CSC2/452 Computer Organization
The Processor Pipeline and Performance

Sreepathi Pai

URCS

October 21, 2019
Outline

Recap

More Pipeline Details

Software and the Pipeline

RAM
CPU and RAM

CPU

- Registers
- Fetch
- Decode
- Execute
- Memory
- Writeback

- AGU
- L/SU
- ALU
- FPU

RAM
Outline

Recap

More Pipeline Details

Software and the Pipeline

RAM
Cycle

- Most digital circuits operate on a clock
- Inputs are read/written on a clock pulse
- The time between two consecutive clock pulses is called a cycle
  - Shorter cycles imply higher frequencies
  - A cycle is roughly a step
- Time for a circuit to do its task is usually expressed in cycles
  - Number of steps to do a task
- Moore’s law allowed higher frequencies
  - Same number of steps/cycles, but 2x the speed if frequency doubled
Results are written to register file in WB stage

So the sequence `addq %rax, %rbx ; subq %rbx, %rcx` will have to stall to allow the `add` to reach the WB stage

Usually by inserting bubbles in between the two instructions
Bypassing and Forwarding

- Results are bypassed (or forwarded, in time) to the next instruction
- Now the sequence `addq %rax, %rbx ; subq %rbx, %rcx` will not stall
  - MEM will forward results to EX
Goals: Minimize Time

\[ T = \frac{W \times t}{P} \]

- Minimize \( T \)
  - in a pipelined machine, how should we write code to minimize \( T \)?
  - how can we measure the effectiveness of our methods?
Minimizing $T$

- To minimize $T$, simplistically, we need to:
  - Reduce $W \times t$
  - Increase $P$
The Profiler

- A profiler is a tool that tells you where most of the time in your program is being spent.
- Different levels of information reporting
  - Instructions
  - Lines of code
  - Functions (most common reporting)
- Tools
  - `gprof`, you must compile your code as `gcc -pg` [See textbook for an extended example]
  - Linux `perf` (with its `perf record` and `perf report` subcommand)
Using the Profiler

- The profiler tells you which part of the code takes the most time
  - Sometimes called a “hot region”
  - Often this is a loop, so called a “hot loop”

- Performance optimization usually focuses on reducing the time for this region

- Profiler results can be used to find upper bounds on speed up possible
  - If a code takes 50% of the time, maximum speedup is 2x
  - If a code takes 90% of the time, maximum speedup is 10x
  - If a code takes 99% of the time, maximum speedup is 100x
  - If a code takes 75% of the time, what is the maximum speedup?
Timing Regions of Code

```c
start = timer();
for(i = 0; i < 10000; i++) {
    // loop body
}
stop = timer();
duration = start - stop;
```

- You can also time individual pieces of code
- Bracket the code with the correct form of `timer()` function
  - There is no such function called `timer`
  - There are many different timer functions available
- Usually interest in measuring two things:
  - Physical Time (i.e. seconds)
  - Cycles (can be converted to physical time)
Counting cycles

- Most processors keep a count of cycles they’ve been executing for.
- The current value of cycle can be read through an instruction:
  - RDTSC on x86 processors (Read Time Stamp Counter).
- Get cycles at beginning of region, and again at end of region, subtract:
  - `timer()` in this case is RDTSC.
- RDTSC is useful for very tiny regions of code:
  - Using it correctly is hard because instructions can execute out-of-order.
  - Difficult to identify “beginning” and “end” of region.
  - Intel has a whitepaper on using RDTSC correctly.
Timing regions

- Get time at beginning of region, and again at end of region, subtract
  - We’re hoping the region is large enough that errors due to out-of-order execution are relatively small
- Different timer() functions to get time
- gettimeofday - returns current time of day
  - DO NOT USE THIS FUNCTION FOR TIMING CODE
- clock_gettime - POSIX timers
  - supports multiple clocks
  - for timing, use CLOCK_MONOTONIC_RAW on Linux
  - CLOCK_MONOTONIC is appropriate on other systems
Performance Counters

- Nearly every processor contains a "Performance Monitoring Unit" (PMU)
- Counts events happening inside the processor
  - Can be used to explain performance
- Linux `perf` can be used to access these counters
- The Performance API library (PAPI) also provides functions to get counters for regions of code
clock_gettime(CLOCK_MONOTONIC_RAW, &start);

v3[0] = v1[0] + v2[0];
for(i = 1; i < N; i++) {
    v3[i] = v1[i] + v2[i] + v3[i-1];
}

clock_gettime(CLOCK_MONOTONIC_RAW, &end);

▶ We’re adding two arrays and a running sum
▶ \( N \) is on the order of millions
▶ On my laptop, this takes around 0.026 seconds (i.e. 26 milliseconds)
Data Dependences: Rewritten

```c
clock_gettime(CLOCK_MONOTONIC_RAW, &start);

v3[0] = v1[0] + v2[0];
t = v3[0];
for(i = 1; i < N; i++) {
    v3[i] = v1[i] + v2[i] + t;
    t = v3[i];
}

clock_gettime(CLOCK_MONOTONIC_RAW, &end);

▶ We’re adding two arrays and a running sum
▶ \( N \) is on the order of millions
▶ On my laptop, this takes around 0.011 seconds (i.e. 11ms)
Possible Reasons

▶ Although both loops have a long dependency, one is a dependency through a register (t) and the other is through memory (v3[i−1])

▶ Here is the output of "perf stat -e cycle_activity.stalls_mem_any -r 10"

With v3[i]:

40,911,600 cycle_activity.stalls_mem_any ( +- 1.97% )

With t:

35,815,916 cycle_activity.stalls_mem_any ( +- 1.25% )

(there are lots of other events in the PMU, this is unlikely the sole explanation)
Control Dependences

(Adapted from an example on StackOverflow)

```c
    clock_gettime(CLOCK_MONOTONIC_RAW, &start);

    // repeat to amplify effects
    for(int j = 0; j < 10000; j++) {
        for(i = 0; i < N; i++) {
            if(p[i] >= 128) {
                sum = sum + p[i];
            }
        }
    }

    clock_gettime(CLOCK_MONOTONIC_RAW, &end);
```

The array \( p[i] \) contains \( N = 65536 \) random numbers between 0 to 256.
The two scenarios

We run the loop as is:

SORT: 0
sum: 62956660000
Time: 6.048150391s

We run the loop after sorting the array:

SORT: 1
sum: 62956660000
Time: 1.730157287s
The Branch

```c
if(p[i] >= 128)
    sum = sum + p[i];
```

- This branch introduces a control dependence, which adds to `sum` in only 50% of the iterations.
- The CPU uses branch prediction to deal with control dependences.
- However, this branch is extremely hard to predict (it was constructed that way!)
A simple branch predictor

- A branch predictor can be seen as a finite-state machine with two states:
  - TAKEN: the branch is predicted to be taken
  - NOT TAKEN: the branch is predicted to be not taken
- The machine’s initial state can be any of these two states.
- When a branch is mispredicted, the machine transitions to the other state.
- The current state is stored on a per-branch basis.
When sorted, the branch is easy to predict – it is false in the beginning of the array, and switches to true later:

1,317,442,105 branches # 761.098 M/sec  
384,689 branch-misses # 0.03% of all branches

When not sorted, it is very hard to predict:

1,313,673,245 branches # 217.699 M/sec  
327,534,242 branch-misses # 24.93% of all branches

(note: this includes all branches, including those from the loop, which are predicted well)
Outline

Recap

More Pipeline Details

Software and the Pipeline

RAM
DRAM Organization

- Most RAM in today’s machines is Dynamic RAM (DRAM)
  - Volatile, loses contents when powered off
  - High density, usually requires just a transistor and a capacitor to implement
  - Dynamic, contents of “memory cell” decay with time, must be refreshed periodically

- Organization
  - DIMM (Dual Inline Memory Module, what you can purchase)
  - Rank – a set of banks addressed simultaneously
  - Bank – a (usually 2D) fixed size array
  - Array – memory cells
DRAM Organization (contd.)

[Diagram showing DRAM organization with detailed labels for Rank, Rows, Columns, Bank, Chip, Memory Controller, and Channel.]
Addressing RAM

- Most programs only look at virtual addresses
  - Addresses from 0 to $2^n - 1$, where $n = 64$ on most modern systems
- These addresses are translated to physical addresses
  - Usually 48-bits on today’s systems (about 256 TiB)
  - Physical addresses go from 0 to $M$ where $M$ is the size of memory in bytes
- DDR3 RAM (Double Dual Rate RAM) usually transfers 64 bits (8 bytes) of data at a time
  - Called a row
Mapping Virtual to Physical Addresses

Problem: We must take 64-bits of (linear) virtual memory address and map it to physical memory.

- Physical memory is organized as:
  - Memory Controller – interfaces CPU to memory
  - Channel – path to transfer 64-bits at a time
  - Rank
  - Bank
  - Row
  - Column
Creating Addresses

- Ranks: 2
- Rows: $2^{13}$
- Columns: $2^{11}$
- Banks: 16

How many bits?
Example Mapping

Most mappings are proprietary and not documented

General goal is to maximize bandwidth and throughput
  - Usually by increasing parallelism
  - Spread data over chips, so data can be collected simultaneously from multiple chips
Matrix Multiply – IJK

- Multiplying two matrices:
  - A \((m \times n)\)
  - B \((n \times k)\)
  - C \((m \times k)\) [result]

- Here: \(m = n = k\)

```c
for(ii = 0; ii < m; ii++)
    for(jj = 0; jj < n; jj++)
        for(kk = 0; kk < k; kk++)
            C[ii * k + kk] += A[ii * n + jj] * B[jj * k + kk];
```
Matrix Multiply – IKJ

for(ii = 0; ii < m; ii++)
    for(kk = 0; kk < k; kk++)
        for(jj = 0; jj < n; jj++)
            C[ii * k + kk] += A[ii * n + jj] * B[jj * k + kk];
Performance of the two versions?

- on 1024x1024 matrices of ints
- which is faster?
- by how much?
Performance of the two versions

- on 1024x1024 matrices
- Time for IJK: 0.554 s ± 0.003s (95% CI)
- Time for IKJ: 6.618 s ± 0.032s (95% CI)
What caused the nearly 12X slowdown?

- Matrix Multiply has a large number of arithmetic operations
  - But the number of operations did not change
- Matrix Multiply also refers to a large number of array elements
  - Order in which they access elements changed
  - But why should this matter?
Next Class

- Caches and the Memory Hierarchy