Parallelism

Introduction to Computer Architecture
David Black-Schaffer

Parallelism we’ve already seen

Contents

- Parallelism we’ve already seen
- Multicore
  - Why and what
- Parallel programming
  - Synchronization
  - Load balancing
- Multicore difficulties
  - Amdahl’s law
  - Resource sharing
- Synchronization
  - How do we synchronize in hardware?
- Cache coherency
  - How do we keep caches synchronized?
- Instruction level parallelism
  - How can we make our processor compute faster?

Parallelism in the ALU

- AND operation: 32 parallel AND gates
- Multiplier:

Parallelism in the pipeline

- Compare all tags at the same time in a fully-associative cache
  - Need n comparators

Parallelism in the cache

- Compare all tags at the same time in a fully-associative cache
  - Need n comparators
- AND operation: 32 parallel AND gates
- Multiplier:
Parallellism in VM

- Look up the address translation in the TLB in parallel with the cache.

Multicore

Why did we have parallelism?

- ALU
  - Faster to do 32 ANDs at once vs. 32 1-bit ANDs one at a time
  - Faster than serial multiplication
- Pipeline
  - Faster to do 5 instructions at once
- Cache
  - Faster to check all tags at once vs. one at a time
- TLB
  - Faster than waiting for the TLB before accessing the cache

Why do we care about parallelism?

- To make programs faster
  - Increase the clock speed (more instructions per second)
  - Do the instructions more efficiently (fewer clock cycles per instruction)
  - Do more instructions at the same time (more instructions per second)

Why not just increase the clock speed?

- Power for a transistor:
  - \( P \propto V^2 \)
  - \( V \) = Voltage for a 1
  - \( C \) = Capacitance of a transistor

- It doesn’t really help:
  - As transistors get smaller, we don’t get more power out of them
  - \( V \) is the most effective way to reduce power

- But \( V \) is not going down anymore:
  - As you get closer to the threshold voltage, leakage current (quantum tunneling that happens even when the transistor is off)

Why did we hit the power wall?

- Building better transistors
- Building better circuits
- Building more circuits
So we can’t increase clock speed for performance...

- Number of transistors (millions)
- Clock speed (GHz)
- Power (W)
- Number of cores

Why so hot? Can’t reduce voltage power goes up if we increase frequency.

The Power Wall
You expected to scale something better than this.

Result: Multicore
Need to find software’s performance somehow.

Multicore: selling you a new chip you can’t use

- Intel can’t make the clock speed faster because it will use too much power since they can’t reduce Voltage
- Need to find something useful to do with all the new transistors they get as transistors become smaller
- So how do they convince you to buy a new processor?

• Multicore!
  - 2 cores 2x faster
  - 4 cores 4x faster

Is multicore faster?

The era of multicore...

Why multicore?

• Can’t reduce Voltage...
  so power increases if we try to increase frequency

• Need to find some way to get more performance for all those extra transistors...Multicore!
  - 2 cores = 2x faster
  - 4 cores = 4x faster

• But multicore doesn’t help for most programs...

Parallel Programming
Parallel speedups

- 4 cores isn't any faster, unless:
  - Parallelization
    - Divide up the work across all the processor cores
    - Easy: for some problems, hard for most
  - Easy problem: blurring an image

Harder problem: adding up numbers

Long list of numbers to add up: do it in parallel

1. Processor 1 has to synchronize with processors 2 and 3.
2. Processor 1 has to synchronize with processors 4 and 5.
3. Processor 3 has to synchronize with processors 6 and 8.
4. Processor 0 has to synchronize with processors 1 and 2.

Each computation needs to wait for (synchronize) two other computations. This is complicated and slows down the program.

Problem: synchronization!

Harder problem: counting words

- How do I divide this text up to count the words in parallel on two processors?
  - Need to find the "half way" point that is not in the middle of a word

Step 1: Not Parallel (serial)
- Guess a half-way point
- Go back until we find a space

Step 2: Parallel
- Send each half to a processor

Parallel difficulties

Lorum ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
The problem with serial code

- How much does that little tiny almost insignificant non-parallel part hurt us?

**Amdahl’s law**

**Parallel performance is limited by the serial part of your code**

\[ \text{Speedup} = \frac{1}{(1 - P) + \frac{P}{S}} \]

- Example:
  - \( P = 75\% \)
  - \( S = 100,000 \)
  - \( \frac{P}{S} = \frac{75}{100,000} \)
  - \( P/\text{Speedup for parallel part} \)

**Problem: resource sharing**

- How much cache do you get?
  - Large cache on-chip
  - But shared between all cores
- Performance is depends on cache!

**Extreme multicore (manycore)**

Even a 99% parallel program only gains 93x faster at 1000 cores.

Nvidia Kepler – 1536 cores

Intel Xeon Phi – 62 cores

**Summary**

- Transistors got smaller, but threshold Voltage didn’t →
- Can’t reduce Voltage → higher power at higher frequency
  - Can’t keep increasing clock frequency
- Move to multicore
- Hard to program
  - Parallel code is hard to write (synchronization, load balancing)
  - Amdahl’s law limits the improvement (99% parallel → max 100x faster)
- Shared resources make it hard to understand
- Fundamental problem: programmability is inefficient!
Synchronization

Example: your bank account has 510 EUR in it.
- You have ordered online for 500 EUR
- You withdraw 200 EUR
- What happens when they each run on a different processor at the bank?

If they arrive one after another:
- You have ordered online for 500 EUR
- You withdraw 200 EUR

If they arrive at the same time:
- You have ordered online for 500 EUR
- You withdraw 200 EUR

What was the problem?
- Two programs (on different CPU cores) accessed the same data at (about) the same time
- While processor 1 was updating the data (but before it had written the results back)
- processor 2 read the data
- The data processor 2 read was outdated because processor 1 was in the process of updating it (but hadn’t done so yet)

Processor 1
- Data
- read data
- compute
- write 10

Processor 2
- Data
- read data
- compute
- write 310

Locks
Protecting data with locks

- A lock is a variable that protects data
  - You must acquire (get) the lock before you access the data
  - While you have the lock, no one else is allowed to access it
  - When you are done, you release (give up) the lock
  - Once you don't have it

If (lock(bal_lk)){
  load bal
  if (bal>=500)
  store bal=500
  else
  fail
  unlock(bal_lk)
}

Only one processor can have the lock at a time, so we get exclusive access.

Locking gives exclusive access

- You have ordered online for 500 EUR
- You withdraw 200 EUR

You have ordered online (500 EUR)

Processor 1

Processor 2

Locking gives exclusive access

- You have ordered online for 500 EUR
- You withdraw 200 EUR

Locking protects access to bal so only one processor can change it at a time.

Processor 1

Processor 2

Time

How do we implement a lock

- A lock is a regular memory location
  - If 0, you can take the lock, if 1 someone else has it
  - Need to do a read-check-write
  - What if someone interrupts?

Atomic swap:
  - write 1 into the lock and read back results in one instruction
  - If we get back 0, then no one had the lock and we do now
  - If we get back 1, someone else had the lock and we have to try again

This has to be done in hardware!
- Need to guarantee that no other instructions come between the read and write

Synchronization summary

- Need to make sure that other processors do not read outdated values
- Use a lock to prevent this
  - Check if the data is locked and if so, wait
  - Lock the data when we start
  - Do the update
  - Write the data back
  - Unlock the data when done

Used correctly, this prevents “data races” where we data gets corrupted because different processors have different versions of the data

Using locks correctly is hard.
**What happens with caches?**

- **You have ordered online for 500 EUR**
- **You withdraw 200 EUR**

<table>
<thead>
<tr>
<th>Time</th>
<th>Processor 1</th>
<th>Cache 1</th>
<th>DRAM</th>
<th>Processor 2</th>
<th>Cache 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
<td>sleep</td>
</tr>
<tr>
<td>5</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
<tr>
<td>10</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
<tr>
<td>15</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
<tr>
<td>20</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
<tr>
<td>25</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
<tr>
<td>30</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
<tr>
<td>35</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
<tr>
<td>40</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
<tr>
<td>45</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
<tr>
<td>50</td>
<td>read</td>
<td>load</td>
<td></td>
<td>write</td>
<td>store</td>
</tr>
</tbody>
</table>

Three different values for bal in the system!

Q: What is the problem here?

A: The caches don’t communicate to see what all other caches are doing.

Q: How did snooping help?

A: The caches don’t communicate to know when they have to update DRAM. There is no way for Cache 2 to know that Cache 1 has updated the value and not written back to DRAM.

**Cache coherency problem**

- Multiple processors each have their own (private) cache
  - When they bring data into the cache they get the latest value from DRAM
  - How do they know other processors don’t have newer versions in their caches?
  - Coherency: need to keep track of who has the latest data so we always get the most recent version

- One solution: snooping
  - Each cache snoops on what the others are doing (connect address wires)
  - If it sees another processor write to a line it has, then it invalidates that line
  - If the line is dirty it needs to write it back so the other cache loads it

**Snooping**

- All caches listen to see what all other caches are doing
  - If a cache writes new data we invalidate that line locally
  - If a cache reads data that we have and is dirty
    - We tell it to wait while we write the data back to DRAM
  - We do this by tracking cache line state:
    - Modified (I have written to it but not updated the DRAM)
    - Shared (same as in DRAM, multiple caches have a copy)
    - Invalid (have to fetch from DRAM)

- Need to update the state correctly: example
  - Cache 1 and 2 have line X in Shared state (both have the same data as in DRAM)
    - Cache 2 writes to line X
      - Cache 2 changes line X to Modified state (it has changed X)
      - Cache 1 changes line X to Invalid state (the data is out of date)
    - Cache 1 wants reads line X
      - Cache 2 writes line X to DRAM and changes it to Shared (same as in DRAM)
      - Then Cache 1 reads the line as Shared (same as in DRAM)

**Locking is hard**

  - Equipment control and operator control were not synchronized correctly
  - Ended up having outdated information for the equipment control
  - Improper use of locks when communicating between user and machine
  - 6 accidents that gave patients 100x radiation
  - Improper use of locks when handling simultaneous failures
  - Multiple failures at the same time corrupted the system
  - Blocked alarm indications
  - $55$ people without electricity

**Parallelism in the memory hierarchy: coherency**
Coherency summary

- Multiple caches mean we can have multiple copies of data
- Coherency keeps them synchronized
- We implement coherency by keeping extra state bits (Shared, Modified, Invalid) for each cache line (just like Valid and Dirty)
- All caches need to keep track of what all others are doing to make sure everyone has the latest version of the data
- This is tricky!

Instruction-level Parallelism

- Clever insight: 1/3 of all instructions are ld/st. Can we do them in parallel?
- What do we need to do ld/st in parallel with other operations?
  - Add a second ALU
  - More register file ports
  - Fetch two instructions per cycle
- The compiler puts instructions together in pairs, and we execute them
  - Dual-issue pipeline

Example: static dual-issue

```
1: add r1, r2, r3  1: add r1, r2, r3
2: sub r7, r3, r0  2: sub r7, r3, r0
3: or r5, r8, r9   3: or r5, r8, r9
4: add r1, r2, r3  4: add r1, r2, r3
5: st r8, r9      5: st r8, r9
```

In the best case we now have a IPC of 2.0 (or CPI of 0.5) with a dual-issue design we can be up to 2x faster.

Dual-issue pipeline

- Regular Path
- Ld/St Path
- Added:
  - More ALUs
  - Multiple register file ports
  - More hw for forwarding logic and ports
- Now we can issue both a ld/st and any other instruction at the same time!

Parallelism in the pipeline

```
Cycle 1: add r1, r2, r3  Cycle 2: add r1, r2, r3
Cycle 3: sub r7, r3, r0  Cycle 4: sub r7, r3, r0
Cycle 5: or r5, r8, r9   Cycle 6: or r5, r8, r9
Cycle 7: st r8, r9      Cycle 8: st r8, r9
Cycle 9: nop
```

Note: we are doing the same out-of-order!
**Superscalar: dynamic ILP**

- What if we make things WAY more complicated:
  - Have the processor look ahead at the next 100 instructions and figure out which ones are independent
  - Don't even have to do them in order! (out-of-order execution)
- What do we need?
  - Somewhere to store instructions that are waiting (reservation stations)
  - Something to schedule them
  - A buffer at the end to finish them in-order (remember we still have to do what the ISA promises even if we execute out-of-order)

**Q:** What is the benefit of executing instructions out-of-order?

- More independent instructions for ILP
- More complicated work for the compiler
- Simpler pipeline
- All of the above

**A:** More independent instructions for ILP

If we can execute instructions out-of-order, we can find instructions that come after the current one that are independent. This gives us more instructions to run in parallel, which will be faster. This is incredibly complex, though.

**Intel Haswell (due in 2013)**

- Out-of-order superscalar
- This is incredibly complex
- Take the advanced computer architecture class to learn more!

**ILP summary**

- Instruction Level Parallelism (ILP)
  - Find independent instructions that we can execute in parallel
  - Need extra hardware to do them in parallel
  - Compiler or hardware needs to determine dependencies
- Complex, messy, and very important!
  - ~2-3x performance gain from superscalar and out-of-order
  - ~3-4x power cost
  - Power efficient processors are usually in-order

![ILP summary image](image-url)