CS33 MT2 Review Session

Prepared by UPE for Professor Eggert's CS33.

Slides available at https://goo.gl/YnEGdX
Sign in at https://goo.gl/forms/5l9ZtQOfX1E8eccn1

Floating Point

IEEE Floating Point

[s][e][f]

- 32-bit ‘Single’
  - 1 sign bit, 8 exponent bits, 23 fraction bits
- 64-bit ‘Double’
  - 1 sign bit, 11 exponent bits, 52 fraction bits

Floating Point (cont.)

Normalized Numbers

When exponent in range 1 <= e <= 254

$$result = +- 2^{(e - 127)} * 1.f \qquad \text{(in base 2)}$$

Floating Point (cont.)

Denormalized “Tiny” Numbers

When e == 0

$$result = +- 2^{(e - 127)} * 0.f \qquad \text{(in base 2)}$$

Floating Point (cont.)

- Infinity
  - `e = 255, f = 0`
- Zero
  - `s = 0 or 1, e = 0, f = 0`
- NaN
  - `e = 255, f != 0`
  - Anything a NaN touches turns into NaN (except boolean)
  - Comparing NaN to a number as boolean result = 0
    - `(NaN == 4)  // equals 0`
    - `(NaN == NaN)  // also equals 0`

Floating Point (cont.)

- Problems
  - Overflow and underflow
  - Division by zero
  - Inexact
  - Invalid operand (square root of negative)
- Tips
  - Half of all numbers between -1 and 1 in floating point
  - Write calculations to return results in that range
  - `printf("%.17g\n", value)`
    - Usually the best (lossless) printing of floating point value

Program Optimization

Measuring Performance

In a shell, use the ‘time’ command

```
$ time …
```

Ex: `$ time sort large_file.txt`

In C, use the function

```
clock_gettime(CLOCK.REALTIME, &ts)
// ts = ‘timespec’ struct containing:
//      time_t s(seconds since 1970/1/1)
//      long ns (nanoseconds since start of that second)
```

Program Optimization

Hoisting: doing more computation out of a loop

```
ListObject my_list;
for (i = 0; i < my_list.size(); ++i) {
    //do something with my_list that doesn't change its size
}
```

Compiler generally can't know `my_list.size()` will stay the same

Program Optimization

Hoisting: doing more computation out of a loop

Compared to:

```
ListObject my_list;
int size = my_list.size();
for (i = 0; i < size; ++i) {
    //do something with my_list that doesn't change its size
}
```

Faster, because no repeated calls to `my_list.size()`!

Program Optimization

Loop Unrolling: doing more in the loop to reduce overhead

```
    for (i = 0; i < 100; ++i) {
        // Do something with i
    }
vs
    for (i = 0; i < 100; i += 2) {
        // Do something with i
        // Do something with i + 1
    }
```

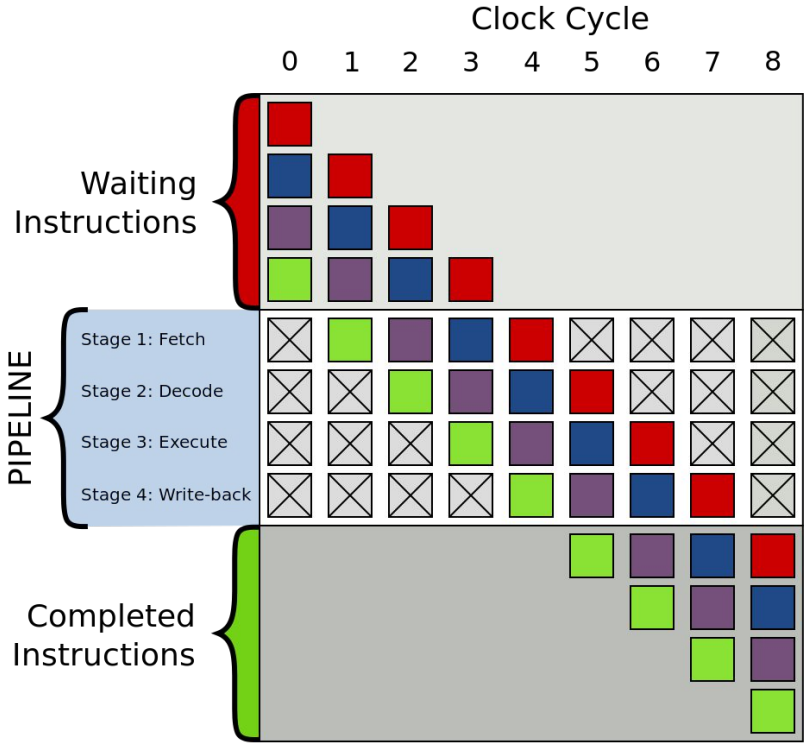
Instruction-level Parallelism

Breaking x86 instructions into lower instructions (**μops**) and executing these in parallel

Instruction-level Parallelism

Pipelining / Superscalar Execution

- Hardware translator turns a x86 instruction (like addl reg1, reg2) into the low level µops needed to perform this instruction
- µops are then “pipelined” to be done in parallel
- Problem?
  - Killed by conditional jumps
  - How can you know where you’ll jump next, i.e. which instruction will be done next?
  - Guess! Go with what usually happens, keep track, and don’t commit results if guessed incorrectly



Instruction-level Parallelism

Out of Order Execution

Instruction-level Parallelism

Out of Order Execution

In-Order Processor

1. Fetch the instruction
2. If inputs available, dispatch instruction to functional unit. If not, stall until available.
3. Instruction executed by functional unit
4. Functional unit writes results back to register file

Instruction-level Parallelism

Out of Order Execution

In-Order Processor

1. Fetch the instruction
2. If inputs available, dispatch instruction to functional unit. If not, stall until available.
3. Instruction executed by functional unit
4. Functional unit writes results back to register file

Out-of-Order Processor

1. Fetch the instruction
2. Dispatch instruction to instruction queue
3. Instruction waits in queue until inputs available. Allowed to leave queue before older instructions
4. When inputs available, instruction sent to functional unit and executed
5. Results queued
6. After older instructions have written their results, this instruction get to write its results

Instruction-level Parallelism

Out of Order Execution

- Instruction Control Unit
  - Talks to Instruction Cache
  - Fetches instructions, gives to Execution Unit
- Execution Unit
  - Is given µops to perform
  - Gives µops to hardware that can do it
    - Instruction either retired (commit its effect) or flushed (discarded)
  - Register renaming: using register other than named one to allow parallelism

Memory Hierarchy

- The general principle behind **caching** is that for each level *k*, a smaller and faster device at *k* serves as a cache for a larger slower device at *k+1*.
  - For example, DRAM is a cache to the disk, and the L1 – L3 caches are caches to DRAM.
- At the L1 cache, we often see a distinction between the L1 data and L1 instruction caches. The reasoning for this is twofold: first, as explored in ILP, we can fetch both the instruction and data in parallel. Secondly, it allows us to take advantage of … locality!

Memory Hierarchy (cont.)

- Cache blocks are generally stored in some format *like:*
  `[set][valid][dirty][tag][-- data --]`
  - The `set` and `tag` fields indicate the block of data being cached
  - We cache at a coarser granularity than single bytes
  - The distribution of bits to `set` and `tag` can heavily influence the likeliness that we're storing the right things in the cache. *(Although, this goes down a discussion about associativity that isn't too important for this class!)*

Memory Hierarchy (cont.)

- Cache blocks are generally stored in some format *like:*
  `[set][valid][dirty][tag][-- data --]`
  - The `set` and `tag` fields indicate the block of data being cached
  - We cache at a coarser granularity than single bytes
  - The distribution of bits to `set` and `tag` can heavily influence the likeliness that we're storing the right things in the cache. *(Although, this goes down a discussion about associativity that isn't too important for this class!)*

Memory Hierarchy (cont.)

- When we write programs, we want to take advantage of locality. This is usually expressed in two forms, spatial and temporal.
  - **Spatial locality:** accessing some data in some memory address probably means that you're going to be accessing data in nearby memory. Concretely, stride-1 reference patterns are good.
  - **Temporal locality:** you're probably going to be accessing data more than just once. So, repeated references to variables is great.

Thread Level Parallelism

- What's the difference between a process and a thread? *(This is a fairly common interview problem.)*
  - Threads are more lightweight
  - Threads share a memory space, while process have their own memory spaces. That said, threads maintain their own registers.
- We can use TLP to split a program, ideally an *embarrassingly parallel* program, into independent tasks.
- Be careful about race conditions!

Synchronization

- Semaphores allow us to enforce the number of threads that can enter a section of code. *(They're essentially glorified counters!)*
  - When the counter is nonzero, the door is open and the thread can pass.
  - When the counter is zero, the door is closed.
- We rely on three primitives to control the value of the counter: `sem_init`, `sem_wait`, and `sem_post`.

Synchronization (cont.)

- One way to look at the semaphore is to liken it to a doorman. The function that initializes the semaphore is the host, while the threads attempting to enter the critical section are the guests.
  - We begin with `sem_init` informing the the semaphore how many guests should be allowed;
  - The guests subsequently `sem_wait` until there is available capacity, which allows them to check in and enter the critical section; and
  - The guests check out using `sem_post`.

Synchronization (cont.)

The primitives are declared as follows:

```
int sem_init(sem_t *s, int pshared, unsigned int value);
int sem_wait(sem_t *s); // Waits until s > 0. When this happens, s
                        // is decremented and 0 is returned.
int sem_post(sem_t *s); // Increments s by one.
```

Q1

What are the purposes of having denormalized numbers?

Q1

What are the purposes of having denormalized numbers?

Represent even smaller numbers

Q1

Consider floating point with 4 fractional bits and 3 exponent bits (bias = 3)

Q1

Consider floating point with 4 fractional bits and 3 exponent bits (bias = 3)

Without denormalized numbers, smallest nonzero number:

$0\ 001\ 0000 = V1 = 2^{1-3} = 2^{-2}$

next smallest number:

$0\ 001\ 0001 = V2 = 2^{-2} + 2^{-6}$

Q1

Consider floating point with 4 fractional bits and 3 exponent bits (bias = 3)

Without denormalized numbers, smallest nonzero number:

`0 001 0000` = `V1` = $2^{1-3} = 2^{-2}$

next smallest number:

`0 001 0001` = `V2` = $2^{-2} + 2^{-6}$

Clearly `V1 != V2`, but if we did `V2 - V1`, we get 2^{-6} which is too small to represent and would get rounded down, meaning `V2 - V1 = 0, but V2 != V1.`

Q2

What’s the point of having a varying range between numbers?

Q2

- As numbers increase, significance of smaller values decrease
  - Value of 1 relative to 10 is quite significant
  - Value of 1 relative to 10^9 is insignificant

Q2

- As numbers increase, significance of smaller values decrease
  - Value of 1 relative to 10 is quite significant
  - Value of 1 relative to 10^9 is insignificant

- Floating point allows representation of very large numbers by losing precision
  - Accomplished by each bit contributing a value that increases along with the exponent of the number

Q3

Under what circumstances will a compiler perform optimization?

Q3

Under what circumstances will a compiler perform optimization?

Only when the compiler is *sure* the behavior would be the same as the un-optimized case.

Q3

Under what circumstances will a compiler perform optimization?

Only when the compiler is *sure* the behavior would be the same as the un-optimized case.

Is this

```
*xp += *yp;
*xp += *yp;
```

the same as this?

```
*xp += 2 * (*yp);
```

Q3

Under what circumstances will a compiler perform optimization?

Only when the compiler is *sure* the behavior would be the same as the un-optimized case.

Is this

```
*xp += *yp;
*xp += *yp;
```

the same as this?

```
*xp += 2 * (*yp);
```

Not if `xp == yp`

Then the result is `*xp += 3 * (*xp)`

Q4

How can you tell if an optimization is useful in the long run?

Q4

How can you tell if an optimization is useful in the long run?

Use Amdahl's Law!
a = what fraction of your program is the hog
k = hog speedup factor

$$\text{Speedup} = \frac{1}{(1-a) + \frac{a}{k}}$$

Q4

How can you tell if an optimization is useful in the long run?

Example:
Speedup half of your program (a = 0.5) to be twice as fast (k = 2)
Total speedup is only 1.33!

$$\frac{1}{(1-a)+\frac{a}{k}}$$

Q5

Suppose you want a thread T1 to wait until thread T2 finishes, and that T2 is detached. Explain how to arrange for this reliably, assuming the threads cooperate by executing code that you specify. Your code may invoke any of the `pthread_*` or `sem_*` primitives discussed in the book or in class. Explain any assumptions you make and any race conditions that you couldn't fix.

Q5

Suppose you want a thread T1 to wait until thread T2 finishes, and that T2 is detached. Explain how to arrange for this reliably, assuming the threads cooperate by executing code that you specify. Your code may invoke any of the `pthread_*` or `sem_*` primitives discussed in the book or in class. Explain any assumptions you make and any race conditions that you couldn't fix.

Can't we just use `pthread_join`?

No, because T2 is detached! Instead, we need rely on semaphores (or more specifically, mutexes) to solve this problem. More specifically, we need to implement `pthread_join` using semaphores.

Q5

Here's an idea: what if we enforce an order on the execution of critical sections?

```
void t1 {
  sem_wait(&mutex);
  // Do work here ...
  sem_post(&mutex);
}
```

Q5

Here's an idea: what if we enforce an order on the execution of critical sections?

```
void t2 {
  // Do work here ...
  sem_post(&mutex);
}
```

Q5

```
sem_t mutex;
sem_init(&mutex,
    0 /* pshared */,
    0 /* value */);

void t1 {
  sem_wait(&mutex);
  // Do work here ...
  sem_post(&mutex);
}
```

```
void t2 {
  // Do work here ...
  sem_post(&mutex);
}
```

In this solution, we take initialize `mutex` to zero such that `t1`'s critical section is only "unlocked" after `t2` completes its work. The race condition that we potentially encounter is that `t1` can begin doing its work before `t2` entirely terminates.

Q6

Discuss the tradeoffs between *write-through* and *write-back* write hit policies.

Q6

- Under the **write-through** policy, for each write hit, write to the cache and also to the backing memory.
  - Memory and cached are always synchronized and consistent
  - However, we incur the penalty of writing to memory each time we write to cache, which negates the benefit of the cache for writes.

Q6

- Under the **write-back** policy, for each write hit, write only to the cache but mark the cache block as dirty. We write back to memory upon eviction.
  - We benefit immensely from temporally localized programs!
  - Although we're guaranteed that we perform less or equally many writes as with the write-through policy, we run into the issue of unpredictability in our writes. For example, if we're developing a program for a nuclear plant, we might want to be sure of how long a write will take.
  - In the case that we have a program that performs many random reads, we may perform just as many writes but occupy extra space for the dirty bit.