Topic 14

Parallelism in Computer

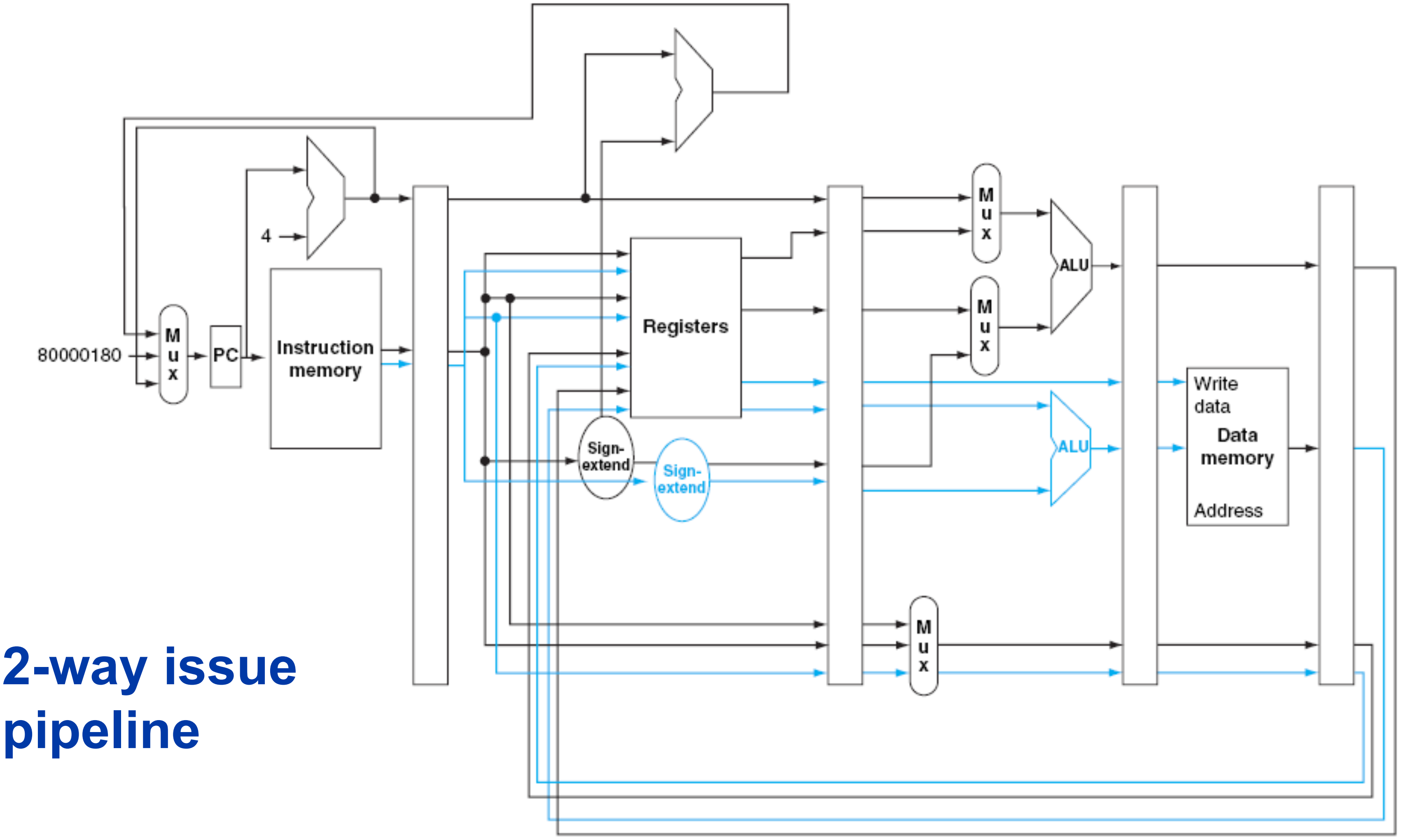
Introduction

- Computer-level parallelism
  - Goal: connecting multiple computers to get higher performance
  - Multiprocessors or multicore microprocessor
  - Major concerns: scalability, availability, power efficiency
- Instruction-level parallelism
  - Single program run on multiple processors
  - Parallel execution of instructions
- Job-level (process-level) parallelism
  - High throughput for independent jobs

Instruction-Level Parallelism (ILP)

- Executing multiple instructions in parallel
  - Typical mechanism: pipelining
- To increase ILP
  - Deeper pipeline
    - Less work per stage ⇒ shorter clock cycle, but same CPI (=1)
  - Multiple issue
    - Replicate pipeline stages ⇒ multiple pipelines
    - Start multiple instructions per clock cycle
      - CPI < 1, so use Instructions Per Cycle (IPC)
    - But instruction dependencies reduce effectiveness in practice

Multiple Issue – Hardware Support



2-way issue pipeline

Multiple Issue

- Static multiple issue (by compiler)
  - Groups instructions to be issued together
  - Detects and avoids hazards
  - Packs them into “issue slots”
  - Depends on good/smart compiler
- Dynamic multiple issue (by CPU)
  - Examines instruction stream and chooses instructions to issue each cycle
    - Compiler can help by reordering instructions
  - Resolves hazards using advanced techniques
    - Done at runtime

Speculation for Multiple Issue

- "Guess" what to do with an instruction
  - Start depending instruction earlier
  - Examples
    - Speculate on branch target (e.g. taken or not taken)
      - Roll back if path taken is different
    - Speculate on store then load (typically, load doesn't depend on store)
      - Roll back if load does depend on store
- Check whether guess was right
  - If so, complete the operation
  - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue

Software/Hardware Speculation

- Software – Compiler
  - Can reorder instructions based on speculation
  - Can include “fix-up” instructions to recover from incorrect guess
- Hardware – Processor
  - Can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

Speculation and Exceptions

- What if exception occurs due to a speculatively executed instruction?
  - Static speculation (software based speculation)
    - Can add ISA support for deferring exceptions until exceptions are validated
  - Dynamic speculation (hardware based speculation)
    - Can buffer exceptions until instruction is validated to be no longer speculative

MIPS with Static Dual Issue

- Example: two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned memory
    - ALU/branch instruction together with load/store instruction in an issue packet
    - Pad nop if necessary

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result if data dependency in the same packet

    -
      ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still need 1 stall, but now affecting more instructions
  - More aggressive scheduling required

Scheduling Example

- Schedule following for dual-issue MIPS

```
Loop: lw   $t0, 0($s1)      # $t0=array element
      addu $t0, $t0, $s2    # add scalar in $s2
      sw   $t0, 0($s1)      # store result
      addi $s1, $s1,–4      # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
`Loop:`	nop	`lw $t0, 0($s1)`	1
	`addi $s1, $s1,–4`	nop	2
	`addu $t0, $t0, $s2`	nop	3
	`bne $s1, $zero, Loop`	`sw $t0, 4($s1)`	4

- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we'd like
- Programs have real dependencies that limit ILP
- Some dependencies are hard to eliminate
- Some parallelism is hard to expose
- Memory delays and limited bandwidth
- Speculation can help if done well

Parallelism on Various Levels

- Computer-level parallelism
  - Goal: connecting multiple computers to get higher performance
  - Multiprocessors or multicore microprocessor
  - Major concerns: scalability, availability, power efficiency
- Instruction-level parallelism
  - Single program run on multiple processors
  - Parallel execution of instructions
- Job-level (process-level) parallelism
  - High throughput for independent jobs

Parallelism in Hardware and Software

- Hardware
  - Serial: e.g., Pentium 4
  - Parallel: e.g., quad-core Xeon e5345
- Software
  - Sequential: e.g., MATLAB program
  - Concurrent: e.g., Java program
- Sequential/concurrent software can run on serial/parallel hardware
  - Challenge: making effective use of parallel hardware

Parallel Programming

- Parallel hardware is not efficiently used
  - Parallel software is the challenge
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it’s easier!
- Difficulties in parallel programming
  - Partitioning
  - Coordination
  - Communications overhead

Amdahl’s Law

- Sequential part can limit speedup
- Example: 100 processors, how to achieve 90× speedup?

$$\text{Speedup} = \frac{\mathit{Time\ before}}{(\mathit{Time\ before} - \mathit{Time\ affected}) + \mathit{Time\ affected}/100}$$

$$= \frac{1}{(1 - \mathrm{F}_{\text{parallelizable}}) + \mathrm{F}_{\text{parallelizable}}/100} = 90$$

  - F: fraction of
  - Solving: $\mathrm{F}_{\text{parallelizable}} = 0.999$
- **Need sequential part to take only 0.1% of original time**

Scaling in Size & Performance

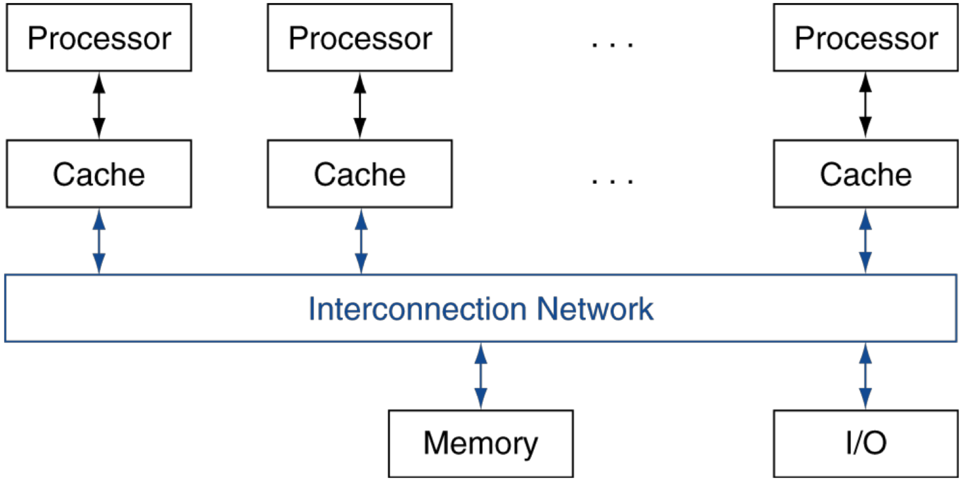
- Example: sum of 10 scalars and sum of two 10 × 10 matrixes
  - Single processor
    - 10 additions (scalar) plus 100 additions (matrix) = 110 additions
  - 10 processors (only parallel additions can benefit from multiprocessors)
    - 100 additions / 10 processors + 10 additions = 20 additions
    - Speedup = 5.5 (**55%** of potential)
  - 100 processors
    - 100 additions / 100 processors + 10 additions = 11 additions
    - Speedup = 10 (**10%** of potential, worse efficiency, although more speedup)
- What if matrix size is 100 × 100?
  - 10 processors
    - Speedup = 9.9 (**99%** of potential)
  - 100 processors
    - Speedup = 91 (**91%** of potential)
- We have assumed load balanced across processors
  - Otherwise for 100 × 100 matrix on 100 processors
    - One processor do 2% instead of 1%
      - 48% usage, assuming 10 sequential additions are done by the heavy loaded proc
      - Higher efficiency if sequential additions shifted to other procs
    - One processor do 5% instead of 1% - 20% usage

Strong vs. Weak Scaling

- Strong scaling
  - Speed-up achieved on multiprocessor with fixed size of the problem
- Weak scaling:
  - Speed-up achieved with problem size proportional to number of processors
- Strong vs. weak scaling
  - Harder to achieve good speed-up with strong scaling than with weak scaling!
  - But less applications can do weak scaling

Multiprocessor – Shared Memory

- SMP: shared memory multiprocessor
  - Single physical address space for all processors
    - Multiple virtual spaces map to the same physical memory
    - Multiple cache hierarchy – Cache Coherence problem
    - May be multi-chip on same board or multi-core in same chip
    - Running single OS
  - Communicate through shared variables in common memory
    - Variable synchronization – using locks



Cache Coherence

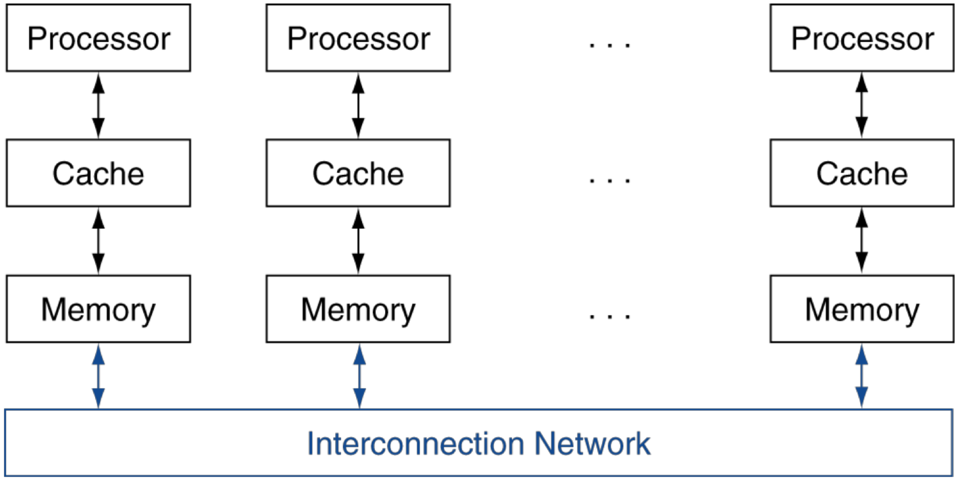
- Suppose two CPU cores share a physical address space
  - Write-through caches

Time step	Event	CPU A's cache	CPU B's cache	Memory X
0				0
1	CPU A reads X	0		0
2	CPU B reads X	0	0	0
3	CPU A writes 1 to X	1	0	1

- Cache coherence: Reads return most recently written value

Multiprocessor – Message Passing

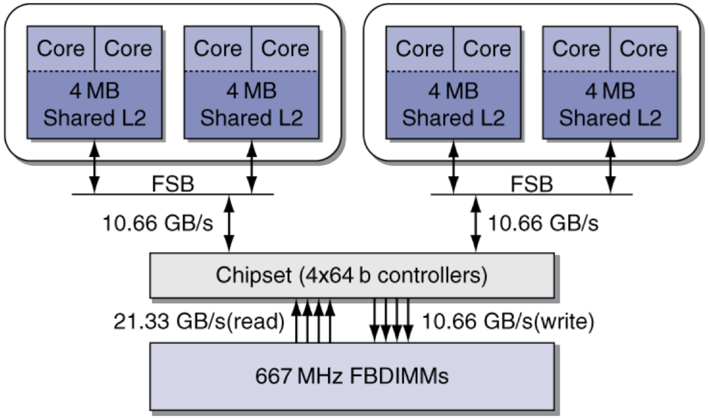
- Cluster: collection of computers/processors
  - Each processor has private physical address space
  - Each processor runs its own OS (same or different)
  - Hardware sends/receives messages between processors
    - Through LAN or special messaging network
    - Managed by their OS's



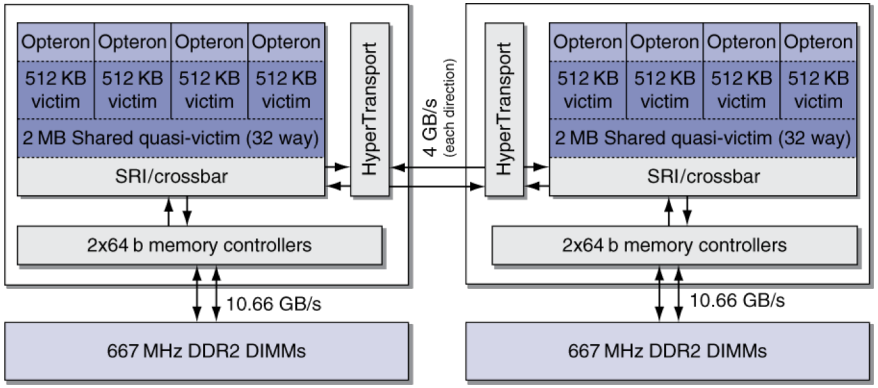
Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system
    - E.g., Ethernet/switch, Internet
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, …
- Pros
  - High availability (by hardware and OS redundancy)
  - Scalable (by easy connection and easy communication)
  - Affordable
- Cons
  - Administration cost
    - Administrating multiple processors (unlike SMP)
  - Low interconnect bandwidth
    - better processor/memory bandwidth on an SMP
  - OS overhead

Example Systems

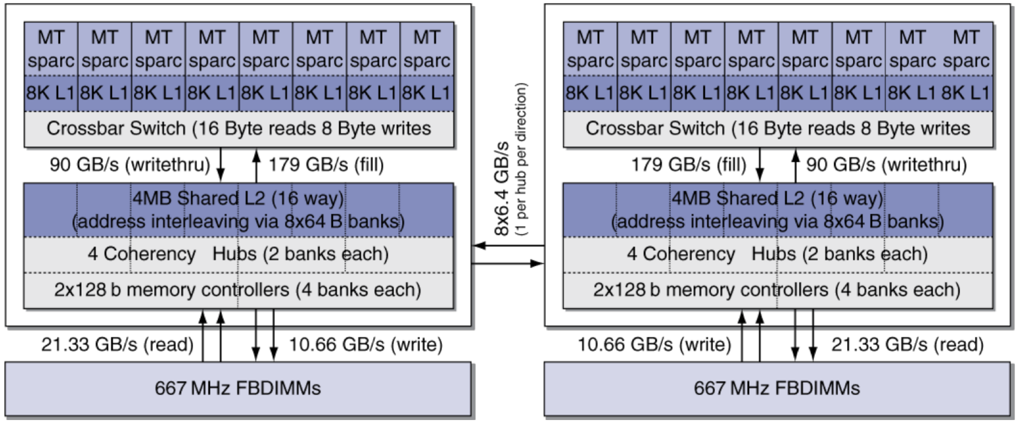


2 × quad-core
Intel Xeon e5345
(Clovertown)

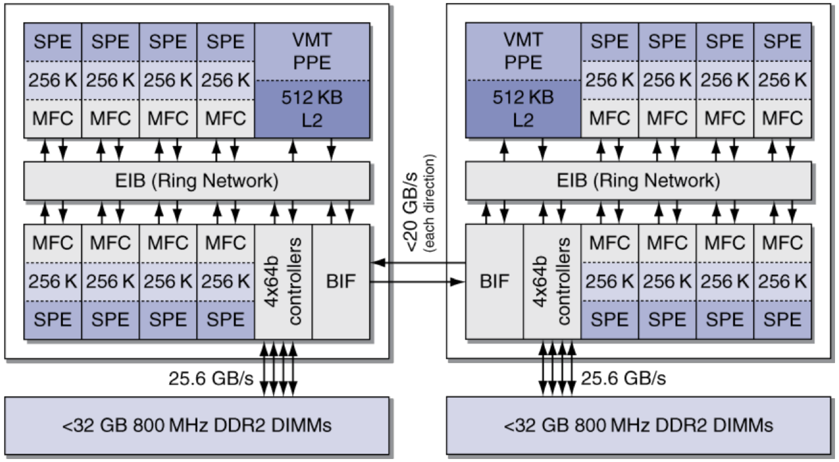


2 × quad-core
AMD Opteron X4 2356
(Barcelona)

Example Systems


MT sparc
8K L1
Crossbar Switch (16 Byte reads 8 Byte writes
90 GB/s (writethru)
179 GB/s (fill)
4MB Shared L2 (16 way)
(address interleaving via 8x64 B banks)
4 Coherency Hubs (2 banks each)
2x128 b memory controllers (4 banks each)
21.33 GB/s (read)
10.66 GB/s (write)
667 MHz FBDIMMs
8x6.4 GB/s
(1 per hub per direction)

2 × oct-core
Sun UltraSPARC
T2 5140 (Niagara 2)



2 × oct-core
IBM Cell QS20

Concluding Remarks

- Goal: higher performance by using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- Many reasons for optimism
  - Changing software and application environment
  - Chip-level multiprocessors with lower latency, higher bandwidth interconnect
- An ongoing challenge for computer architects!