Sequential Program Optimization

ICS432

Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Program Optimization

- You have a program that you need to make faster
  - i.e., as close to the computer's peak performance as possible
- You can pick better algorithms / data structures
- This is expected of a CS graduate based on what was learned in courses like 211 / 311
  - e.g., Don't do a linear search in a sorted array
  - e.g., Use a heap instead of a list when it make sense
- And then you get into the "dark art" :)

Optimizing and Implementation

- Do not change the spirit of the algorithm or the data structures
  - Because you’re using good ones
- But instead modify the details of how the code is written
  - Shuffle lines of code around
  - Tweak data structure implementations
  - Use bitwise operations
  - Make sure you don’t place too many system calls (e.g., memory allocation)

By-hand Optimization

- Your profiler told you that most of the time is spent in some part of the code
- You focus on this part of the code, and start tweaking it
  - In ICS312 I go through a small piece of code that we try to hand-optimize in class
- Let’s look at well-known code-optimization techniques and see why they would accelerate code
  - And let’s see which ones a compiler is able to do…

By-hand Code Optimization

- Move code outside of loop when possible

```
for (i=0; i < n; i++) {
    x += i * (n * 3);
}
```

```
int tmp = n*3;
for (i=0; i < n; i++) {
    x += i * tmp;
}
```

```
for (i=0; i < f(n); i++) {
    x += i;
}
```

```
int tmp = f(n);
for (i=0; i < tmp; i++) {
    x += i;
}
```

Only valid if f() has no side-effects

By-hand Code Optimization

- Move code outside of loop when possible

```
for (i=0; i < n; i++) {
    x += i * (n * 3);
}
```

Compilers can do this

```
int tmp = n*3;
for (i=0; i < n; i++) {
    x += i * tmp;
}
```

Compilers might not do this! (unless you enable costly inter-procedural analysis)

```
for (i=0; i < f(n); i++) {
    x += i;
}
```

```
int tmp = f(n);
for (i=0; i < tmp; i++) {
    x += i;
}
```

Only valid if f() has no side-effects

By-hand Code Optimization

- Avoid using arrays

```
for (i=0; i < n; i++) {
   A[i] = 1;
}
```

```
int *A_ptr = &(A[0]);
for (i=0; i < n; i++) {
   *A_ptr = 1;
   A_ptr++;
}
```

- When you write A[i] in high level code, this is really an address computation: &(A[0]) + i * sizeof(element)
- So it's one addition and one multiplication (or a shift)
- Maintaining a pointer as in the code to the right is only one addition

By-hand Code Optimization

- Avoid using arrays

Compilers can do this

```
for (i=0; i < n; i++) {
    A[i] = 1;
}
```

```
t *A_ptr = &(A[0]);
for (i=0; i < n; i++) {
    *A_ptr = 1;
    A_ptr++;
}
```

- When you write A[i] in high level code, this is really an address computation: &(A[0]) + i * sizeof(element)
- So it’s one addition and one multiplication (or a shift)
- Maintaining a pointer as in the code to the right is only one addition

By-hand Code Optimization

- Loop Unrolling

```
for (i=0; i < 21; i++) {
    A[i] = 1;
}
```

```
for (i=0; i < 20; i+=2) {
    A[i] = 1; A[i+1] = 1;
}
A[20] = 1;
```

- Above we unroll by a “factor” 2
- But we have 21 iterations
- So there is “left over” work to do after the loop

By-hand Code Optimization

- Loop Unrolling

```
for (i=0; i < 21; i++) {
    A[i] = 1;
}
```

```
for (i=0; i < 20; i+=2) {
    A[i] = 1; A[i+1] = 1;
}
A[20] = 1;
```

- The code on the right does half the number of comparisons to the loop bound!
- Unrolling the full loop would in principle be faster! (no comparisons!)
- But then there are instruction cache issues
  - There would be cache misses when fetching instructions, which may negate the benefit of loop unrolling

By-hand Code Optimization

- Loop Unrolling

Compilers can do this

```
for (i=0; i < 21; i++) {
    A[i] = 1;
}
```

```
for (i=0; i < 20; i+=2) {
    A[i] = 1; A[i+1] = 1;
}
A[20] = 1;
```

- The code on the right does half the number of comparisons to the loop bound!
- Unrolling the full loop would in principle be faster! (no comparisons!)
- But then there are instruction cache issues
  - There would be cache misses when fetching instructions, which may negate the benefit of loop unrolling

By-hand Code Optimization

- Function inlining

```
int f(int x) {
return x + 2;
}
. . .
for (i=0; i < 20; i++) {
   A[i] = f(i);
}
```

```
for (i=0; i < 20; i+=2) {
   A[i] = i + 2
}
```

- The code on the right does not have any function call
  - See ICS312 to understand how expensive function calls are

By-hand Code Optimization

- Function inlining

```
int f(int x) {
return x + 2;
}
. . .
for (i=0; i < 20; i++) {
   A[i] = f(i);
}
```

Compilers can do this

```
for (i=0; i < 20; i+=2) {
   A[i] = i + 2
}
```

- The code on the right does not have any function call
  - See ICS312 to understand how expensive function calls are

Optimization Technique Galore

- There are dozens of known optimization techniques
- The ones we saw are relatively simple
- Some are even simpler
  - e.g., strength reduction
    - e.g., don’t do “i * 2” but do “ i << 1”
    - e.g., don’t do “x = a / 4.0” but do “x = a * 0.25”
- Some are really complicated, for instance, instruction scheduling…
  - Something all compilers do at the assembly level, but that used to be done in high-level code

Instruction Scheduling

- Modern computers have multiple functional units that could be used in parallel
  - But only if instructions are in a good order
- Instruction scheduling:
  - Think of your program as a set of n assembly instructions
  - Consider all possible permutations of the instructions: fact(n) permutations
  - Among these permutations some number lead to a correct program outcome
  - Among these correct permutations one is fast because it uses all functional units to the max
  - Instruction scheduling is the problem of finding which permutation that is!

Conclusion

- A lot can be done to make code faster
- Compilers do sophisticated optimizations (decades of research and development)
- The days of transforming your code into an unreadable mess to make it fast are over!
  - And have been for a while
- But there are few things that compilers can’t / won’t do (yet), or at least not in all cases and for any code
- A difficult such thing we look at in the next set of lecture notes is data locality…