Multithreaded processors

Hung-Wei Tseng
Simultaneous Multi-Threading (SMT)
Simultaneous Multi-Threading (SMT)

- Fetch instructions from different threads/processes to fill the not utilized part of pipeline
- Exploit “thread level parallelism” (TLP) to solve the problem of insufficient ILP in a single thread
- Keep separate architectural states for each thread
  - PC
  - Register Files
  - Reorder Buffer
- Create an illusion of multiple processors for OSs
- The rest of superscalar processor hardware is shared
- Invented by Dean Tullsen
  - Now a professor in UCSD CSE!
  - You may take his CSE148 in Spring 2015
Simplified SMT-OOO pipeline
Simultaneous Multi-Threading (SMT)

- Fetch 2 instructions from each thread/process at each cycle to fill the not utilized part of pipeline
- Issue width is still 2, commit width is still 4

T1 1: lw $t1, 0($a0)
T1 2: lw $a0, 0($t1)
T2 1: sll $t0, $a1, 2
T2 2: add $t1, $a0, $t0
T1 3: addi $a1, $a1, -1
T1 4: bne $a1, $zero, LOOP
T2 3: lw $v0, 0($t1)
T2 4: addi $t1, $t1, 4
T2 5: add $v0, $v0, $t2
T2 6: jr $ra

Can execute 6 instructions before bne resolved.
SMT

- Improve the throughput of execution
  - May increase the latency of a single thread
- Less branch penalty per thread
- Increase hardware utilization
- Simple hardware design: Only need to duplicate PC/Register Files
- Real Case:
  - Intel HyperThreading (supports up to two threads)
    - Intel Pentium 4, Intel Atom, Intel Core i7
  - AMD FX, part of A series
Simultaneous Multithreading

- SMT helps covering the long memory latency problem
- But SMT is still a “superscalar” processor
- Power consumption / hardware complexity can still be high.
  - Think about Pentium 4
Chip multiprocessor (CMP)
Chip Multiprocessor (CMP)

- Multiple processors on a single die!
- Increase the frequency: increase power consumption by cubic!
  - Doubling frequency increases power by 8x, doubling cores increases power by 2x
- But the process technology (Moore’s law) allows us to cram more core into a single chip!
- Instead of building a wide issue processor, we can have multiple narrower issue processor.
  - e.g. 4-issue v.s. 2x 2-issue processor
- Now common place
- Improve the throughput of applications
Speedup a single application on multithreaded processors
Parallel programming

• The only way we can improve a single application performance on CMP/SMT.

• Parallel programming is difficult!
  • Data sharing among threads
  • Threads are hard to find
  • Hard to debug!
    • Locks!
    • Deadlock
Shared memory

• Provide a single physical memory space that all processors can share
• All threads within the same program shares the same address space.
• Threads communicate with each other using shared variables in memory
• Provide the same memory abstraction as single-thread programming
Simple idea...

- Connecting all processor and shared memory to a bus.
- Processor speed will be slow b/c all devices on a bus must run at the same speed.

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
</table>

Bus

Shared $
Memory hierarchy on CMP

- Each processor has its own local cache
Cache on Multiprocessor

• Coherency
  • Guarantees all processors see the same value for a variable/memory address in the system when the processors need the value at the same time
    • What value should be seen

• Consistency
  • All threads see the change of data in the same order
    • When the memory operation should be done
Simple cache coherency protocol

• Snooping protocol
  • Each processor broadcasts / listens to cache misses

• State associate with each block (cacheline)
  • Invalid
    • The data in the current block is invalid
  • Shared
    • The processor can read the data
    • The data may also exist on other processors
  • Exclusive
    • The processor has full permission on the data
    • The processor is the only one that has up-to-date data
Simple cache coherency protocol

Invalid

Shared

Exclusive

read/write miss (bus)

write miss (bus)
write miss (processor)
write request (processor)

write hit

write back data

read miss (bus)

write back data

read miss (hit)
Cache coherency practice

- What happens when core 0 modifies 0x1000?, which belongs to the same cache block as 0x1000?

![Cache Coherency Diagram]

- Core 0
- Core 1
- Core 2
- Core 3
- Shared $ 0x1000
- Local $ 0x1000
- Write miss 0x1000
- Bus
- Shared $
Cache coherency practice

- Then, what happens when core 2 reads 0x1000?
It’s show time!

• Demo!

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>while(1)</td>
<td>while(1)</td>
</tr>
<tr>
<td>printf(&quot;%d &quot;,a);</td>
<td>a++;</td>
</tr>
</tbody>
</table>

Cache coherency practice

- Now, what happens when core 2 writes 0x1004, which belongs the same block as 0x1000?
- Then, if Core 0 accesses 0x1000, it will be a miss!
4C model

- **3Cs:**
  - Compulsory, Conflict, Capacity

- **Coherency miss:**
  - A “block” invalidated because of the sharing among processors.
    - True sharing
      - Processor A modifies X, processor B also want to access X.
    - False Sharing
      - Processor A modifies X, processor B also want to access Y. However, Y is invalidated because X and Y are in the same block!
Threads are hard to find

- To exploit CMP parallelism you need multiple processes or multiple “threads”
- Processes
  - Separate programs actually running (not sitting idle) on your computer at the same time.
  - Common in servers
  - Much less common in desktop/laptops
- Threads
  - Independent portions of your program that can run in parallel
  - Most programs are not multi-threaded.
- We will refer to these collectively as “threads”
  - A typical user system might have 1-8 actively running threads.
Hard to debug

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>int loop;</td>
<td>void* modifyloop(void *x)</td>
</tr>
<tr>
<td>int main()</td>
<td>{</td>
</tr>
<tr>
<td>{</td>
<td>sleep(1);</td>
</tr>
<tr>
<td>pthread_t thread;</td>
<td>loop = 0;</td>
</tr>
<tr>
<td>loop = 1;</td>
<td>return NULL;</td>
</tr>
<tr>
<td>pthread_create(&amp;thread, NULL, modifyloop, NULL);</td>
<td></td>
</tr>
<tr>
<td>pthread_join(&amp;thread, NULL);</td>
<td></td>
</tr>
<tr>
<td>while(loop)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>continue;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>printf(stderr,&quot;finished\n&quot;);</td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
</tr>
</tbody>
</table>
Q & A