Multithreaded processors

Hung-Wei Tseng
Announcement

• CAPE (Course Evaluation)
• Special office hour: 10a-12p this Friday @ 2128
• Final exam:
  • 9/5 (Fri.) 3p-6p here
  • Cumulative.
  • Consists of multiple choices, short answer questions, calculations, open-ended questions.
  • No open book, no cheat sheet allowed
  • Small pizza party afterwards
Outline

- Recap: SuperScalar + OOO
- Simultaneous multithreading
- Chip multiprocessor
- Parallel programming
SuperScalar + OOO
Limitation of compiler optimizations

- Compiler can only optimize “static instructions”
- The left-hand side in the table
- Compiler cannot re-order 2, 5 and 4,5
  - Hardware can do this with branch prediction
- Compiler can generate false dependencies due to limited registers
- Instructions 1, 3 do not depend on each other

<table>
<thead>
<tr>
<th>static instructions</th>
<th>dynamic instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP: lw $t1, 0($a0)</td>
<td>1: lw $t1, 0($a0)</td>
</tr>
<tr>
<td>add $v0, $v0, $t1</td>
<td>2: add $v0, $v0, $t1</td>
</tr>
<tr>
<td>addi $a0, $a0, 4</td>
<td>3: addi $a0, $a0, 4</td>
</tr>
<tr>
<td>bne $a0, $t0, LOOP</td>
<td>4: bne $a0, $t0, LOOP</td>
</tr>
<tr>
<td>lw $t0, 0($sp)</td>
<td>5: lw $t1, 0($a0)</td>
</tr>
<tr>
<td>lw $t1, 4($sp)</td>
<td>6: add $v0, $v0, $t1</td>
</tr>
<tr>
<td></td>
<td>7: addi $a0, $a0, 4</td>
</tr>
<tr>
<td></td>
<td>8: bne $a0, $t0, LOOP</td>
</tr>
</tbody>
</table>
Register renaming

- We can remove false dependencies if we can store each new output in a different register.
- Maintain a map between “physical” and “architectural” registers.
- Architectural registers: the registers visible to compilers and programmers.
- Physical registers: the internal registers used for execution.
  - Larger number than architectural registers.
  - Modern processors have 128 physical registers.
The instruction window

Schedule

Execute
Simplified OOO pipeline

Instruction Fetch → Instruction Decode → Register renaming logic → Schedule → Execution Units → Data Cache → Reorder Buffer/Commit
Dynamic execution with register naming

- Register renaming, dynamic scheduling with 2-issue pipeline
- Assume that we fetch/decode/renaming/retire 4 instructions into/from instruction window each cycle
- If issue width increases, performance may improve without further compiler optimization

```
1: lw  $p5 , 0($p1)
2: add $p6 , $p4, $p5
3: addi $p7 , $p1, 4
4: bne $p7 , $p2, LOOP
5: lw  $p8 , 0($p7)
6: add $p9 , $p6, $p8
7: addi $p10, $p7, 4
8: bne $p10, $p2, LOOP
9: lw  $p8 , 0($p7)
10: add $p9 , $p6, $p8
11: addi $p10, $p7, 4
12: bne $p10, $p2, LOOP
```
Problems with OOO+Superscalar

- The modern OOO processors have 3-5 issue widths
- Keeping instruction window filled is hard
  - Branches are every 4-5 instructions.
  - If the instruction window is 32 instructions the processor has to predict 6-8 consecutive branches correctly to keep IW full.
- Hardware complexity & Area Efficiency
  - Multi-ported Register File $\sim$ IW$^4$
  - IQ size $\sim$ IW$^4$
  - Bypass logic $\sim$ IW$^4$
  - Wiring delay
- The ILP within an application is low
  - Usually 1-2 per thread
  - ILP is even lower if data depends on memory operations (if cache misses) or long latency operations
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
Simplified SMT-OOO pipeline

Instruction Fetch: T0 → Instruction Decode → Register renaming logic → Schedule → Execution Units → Data Cache → ROB: T0

Instruction Fetch: T1

Instruction Fetch: T2

Instruction Fetch: T3
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
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

• How many of the following about SMT are correct?
  • SMT makes deep pipelining processor more tolerable to mis-predicted branches
  • SMT can improve the throughput of a single-threaded application
  • SMT processor can better utilize hardware during cache misses comparing with superscalar processors with the same issue width
  • SMT processors can have higher cache miss rates comparing with superscalar processors with the same cache sizes when executing the same set of applications.

A. 0
B. 1
C. 2
D. 3
E. 4
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 per core)
    • Intel Pentium 4, Intel Atom, Intel Core i7
  • AMD FX, part of A series (Clustered multithreading)
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
• You may combine CMP and SMT together
  • Like Core i7
Die photo of a CMP processor
CMP advantages

- How many of the following are advantages of CMP over traditional superscalar processor
  - CMP can provide better energy-efficiency within the same area
  - CMP can deliver better instruction throughput within the same die area
  - CMP can achieve better ILP for each running thread
  - CMP can improve the performance of a single-threaded application without modifying code

A. 0
B. 1
C. 2
D. 3
E. 4
Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

Which one do you think is better?

A. P1
B. P2
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.

![Diagram showing four cores connected to a bus with shared memory](image-url)
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

- Read/write miss (bus)
- Read miss (processor)
- Write miss (bus)
- Write request (processor)
- Read miss (bus)
- Write back data
- Read miss/hit

Invalid → Exclusive → Shared → Invalid

Exclusive → Shared → Exclusive

Write hit → Invalid

Write miss (bus) → Exclusive

Write request (processor) → Shared

Write back data → Exclusive
Cache coherency practice

• What happens when core 0 modifies 0x1000?, which belongs to the same cache block as 0x1000?
Cache coherency practice

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

Local $\rightarrow$ Excl. 0x1000 $\rightarrow$ Invalid 0x1000 $\rightarrow$ Invalid 0x1000 $\rightarrow$ Invalid 0x1000

Bus $\rightarrow$ Write miss 0x1000 $\rightarrow$ Invalid 0x1000 $\rightarrow$ Invalid 0x1000 $\rightarrow$ Invalid 0x1000

Shared $\uparrow$
Cache coherency practice

- Then, what happens when core 2 reads 0x1000?
Cache coherency practice

- Then, what happens when core 2 reads 0x1000?
Cache coherency practice

- Then, what happens when core 2 reads 0x1000?
Cache coherency practice

• Then, what happens when core 2 reads 0x1000?
Cache coherency practice

- Then, what happens when core 2 reads 0x1000?
Cache coherency practice

- Then, what happens when core 2 reads 0x1000?
Cache coherency

• Assuming that we are running the following code on a CMP with some cache coherency protocol, which output is NOT possible? (a is initialized to 0)

```
while(1)
    printf("%d ",a);
while(1)
    a++;
```

<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>

A. 0 1 2 3 4 5 6 7 8 9
B. 1 2 5 9 3 6 8 10 12 13
C. 1 1 1 1 1 1 1 1 1 1
D. 1 1 1 1 1 1 1 1 1 100
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?
Cache coherency practice

• Now, what happens when core 2 writes 0x1004, which belongs the same block as 0x1000?
Cache coherency practice

- Now, what happens when core 2 writes 0x1004, which belongs the same block as 0x1000?
Cache coherency practice

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

```
Core 0
invalid 0x1000
Local $

Core 1
invalid 0x1000

Core 2
Excl. 0x1000

Core 3
invalid 0x1000

Bus
Write miss 0x1004

Shared $
```
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.
  • Servers can have more if needed (the sysadmins will hopefully configure it that way)
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>continue;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>fprintf(stderr,&quot;finished\n&quot;);</td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
</tr>
</tbody>
</table>
Q & A