Advanced Processors

Instruction-Level Parallelism (ILP)
Multiple Issue

Fetch multiple (e.g. 2, 4) instructions in parallel, and then consider how many and which of them to execute in parallel

- Static multiple issue
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

- Dynamic multiple issue
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

where none independent available, fills-in noop’s
Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)
Scheduling Static Multiple Issue

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary
a = b + c;
e = b - f;

The more things you have 'up in the air' (in parallel), the more temporary registers you need in order to 'name' those 'pending' values.

Does the compiler know for sure if i!=j (OK to reorder sd−ld) or i==j (fwd in reg.)?
If unknown to compiler, static sch. impossible => dynamic scheduling at runtime (ooo pipe)

What if the program is?:
RAW dependence?

This is 'Static' Scheduling, at Compile Time

- Does the compiler know for sure if i!=j (OK to reorder sd−ld) or i==j (fwd in reg.)?
- If unknown to compiler, static sch. impossible => dynamic scheduling at runtime (ooo pipe)
Dynamic Multiple Issue

- “Superscalar” processors
  - checks dependencies and
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Allows executables to run on newer processors, with same ISA but different pipeline, without needing to be recompiled
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example
  
  ```
  ld   x31, 20(x21)
  add  x1, x31, x2
  sub  x23, x23, x3
  andi x5, x23, 20
  
  Can start sub while add is waiting for ld
  ```

Out-of-Order (ooo) Execution

In-Order Commit

(so as to flush results of mis-speculated instructions, and also allow precise exceptions)
Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards
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
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well
Parallelism
Pitfall: Amdahl’s Law

- Improving an aspect of a computer and expecting a proportional improvement in overall performance

\[ T_{improved} = \frac{T_{affected}}{\text{improvement factor}} + T_{unaffected} \]

- Example: multiply accounts for 80s/100s
  - How much improvement in multiply performance to get 5× overall?
  
  \[
  20 = \frac{80}{n} + 20
  \]

- Can’t be done!

- Corollary: make the common case fast

80 seconds out of total 100 seconds

Eπιτάχυνση ενός μέρους μόνον του προγράμματος και όχι ολόκληρου

§1.10 Fallacies and Pitfalls
Scaling Example

- Workload: sum of 10 scalars, and $10 \times 10$ matrix sum
  - Speed up from 10 to 100 processors
- Single processor: Time = $(10 + 100) \times t_{add}$
- 10 processors
  - Time = $10 \times t_{add} + \frac{100}{10} \times t_{add} = 20 \times t_{add}$
  - Speedup = $\frac{110}{20} = 5.5$ (55% of potential)
- 100 processors
  - Time = $10 \times t_{add} + \frac{100}{100} \times t_{add} = 11 \times t_{add}$
  - Speedup = $\frac{110}{11} = 10$ (10% of potential)
- Assumes load can be balanced across processors
Scaling Example (cont)

- What if matrix size is $100 \times 100$?
- Single processor: $\text{Time} = (10 + 10000) \times t_{\text{add}}$
- 10 processors
  - $\text{Time} = 10 \times t_{\text{add}} + \frac{10000}{10} \times t_{\text{add}} = 1010 \times t_{\text{add}}$
  - $\text{Speedup} = \frac{10010}{1010} = 9.9$ (99% of potential)
- 100 processors
  - $\text{Time} = 10 \times t_{\text{add}} + \frac{10000}{100} \times t_{\text{add}} = 110 \times t_{\text{add}}$
  - $\text{Speedup} = \frac{10010}{110} = 91$ (91% of potential)
- Assuming load balanced
Strong vs Weak Scaling

- **Strong scaling**: problem size fixed
  - As in example

- **Weak scaling**: problem size proportional to number of processors
  - 10 processors, $10 \times 10$ matrix
    - Time = $20 \times t_{\text{add}}$
  - 100 processors, $32 \times 32$ matrix
    - Time = $10 \times t_{\text{add}} + \frac{1000}{100} \times t_{\text{add}} = 20 \times t_{\text{add}}$
  - Constant performance in this example
## Instruction and Data Streams

### An alternate classification

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td><strong>SISD:</strong> Intel Pentium 4</td>
<td>SIMD: SSE instructions of x86</td>
<td></td>
</tr>
<tr>
<td>Multiple</td>
<td><strong>MISD:</strong> No examples today</td>
<td>MIMD: Intel Xeon e5345</td>
<td></td>
</tr>
</tbody>
</table>

### SPMD: Single Program Multiple Data
- A parallel program on a MIMD computer
- Conditional code for different processors
SIMD

- Operate elementwise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications
Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to RISC-V
  - v0 to v31: 32 × 64-element registers, (64-bit elements)
- Vector instructions
  - `f1d.v, fsd.v`: load/store vector
  - `fadd.d.v`: add vectors of double
  - `fadd.d.vs`: add scalar to each element of vector of double

Significantly reduces instruction-fetch bandwidth

Data-Level Parallelism
Identical & Independent operations on all elements of a vector (array) - one vector instr. replaces a loop

Chapter 6 — Parallel Processors from Client to Cloud — 11
Vector vs. Scalar

- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology
Vector vs. Multimedia Extensions

- Vector instructions have a variable vector width, multimedia extensions have a fixed width.
- Vector instructions support strided access, multimedia extensions do not.
- Vector units can be combination of pipelined and arrayed functional units.
Multithreading

Performing multiple threads of execution in parallel
- Replicate registers, PC, etc.
- Fast switching between threads

Fine-grain multithreading
- Switch threads after each cycle
- Interleave instruction execution
- If one thread stalls, others are executed

Coarse-grain multithreading
- Only switch on long stall (e.g., L2-cache miss)
- Simplifies hardware, but doesn’t hide short stalls (e.g., data hazards)

One "thread of control" = one (traditional) sequential program. Multiple threads = parallel program.

§ 6.4 Hardware Multithreading

Chapter 6 — Parallel Processors from Client to Cloud — 16
Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when function units are available
  - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT
  - Two threads: duplicated registers, shared function units and caches
Multithreading Example

Cache miss stalls are major concern; coarse MT targets them especially.

longer latency among instructions of a same thread relieves dependencies, but slows down each individual thread.
Future of Multithreading

- Will it survive? In what form?
- Power considerations $\Rightarrow$ simplified microarchitectures
  - Simpler forms of multithreading
- Tolerating cache-miss latency
  - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Two different threads may have two different working sets of data/instructions; is it better to place them in a single cache, or in two different caches as two separate cores would do?
GPU Architectures

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth

- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code

- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)
Example: NVIDIA Fermi

- Multiple SIMD processors, each as shown:

![Diagram showing NVIDIA Fermi architecture](image)

- Instruction register
- SIMD Lanes (Thread Processors)
  - 16 lanes
  - 32-element SIMD instructions
  - Also, massively multithreaded

Chapter 6 — Parallel Processors from Client to Cloud — 26
Example: NVIDIA Fermi

- SIMD Processor: 16 SIMD lanes
- SIMD instruction
  - Operates on 32 element wide threads
  - Dynamically scheduled on 16-wide processor over 2 cycles
- 32K x 32-bit registers spread across lanes
  - 64 registers per thread context
GPU Memory Structures

CUDA Thread

Per-CUDA Thread Private Memory

Thread block

Per-Block Local Memory

Grid 0

Inter-Grid Synchronization

Grid 1

Sequence

GPU Memory
Message Passing

- Each processor has private physical address space
- Hardware sends/receives messages between processors
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, …
- High availability, scalable, affordable
- Problems
  - Administration cost (prefer virtual machines)
  - Low interconnect bandwidth
    - c.f. processor/memory bandwidth on an SMP