Lecture: SMT, Cache Hierarchies

• Topics: memory dependence wrap-up, SMT processors, cache access basics and innovations (Sections B.1-B.3, 2.1)
Problem 0

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume memory dependence prediction, with a default prediction that there is no dependence.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1</td>
<td>[R2]</td>
<td>3</td>
<td>abcd</td>
<td></td>
</tr>
<tr>
<td>LD R3</td>
<td>[R4]</td>
<td>6</td>
<td>adde</td>
<td></td>
</tr>
<tr>
<td>ST R5</td>
<td>[R6]</td>
<td>4</td>
<td>7</td>
<td>abba</td>
</tr>
<tr>
<td>LD R7</td>
<td>[R8]</td>
<td>2</td>
<td>abce</td>
<td></td>
</tr>
<tr>
<td>ST R9</td>
<td>[R10]</td>
<td>8</td>
<td>3</td>
<td>abba</td>
</tr>
<tr>
<td>LD R11</td>
<td>[R12]</td>
<td>1</td>
<td>abba</td>
<td></td>
</tr>
</tbody>
</table>
Problem 0

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume memory dependence prediction, with a default prediction that there is no dependence.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>R1 ← [R2]</td>
<td>3</td>
<td>abcd</td>
<td>4</td>
</tr>
<tr>
<td>LD</td>
<td>R3 ← [R4]</td>
<td>6</td>
<td>adde</td>
<td>7</td>
</tr>
<tr>
<td>ST</td>
<td>R5 → [R6]</td>
<td>4</td>
<td>7</td>
<td>abba</td>
</tr>
<tr>
<td>LD</td>
<td>R7 ← [R8]</td>
<td>2</td>
<td>abce</td>
<td>3</td>
</tr>
<tr>
<td>ST</td>
<td>R9 → [R10]</td>
<td>8</td>
<td>3</td>
<td>abba</td>
</tr>
<tr>
<td>LD</td>
<td>R11 ← [R12]</td>
<td>1</td>
<td>abba</td>
<td>2</td>
</tr>
</tbody>
</table>
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-Threaded (SMT)
  ➢ each thread executes on its own mini processor → simple design, low interference between threads
    Chip Multi-Processing (CMP) or multi-core
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, IFQ, logical regs (and its own mappings from logical to phys regs)

• For performance, each thread could have its own ROB/LSQ (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 → better utilization of resources

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

- Front End
- Front End
- Front End
- Front End
- Execution Engine
- Private/Shared Front-end
- rename
- I-Cache
- Bpred
- Private Front-end
- Rename
- ROB
- Regs
- IQ
- DCache
- FUs
- Shared Exec Engine
### Resource Sharing

#### Thread-1

<table>
<thead>
<tr>
<th>Register</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R1 + R2</td>
</tr>
<tr>
<td>R3</td>
<td>R1 + R4</td>
</tr>
<tr>
<td>R5</td>
<td>R1 + R3</td>
</tr>
</tbody>
</table>

#### Instr Fetch

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

#### Instr Rename

- P65 ← P1 + P2
- P66 ← P65 + P4
- P67 ← P65 + P66

#### Thread-2

<table>
<thead>
<tr>
<th>Register</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>R1 + R2</td>
</tr>
<tr>
<td>R5</td>
<td>R1 + R2</td>
</tr>
<tr>
<td>R3</td>
<td>R5 + R3</td>
</tr>
</tbody>
</table>

#### Instr Fetch

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

#### Instr Rename

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

#### Issue Queue

- P65 ← P1 + P2
- P66 ← P65 + P4
- P67 ← P65 + P66
- P76 ← P33 + P34
- P77 ← P33 + P76
- P78 ← P77 + P35

#### Register File

- FU
- FU
- FU
- FU

---

8
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
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
The Cache Hierarchy
Problem 1

• Memory access time: Assume a program that has cache access times of 1-cyc (L1), 10-cyc (L2), 30-cyc (L3), and 300-cyc (memory), and MPKIs of 20 (L1), 10 (L2), and 5 (L3). Should you get rid of the L3?
Problem 1

• Memory access time: Assume a program that has cache access times of 1-cyc (L1), 10-cyc (L2), 30-cyc (L3), and 300-cyc (memory), and MPKIs of 20 (L1), 10 (L2), and 5 (L3). Should you get rid of the L3?

With L3: $1000 + 10 \times 20 + 30 \times 10 + 300 \times 5 = 3000$
Without L3: $1000 + 10 \times 20 + 10 \times 300 = 4200$
Accessing the Cache

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

Byte address

8 words: 3 index bits

Offset

Data array

Sets

8-byte words
The Tag Array

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

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

32-byte cache line size or block size

Tag array

Data array
**Associativity**

Set associativity $\rightarrow$ fewer conflicts; wasted power because multiple data and tags are read.
Problem 2

• Assume a direct-mapped cache with just 4 sets. Assume that block A maps to set 0, B to 1, C to 2, D to 3, E to 0, and so on. For the following access pattern, estimate the hits and misses:

A B B E C C A D B F A E G C G A
Problem 2

- Assume a direct-mapped cache with just 4 sets. Assume that block A maps to set 0, B to 1, C to 2, D to 3, E to 0, and so on. For the following access pattern, estimate the hits and misses:

```
A B B E C C A D B F A E G C G A
M MH MM H MM HM HMM M M M
```
Problem 3

• Assume a 2-way set-associative cache with just 2 sets. Assume that block A maps to set 0, B to 1, C to 0, D to 1, E to 0, and so on. For the following access pattern, estimate the hits and misses:

A B B E C C A D B F A E G C G A
Problem 3

• Assume a 2-way set-associative cache with just 2 sets. Assume that block A maps to set 0, B to 1, C to 0, D to 1, E to 0, and so on. For the following access pattern, estimate the hits and misses:

A B B E C C A D B F A E G C G A
M MH M MH MM HM HMM M H M
Problem 4

• 64 KB 16-way set-associative data cache array with 64 byte line sizes, assume a 40-bit address

• How many sets?

• How many index bits, offset bits, tag bits?

• How large is the tag array?
Problem 4

- 64 KB 16-way set-associative data cache array with 64 byte line sizes, assume a 40-bit address

- How many sets? 64

- How many index bits (6), offset bits (6), tag bits (28)?

- How large is the tag array (28 Kb)?
Title

• Bullet