Lecture 11: SMT and Caching Basics

• Today: SMT, cache access basics (Sections 3.5, 5.1)
Thread-Level Parallelism

• Motivation:
  ➢ a single thread leaves a processor under-utilized for most of the time
  ➢ by doubling processor area, single thread performance barely improves

• Strategies for thread-level parallelism:
  ➢ multiple threads share the same large processor → reduces under-utilization, efficient resource allocation
    Simultaneous Multi-Threading (SMT)
  ➢ each thread executes on its own mini processor → simple design, low interference between threads
    Chip Multi-Processing (CMP)
How are Resources Shared?

Each box represents an issue slot for a functional unit. Peak thruput is 4 IPC.

- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Fine-grained multithreading can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- 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 (in other words, a new thread can start without a context switch)

• For correctness, each thread needs its own PC, its own logical regs (and its own mapping from logical to phys regs)

• For performance, each thread could have its own ROB (so that a stall in one thread does not stall commit in other threads), I-cache, branch predictor, D-cache, etc. (for low interference), although note that more sharing $\rightarrow$ better utilization of resources

• Each additional thread costs a PC, rename table, and ROB – cheap!
Pipeline Structure

What about RAS, LSQ?
Resource Sharing

Thread-1

Instr Fetch

R1 ← R1 + R2
R3 ← R1 + R4
R5 ← R1 + R3

Instr Rename

P73 ← P1 + P2
P74 ← P73 + P4
P75 ← P73 + P74

Thread-2

Instr Fetch

R2 ← R1 + R2
R5 ← R1 + R2
R3 ← R5 + R3

Instr Rename

P76 ← P33 + P34
P77 ← P33 + P76
P78 ← P77 + P35

Issue Queue

P73 ← P1 + P2
P74 ← P73 + P4
P75 ← P73 + P74
P76 ← P33 + P34
P77 ← P33 + P76
P78 ← P77 + P35

Register File

FU

FU

FU

FU
Performance Implications of SMT

- Single thread performance is likely to go down (caches, branch predictors, registers, etc. are shared) – this effect can be mitigated by trying to prioritize one thread

- While fetching instructions, thread priority can dramatically influence total throughput – a widely accepted heuristic (ICOUNT): fetch such that each thread has an equal share of processor resources

- With eight threads in a processor with many resources, SMT yields throughput improvements of roughly 2-4

- Alpha 21464 and Intel Pentium 4 are examples of SMT
Pentium4 Hyper-Threading

- Two threads – the Linux operating system operates as if it is executing on a two-processor system

- When there is only one available thread, it behaves like a regular single-threaded superscalar processor

- Statically divided resources: ROB, LSQ, issueq -- a slow thread will not cripple throughput (might not scale)

- Dynamically shared: trace cache and decode (fine-grained multi-threaded, round-robin), FUs, data cache, bpred
Multi-Programmed Speedup

- sixtrack and eon do not degrade their partners (small working sets?)
- swim and art degrade their partners (cache contention?)
- Best combination: swim & sixtrack
- worst combination: swim & art
- Static partitioning ensures low interference – worst slowdown is 0.9

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Best Speedup</th>
<th>Worst Speedup</th>
<th>Avg Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>1.48</td>
<td>1.14</td>
<td>1.24</td>
</tr>
<tr>
<td>vpr</td>
<td>1.43</td>
<td>1.04</td>
<td>1.17</td>
</tr>
<tr>
<td>gcc</td>
<td>1.44</td>
<td>1.00</td>
<td>1.11</td>
</tr>
<tr>
<td>mcf</td>
<td>1.57</td>
<td>1.01</td>
<td>1.21</td>
</tr>
<tr>
<td>crafty</td>
<td>1.40</td>
<td>0.99</td>
<td>1.17</td>
</tr>
<tr>
<td>parser</td>
<td>1.44</td>
<td>1.09</td>
<td>1.18</td>
</tr>
<tr>
<td>eon</td>
<td>1.42</td>
<td>1.07</td>
<td>1.20</td>
</tr>
<tr>
<td>perlbmk</td>
<td>1.40</td>
<td>1.07</td>
<td>1.20</td>
</tr>
<tr>
<td>gap</td>
<td>1.43</td>
<td>1.17</td>
<td>1.25</td>
</tr>
<tr>
<td>vortex</td>
<td>1.41</td>
<td>1.01</td>
<td>1.13</td>
</tr>
<tr>
<td>bzip2</td>
<td>1.47</td>
<td>1.15</td>
<td>1.24</td>
</tr>
<tr>
<td>twolf</td>
<td>1.48</td>
<td>1.02</td>
<td>1.16</td>
</tr>
<tr>
<td>wupwise</td>
<td>1.33</td>
<td>1.12</td>
<td>1.24</td>
</tr>
<tr>
<td>swim</td>
<td>1.58</td>
<td>0.90</td>
<td>1.13</td>
</tr>
<tr>
<td>mgrid</td>
<td>1.28</td>
<td>0.94</td>
<td>1.10</td>
</tr>
<tr>
<td>applu</td>
<td>1.37</td>
<td>1.02</td>
<td>1.16</td>
</tr>
<tr>
<td>mesa</td>
<td>1.39</td>
<td>1.11</td>
<td>1.22</td>
</tr>
<tr>
<td>galgel</td>
<td>1.47</td>
<td>1.05</td>
<td>1.25</td>
</tr>
<tr>
<td>art</td>
<td>1.55</td>
<td>0.90</td>
<td>1.13</td>
</tr>
<tr>
<td>equake</td>
<td>1.48</td>
<td>1.02</td>
<td>1.21</td>
</tr>
<tr>
<td>facerec</td>
<td>1.39</td>
<td>1.16</td>
<td>1.25</td>
</tr>
<tr>
<td>ammp</td>
<td>1.40</td>
<td>1.09</td>
<td>1.21</td>
</tr>
<tr>
<td>lucas</td>
<td>1.36</td>
<td>0.97</td>
<td>1.13</td>
</tr>
<tr>
<td>fma3d</td>
<td>1.34</td>
<td>1.13</td>
<td>1.20</td>
</tr>
<tr>
<td>sixtrack</td>
<td>1.58</td>
<td>1.28</td>
<td>1.42</td>
</tr>
<tr>
<td>apsi</td>
<td>1.40</td>
<td>1.14</td>
<td>1.23</td>
</tr>
<tr>
<td>Overall</td>
<td>1.58</td>
<td>0.90</td>
<td>1.20</td>
</tr>
</tbody>
</table>
Memory Hierarchy

- As you go further, capacity and latency increase

- Registers: 1KB, 1 cycle
- L1 data or instruction cache: 32KB, 2 cycles
- L2 cache: 2MB, 15 cycles
- Memory: 1GB, 300 cycles
- Disk: 80 GB, 10M cycles
Accessing the Cache

Direct-mapped cache: each address maps to a unique address.

Byte address

101000

Offset

Data array

8-byte words

Sets

8 words: 3 index bits
The Tag Array

Direct-mapped cache: each address maps to a unique address
Increasing Line Size

A large cache line size → smaller tag array, fewer misses because of spatial locality

Tag array

Byte address

Tag

Offset

10100000

Data array

32-byte cache line size or block size
Associativity

Set associativity → fewer conflicts; wasted power because multiple data and tags are read

Tag array

Byte address

10100000

Tag

Compare

Way-1

Way-2

Data array
Example

- 32 KB 4-way set-associative data cache array with 32 byte line sizes
- How many sets?
- How many index bits, offset bits, tag bits?
- How large is the tag array?
Cache Misses

- On a write miss, you may either choose to bring the block into the cache (write-allocate) or not (write-no-allocate).

- On a read miss, you always bring the block in (spatial and temporal locality) – but which block do you replace?
  - no choice for a direct-mapped cache
  - randomly pick one of the ways to replace
  - replace the way that was least-recently used (LRU)
  - FIFO replacement (round-robin)
Writes

• When you write into a block, do you also update the copy in L2?
  - write-through: every write to L1 → write to L2
  - write-back: mark the block as dirty, when the block gets replaced from L1, write it to L2

• Writeback coalesces multiple writes to an L1 block into one L2 write

• Writethrough simplifies coherency protocols in a multiprocessor system as the L2 always has a current copy of data
Title

• Bullet