Multi-threaded processors

Hung-Wei Tseng
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
AMD K10 architecture

3-issue integer pipeline

3-issue floating point pipeline
intel Nehalem (1st gen core i7)
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: \texttt{lw} $t1, 0($a0)
  2: \texttt{lw} $a0, 4($a0)
  3: \texttt{add} $v0, $v0, $t1
  4: \texttt{bne} $a0, $\text{zero}, \text{LOOP}
  5: \texttt{lw} $t1, 0($a0)
  6: \texttt{lw} $t2, 4($a0)
  7: \texttt{add} $v0, $v0, $t1
  8: \texttt{bne} $t2, $\text{zero}, \text{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

You code looks like this when performing “linked list” traversal.

Cycle #1

Cycle #2

Cycle #3

Cycle #4

Cycle #5

Cycle #6

Cycle #7

Cycle #8
Announcement

• CAPE (Course Evaluation)
• Final review tomorrow
• Hung-Wei’s office hour
  • Thursday: 10:30a-11:30a
  • Friday: 11:00a-12:00p
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-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 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
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
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!
Chip multiprocessor (CMP)
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
- Threads from “the same process” share the same virtual memory address space (i.e. only one page table for all threads).

/* 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 a for thread to terminate
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
Simple cache coherency protocol

read/write miss (bus)

Invalid

read miss(processor)
write miss(bus)

Exclusive

write miss(processor)
write request(processor)
write back data

Shared

write miss(bus)
read miss(bus)
write back data

read miss/hit

write hit
Cache coherency practice

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

![Diagram showing cache coherency practice](image)
Cache coherency practice

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

Invalidate all 0x1000 because 0x1000 and 0x1004 belong to the same cache block!
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)

```c
while(1)
    printf("%d ",a);
    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 to 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!
Hard to debug

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>int loop;</td>
<td></td>
</tr>
<tr>
<td>int main()</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>pthread_t thread;</td>
<td></td>
</tr>
<tr>
<td>loop = 1;</td>
<td></td>
</tr>
<tr>
<td>pthread_create(&amp;thread, NULL, 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;, loop);</td>
<td></td>
</tr>
<tr>
<td>return 0;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

void* modifyloop(void *x) {
    sleep(1);
    printf("Please input a number:\n");
    scanf("%d", &loop);
    return NULL;
}
Performance of multi-threaded programs

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

• In the past, software engineers and hardware engineers work on different sides of the ISA abstraction
  • Software engineers have no idea about what happen in processors/hardware
  • Hardware engineers have no sense about what are the demands of applications
  • This works fine if we can keep accelerating CPUs, but not true anymore

• We need new execution & programming model to better utilize these hardware components
• We need innovative computer architectures to address the challenges from process technologies and the application demands