Multi-threaded processors

Hung-Wei Tseng x Dean Tullsen
OoO SuperScalar Processor

• Fetch instructions in the instruction window
• Register renaming to eliminate false dependencies
• Schedule an instruction to execution stage (issue) whenever all data inputs are ready for the instruction
• Put the instruction in reorder buffer and commit the instruction if the instruction is (1) not mis-predicted and (2) all the instruction prior to this instruction are committed
INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

2.1 THE SKYLAKE MICROARCHITECTURE

The Skylake microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake microarchitecture is depicted in Figure 2-1.

The Skylake microarchitecture offers the following enhancements:

• Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
• Improved front end throughput.
• Improved branch predictor.
• Improved divider throughput and latency.
• Lower power consumption.
• Improved SMT performance with Hyper-Threading Technology.
• Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore subsystem consisting of a number of components including a ring interconnect to multiple slices of L3 (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc. A four-core configuration can be supported similar to the arrangement shown in Figure 2-3.
Dynamic execution with register naming

- Consider the following dynamic instructions

1: lw $t1, 0($a0)
2: lw $a0, 4($a0)
3: add $v0, $v0, $t1
4: bne $a0, $zero, LOOP
5: lw $t1, 0($a0)
6: lw $t2, 4($a0)
7: add $v0, $v0, $t1
8: bne $t2, $zero, LOOP

Assume a superscalar processor with unlimited issue width & physical registers that can fetch up to 4 instructions per cycle, 2 cycles to execute a memory instruction how many cycles it takes to issue all instructions?

A. 1
B. 2
C. 3
D. 4
E. 5
Outline

- Simultaneous multithreading
- Chip multiprocessor
- Parallel programming
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-Threaded (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.
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
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 Zen
SMT

• How many of the following about SMT are correct?
  • SMT makes processors with deep pipelines more tolerable to mis-predicted branches
  • SMT can improve the throughput of a single-threaded application
  • SMT processors 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
Chip multiprocessor (CMP)
A wide-issue processor or multiple narrower-issue processors

What can you do within a 21 mm * 21 mm area?

A 6-issue superscalar processor
3 integer ALUs
3 floating point ALUs
3 load/store units

4 2-issue superscalar processor
4*1 integer ALUs
4*1 floating point ALUs
4*1 load/store units

You will have more ALUs if you choose this!
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 (chip size)
  • 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
CMP v.s. SMT

- 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 multi-threaded processors
Parallel programming

- To exploit CMP/SMT parallelism you need to break your computation into multiple "processes" or multiple "threads"

Processes (in OS/software systems)
- Separate programs actually running (not sitting idle) on your computer at the same time.
- Each process will have its own virtual memory space and you need explicitly exchange data using inter-process communication APIs

Threads (in OS/software systems)
- Independent portions of your program that can run in parallel
- All threads share the same virtual memory space

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)
Create threads/processes

- The only way we can improve a single application performance on CMP/SMT
- You can use fork() to create a child process (CSE120)
- Or you can use pthread or openmp to compose multi-threaded programs

```c
/* Do matrix multiplication */
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
    tids[i] = i;
    pthread_create(&thread[i], NULL, threaded_blockmm, &tids[i]);
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
    pthread_join(thread[i], NULL);
```

Spawn a thread

Synchronize and wait for thread to terminate
Is it easy?

- Threads are hard to find and some algorithms are hard to parallelize
  - Finite State Machine
  - N-body
  - Linear programming
  - Circuit Value Problem
  - LZW data Compression
- Data sharing among threads
  - Architectural or API support
  - You need to use locks to control race conditions
- Hard to debug!
Supporting shared memory model

• Provide a single 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
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
Cache coherency practice

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

```
Core 0
Shared 0x1000
Local $
Write miss 0x1000

Core 1
Shared 0x1000

Core 2
Shared 0x1000

Core 3
Shared 0x1000

Bus

Shared $
```
Cache coherency practice

• Then, what happens when core 2 reads 0x1000?
Simple cache coherency protocol

- **Invalid**
  - write miss (processor)
  - read/write miss (bus)
  - write request (processor)
  - write hit

- **Shared**
  - read miss (processor)
  - write miss (bus)
  - read miss (bus)
  - write request (processor)
  - write back data

- **Exclusive**
  - write miss (processor)
  - write back data
  - write hit
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++;
```

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 1 100
It’s show time!

- Demo!

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
</table>
| `while(1)`  
  `printf("%d ",a);` | `while(1)`  
  `a++;` |
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!
<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>printf(&quot;Please input a number:\n&quot;);</td>
</tr>
<tr>
<td>loop = 1;</td>
<td>scanf(&quot;%d&quot;,&amp;loop);</td>
</tr>
<tr>
<td>pthread_create(&amp;thread, NULL,</td>
<td>return NULL;</td>
</tr>
<tr>
<td>modifyloop, NULL);</td>
<td>}</td>
</tr>
<tr>
<td>while(loop == 1)</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>pthread_join(thread, NULL);</td>
<td>}</td>
</tr>
<tr>
<td>fprintf(stderr,&quot;User input: %d\n&quot;,</td>
<td>}</td>
</tr>
<tr>
<td>loop);</td>
<td>}</td>
</tr>
<tr>
<td>return 0;</td>
<td>}</td>
</tr>
</tbody>
</table>
Performance of multi-threaded programs

• Multi-threaded block algorithm for matrix multiplication
• Demo!
Announcement

• CAPE (Course Evaluation)
  • Drop one more lowest quiz grade (currently 2) if the response rate is higher than 60%
• Homework #5 due next Tuesday
• Special office hour: 2p-3p this Friday @ CSE 3208
Hardware complexity of wide issue processors

- Multi-ported Register File $\sim IW^4$
- IQ size $\sim IW^4$
- Bypass logic $\sim IW^4$
- Wiring delay
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
Simple cache coherency protocol

Invalid

Shared

Exclusive

read/write miss (bus)

read miss (processor)

write miss (bus)

write request (processor)

write back data

write hit

read miss/hit

write back data