4 iterations!

I.e., 3.5 cycles per iteration

Strip Mining

- **Unknown number of loop iterations?**
  - Number of iterations = *n*
  - Goal: make *k* copies of the loop body
  - Generate pair of loops:
    - First executes *n* mod *k* times
    - Second executes *n* / *k* times
  - Known as “Strip mining”

Questions?

More on Instruction-level Parallelism

- **Tomasulo's algorithm**
  - (lots of) Register renaming
  - More on Friday
- **Branch predictors again**
  - Hardware speculation
- **Multithreading**

Note on Branch Predicting

- **§3.6 spends a lot of space on branch prediction**
  - Branches in real code occur so frequently in "wide issue processors" …
  - … that good prediction is important to performance
- **Lots of techniques beyond what was covered in earlier topic**
- **Few details on how they work**

Another Challenge

- **ILP has its advantages**
  - Transparent to programmers
  - Does not require special compilation for each generation of processor
- **But**
  - Can suffer from cache misses reducing utilization of functional units
  - Memory stalls cannot effectively be covered by more ILP
- **Can we do better?**

Challenge — continued

- With many functional units
- … and lots of instructions executing in parallel
- … leading to more and more pipeline stalls
- Can will fill empty pipeline slots with instructions from *another* program?

Yes! Multithreading

Aka "Hyperthreading"

- **Multiple**
  - Program counters
  - Condition codes
  - Architectural registers
  - Page table pointers
  - Etc.
- **All sharing same functional units**
- **… caches,**
- **… memory,**
- **… execution units,**
- **… etc.**

Traditional Process/Thread Switch

- **Occurs only during kernel call or interrupt**
- **OS must save**
  - Registers & condition codes
  - Program counter & page table pointers
  - Lots of other stuff
- **May invoke scheduling decisions**
- **Caches quickly go stale**
  - Lots of cache and TLB misses for new thread
  - Even more for a new process and address space
- **10s or 100s of thousands of instructions**

Recall that in Linux, a *process* and a *thread* are essentially the same thing.

From Core i7 memory hierarchy, caches and TLBs can support multiple address spaces

What if …

- **Processor could execute two (or more) processes or threads at the "same time"**
- **Each thread executes independently**
  - Separate registers, program counter, etc.
  - (possibly) separate virtual memories
- **IF stage fetches instructions from multiple streams in i-cache**
  - Each instruction is tagged with stream identity
    - To identify which set of registers, etc., to use
  - Multiple instructions issued at same time
    - Regardless of process/thread identity

Multithreading (continued)

- Each thread can make forward progress
- Instructions of a thread fill empty pipeline slots left by hazards and stalls of other threads

Hardware options for multithreading

- **Fine grained**
  - Switch thread on each clock cycle.
  - Prevents stalls form holding up throughput
  - Slows down individual thread execution times
  - Trades increased multi-threading for worse performance of single threads
- **Course grained**
  - Only switches on costly cache stalls (L2, L3)
  - Limited throughput capabilities
  - Suffers from time for stall to propagate through pipeline
  - No major use in processors

Hardware options for threading (cont'd)

- **Simultaneous multi-threading (SMT)**
  - Needs multiple issue, dynamically scheduled processor
  - Uses threads to hide long latency commands by keeping them loaded simultaneously
  - Derivative of fine grained multi-threading

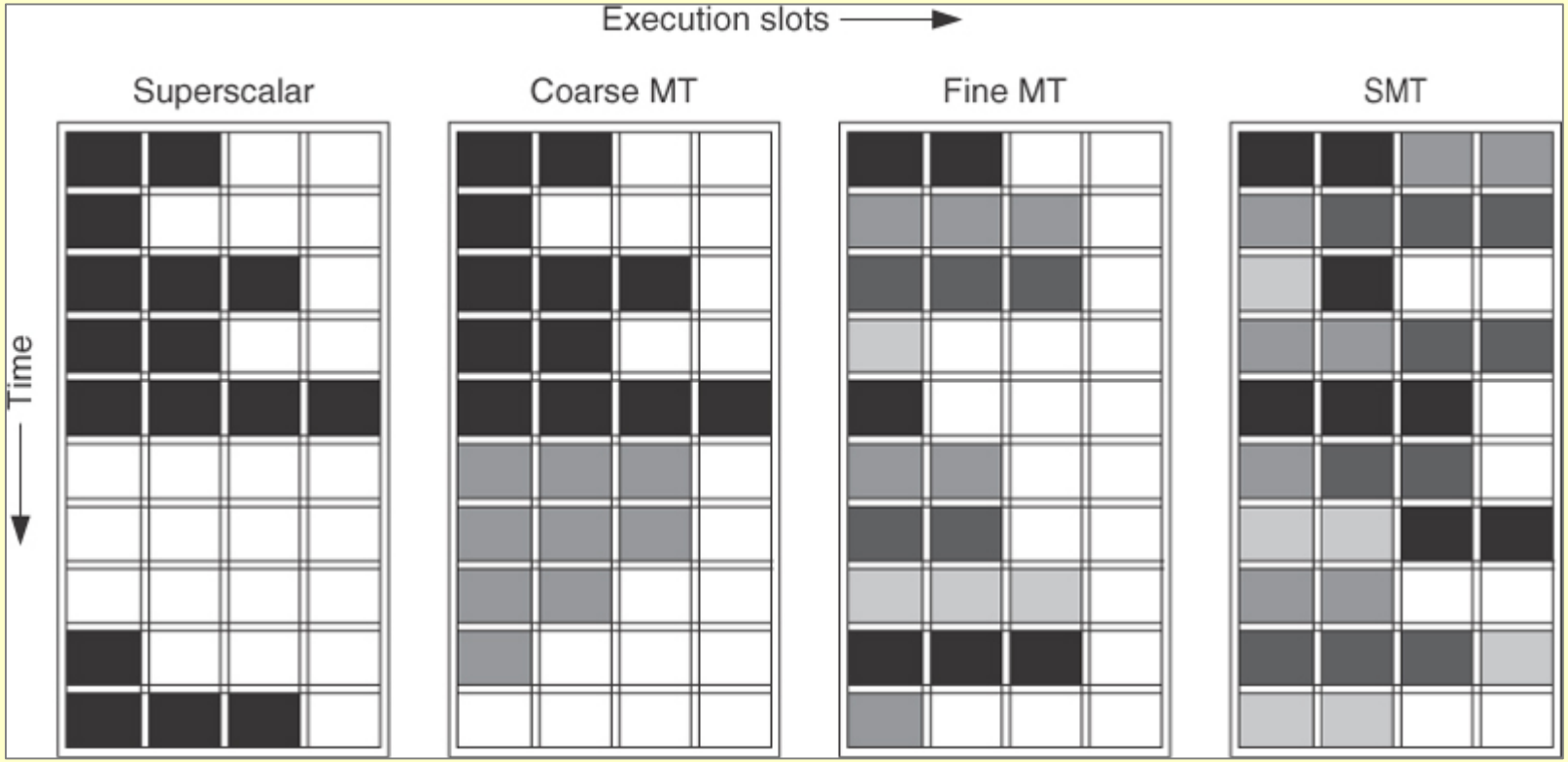

Figure 3.31 in 6th ed.

Figure 3.28 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has eight threads, the Power7 has four, and the Intel i7 has two. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

Comparisons from Fig 3.31

- **Superscalar is left with nothing to do quite often**
  - No way around it
- **Course MT is slightly better**
  - Still has idles when switching threads
- **Fine MT gives good performance …**
  - … if lots of commands are issued on each cycle
- **SMT: one thread active on each cycle**
  - Dynamic scheduler allows instructions from different threads to be executed
  - Times returns to line up with its thread's execution

Hardware multithreading

- **Available on many modern processors**

- **Core i7**
  - Four processor cores
  - Each with own L1 and L2 caches
  - Each capable of two concurrent threads

  - Shared L3 cache
    - Interesting caching issues!

Questions?

Caching Issues

No problem for threads in same core

All threads see same data in all caches

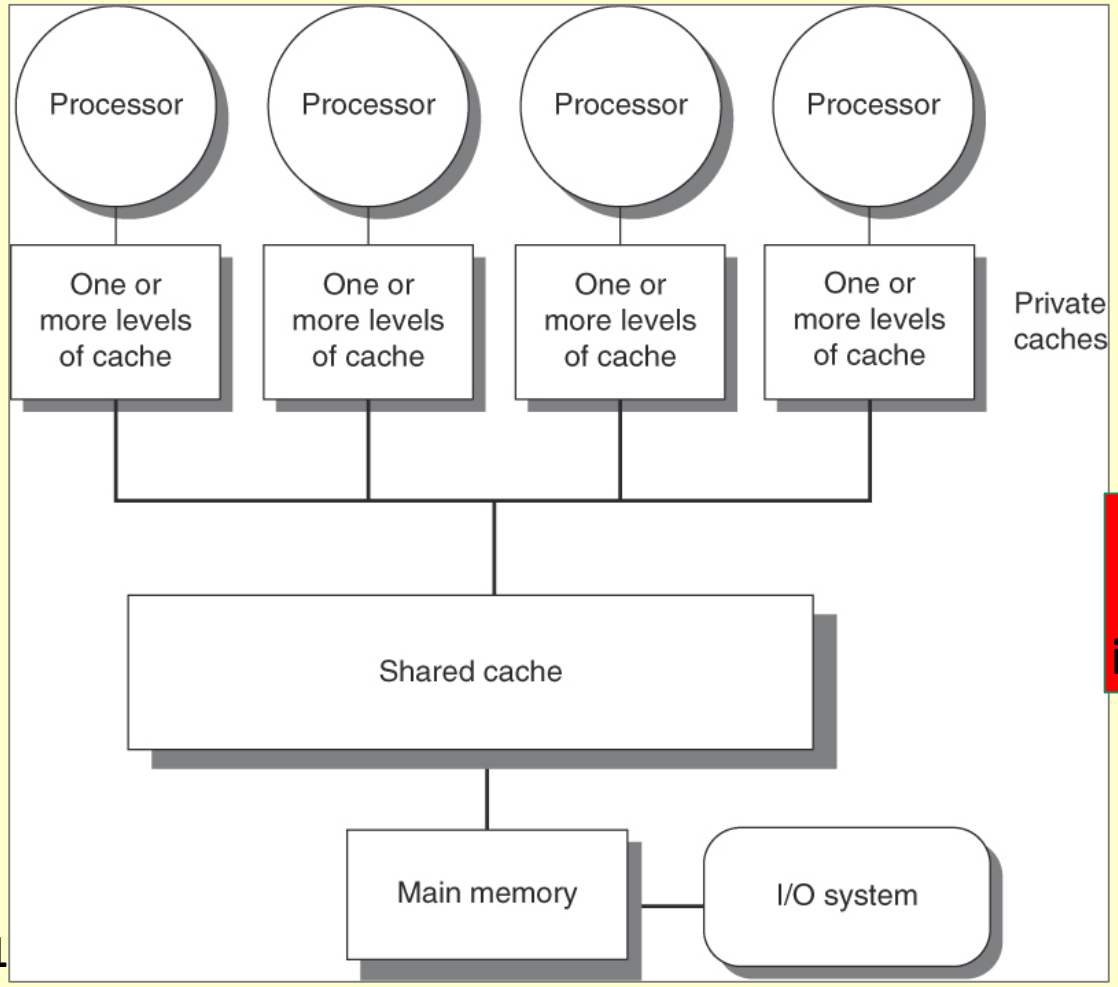

Fig 5.1

Serious problem for threads in other cores!

Threads may see inconsistent data in separate caches

Long-understood problem

- **Large multiprocessor systems**
  - Corporate data centers
  - "Cluster" computers
- **Need for synchronized memory among processors of cluster**

Cache Coherency Methods

Snooping

- **Good for small number of cores**
- **CPU watches shared bus**
  - Detects writes to shared data

Directories

- **Scales better for larger CPU count**
- **Shared cache tracks references to blocks**
  - Notifies affected CPUs on changes

Snooping

- **Cache hardware watches shared bus for writes**
  - … to shared cache (i.e., L3)

- **Strategies:–**
  - Write Invalidate
    - Invalidate own cache of data when write is detected
  - Write Broadcast
    - Update every cache containing shared data
    - More overhead
    - Uncommon

Snooping (continued)

- **States for each cache block — L2**
  - Invalid
  - Shared
    - Read-only
  - Exclusive
    - Read-write
- **Assumes write-back cache!**
- **Assumes *inclusion* property for L1-L2**
  - I.e., if a block is in L1, it must also be in L2!

Snooping

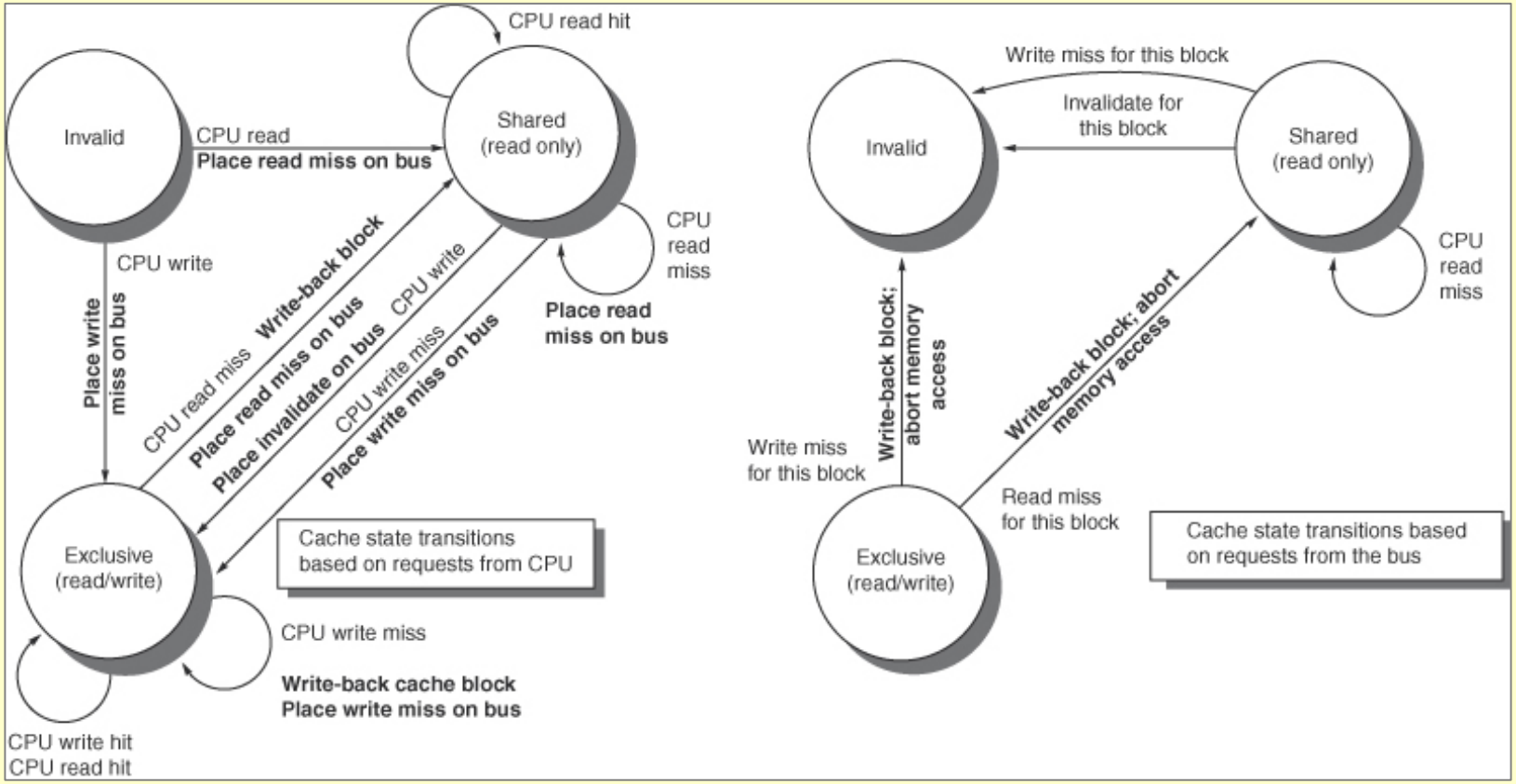

Fig 5.6

Cache Coherency (continued)

- **Much more to this topic**
- **Used to be more important in computer centers than today**
  - Many cluster systems
  - Shared operating system kernel
  - Shared code
- **Still important in multi-core processor design**

Questions?