

Program Optimization

ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Performance

- So far in this course we've mostly talked about concurrency and **correctness**
 - avoid race conditions
 - avoid deadlocks
 - avoid starvation
- We have talked about parallel performance
- But although parallelization is key for performance on multi-core architecture, if the sequential code is inherently slow, the parallel code will also be slow
 - Accelerating a slow code is good, accelerating a fast code is better

How to Improve Performance?

- Option #1: Buy Faster Hardware
 - Only gets you so far for so long
 - Sometimes the amount of hardware to buy would be staggering and one can't just wait for technology improvements and price drops
 - Better to achieve the same effect by modifying the code a little bit

How to Improve Performance?

- Option #2: Modify the algorithm
 - Example: Search for an element in a sorted array
 - First implementation: a linear search
 - Easy to write at first
 - Does the job
 - When performance becomes an issue, replace the linear search by a binary search
 - More complex code
 - Goes much faster for large arrays

How to Improve Performance?

- Option #3: Modify the data structures
 - Example: Linked List
 - The `list.length()` method computes the length by going through the list and incrementing a counter
 - If users call the method often and/or the list is long, this can cause significant overhead
 - Instead, add a *length* attribute to the list class, and do +1 and -1 on it when insertion and removal
 - The new `list.length()` method just return the length attribute
 - This will vastly speeds up `list.length()`, and will minimally slow down `list.insert()` and `list.remove()` an minimally increase memory consumption by 4 bytes
 - Example: Replace a List by a Heap

How to Improve Performance?

- Option #4: Modify the implementation
 - Do not change the spirit of the algorithm but...
 - Shuffle lines of code around
 - to do instructions in a different order
 - to remove optimization blockers
 - Modify code organization
 - e.g., remove classes
 - e.g., modify data structures
 - etc.

How to Improve Performance

- Option #5: Use concurrency
 - That's the option we're mostly focused on in this course
 - But not in this lecture

Code Performance

- Unfortunately, performance conflicts with other concerns
 - Correctness
 - When trying to make code go fast one often breaks it
 - Readability
 - Fast code typically requires more lines!
 - Modularity can hurt performance
 - e.g., Too many classes and virtual methods
 - Portability
 - Code that is fast on machine A can be slow on machine B
 - At the extreme, highly optimized code is not portable at all, and in fact is done in hardware!

Ok, so now what?

- We understand how we can have reasonable definitions of performance
- We know how to measure performance on dedicated systems

- Let's look at a few standard techniques to accelerate code

Optimization Techniques

- **Technique #1:** identify loop constants

```
for (k=0;k<N;k++) {  
    c[i][j] += a[i][k] * b[k][j];  
}
```



```
sum = 0;  
for (k=0;k<N;k++) {  
    sum += a[i][k] * b[k][j];  
}  
c[i][j] = sum;
```

Optimization Techniques

- **Technique #2:** replace array accesses by pointer dereferences

```
for (j=0;j<N;j++)  
    a[i][j] = 2; // 2*N adds, N multiplies
```



```
double *ptr = &(a[i][0]); // 2 adds, 1 multiplies  
for (j=0;j<N;j++) {  
    *ptr = 2;  
    ptr++; // N integer addition  
}
```

Optimization Techniques

- **Technique #3:** Loop Unrolling

```
for (i=0;i<100;i++)  
    a[i] = i;
```



```
i=0;  
do {  
    a[i] = i; i++;  
    a[i] = i; i++;  
    a[i] = i; i++;  
    a[i] = i; i++;  
} while (i<100) // fewer comparisons
```

Optimization Techniques

■ Technique #4: Code Motion

```
sum = 0;
for (i = 0; i <= fact(n); i++)
    sum += i;
```



```
sum = 0;
f = fact(n);
for (i = 0; i <= f; i++)
    sum += i;
```

Optimization Techniques

■ Technique #5: Inlining

```
for (i=0;i<N;i++) sum += cube(i);
...
void cube(i) { return (i*i*i); }
```



```
for (i=0;i<N;i++) sum += i*i*i;
```

Other Techniques

■ Common sub-expression elimination

```
x = a + b - c;
y = a + d + e + b;
```



```
tmp = a + b;
x = tmp - c;
y = tmp + d + e;
```

Other Techniques

■ Dead code elimination

```
x = 12;
...
x = a+c;
```

Seems obvious, but may be "hidden"

```
int x = 0;
...
#ifdef FOO
    x = f(3);
#else
```



```
...
x = a+c;
```

Other Techniques

■ Strength reduction

```
a = i*3;           a = i+i+i;
```

■ Constant propagation

```
int speedup = 3;
efficiency = 100* speedup / numprocs;
x = efficiency * 2;
```

```
x = 600 / numprocs;
```

Now what?

- We have seen a few optimization techniques
- There are many other!
- We could apply them all to the code but this would result in completely unreadable/undebuggable code
- Fortunately, the compiler should come to the rescue
 - To some extent, at least
- Some compiler can do a lot for you, some not so much
- Typically compilers provided by a vendor can do pretty tricky optimizations

What do compilers do?

- All modern compilers perform some automatic optimization when generating code
 - In fact, you implement some of those in a graduate-level compiler class, and sometimes at the undergraduate level.
- Most compilers provide several levels of optimization
 - -O0: No optimization
 - in fact some is always done
 - -O1, -O2, -OX
- The higher the optimization level the higher the probability that a debugger may have trouble dealing with the code.
 - Always debug with -O0
 - some compiler enforce that -g means -O0
- Some compiler will flat out tell you that higher levels of optimization may break some code!

Compiler optimizations

- gcc is a pretty good, free compiler
 - -Os: Optimize for size
 - Some optimizations increase code size tremendously
- Do a "man gcc" and look at the many optimization options
 - one can pick and choose,
 - or just use standard sets via O1, O2, etc.
- The most fancy compilers are typically the ones done by vendors
 - You can't sell a good machine if it has a bad compiler
 - Compiler technology used to be really poor
 - also, languages used to be designed without thinking of compilers (FORTRAN, Ada)
 - no longer true: every language designer has in-depth understanding of compiler technology today

What can compilers do?

- Most of the techniques we've seen!
 - Inlining
 - Assignment of variables to registers
 - It's a difficult problem
 - Dead code elimination
 - Algebraic simplification
 - Moving invariant code out of loops
 - Constant propagation
 - Control flow simplification
 - Instruction scheduling, reordering
 - Strength reduction
 - e.g., add to pointers, rather than doing array index computation
 - Loop unrolling and software pipelining
 - Dead store elimination
 - and many other.....

What can compilers do?

- Most of the techniques we've seen!
 - Inlining
 - Assignment of variables to registers
 - It's a difficult problem
 - Dead code elimination
 - Algebraic simplification
 - Moving invariant code out of loops
 - Constant propagation
 - Control flow simplification
 - **Instruction scheduling, reordering**
 - Strength reduction
 - e.g., add to pointers, rather than doing array index computation
 - Loop unrolling and software pipelining
 - Dead store elimination
 - and many other.....


Instruction scheduling

- Modern computers have multiple functional units that could be used in parallel
- Or at least ones that are pipelined
 - if fed operands at each cycle they can produce a result at each cycle
 - although a computation may require 20 cycles
- Instruction scheduling:
 - Reorder the instructions of a program
 - e.g., at the assembly code level
 - Preserve correctness
 - Make it possible to use functional units optimally

Instruction Scheduling

- One cannot just shuffle all instructions around
- Preserving correctness means that data dependences are unchanged
- Three types of data dependences:
 - True dependence
 - a = ...
 - ... = a
 - Output dependence
 - a = ...
 - a = ...
 - Anti dependence
 - ... = a
 - a = ...

Instruction Scheduling Example

- ```
...
ADD R1, R2, R4
ADD R2, R2, 1
ADD R3, R6, R2
LOAD R4, @2
...
```
- 
- ```
...
ADD R1, R2, R4
LOAD R4, @2
ADD R2, R2, 1
ADD R3, R6, R2
...
```
- Since loading from memory can take many cycles, one may as well do it as early as possible
 - Can't move instruction earlier because of anti-dependence on R4

Limits to Compiler Optimization

- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
 - e.g., data ranges may be more limited than variable types suggest
 - e.g., using an "int" in C for what could be an enumerated type
- Most analysis is performed only within procedures
 - whole-program analysis is too expensive in most cases
- Most analysis is based only on *static* information
 - compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative
 - cannot perform optimization if it changes program behavior under *any* realizable circumstance
 - even if circumstances seem quite bizarre and unlikely

So where are we now?

- We have seen techniques to optimize code
 - reducing the number of instructions
 - instruction scheduling
 - memory access management
- But compilers do a lot of things
- So, does it mean that we, as software developers have nothing to worry about?
 - Sadly, no

Good practice

- Writing code for high performance means working hand-in-hand with the compiler
- Principle #1: Optimize things that we know the compiler **cannot** deal with
 - We'll see a few such examples in the next set of slides
- Principle #2: Write code so that the compiler **can** do its optimizations
 - Remove **optimization blockers**

Optimization blocker: aliasing

- **Aliasing**: two pointers point to the same location
- If a compiler can't tell what a pointer points at, it must assume it can point at almost anything
- Example:

```
void foo(int *q, int *p) {
    *q = 3;
    *p++;
    *q *= 4;}

```

cannot be safely optimized to:

```

    *p++;
    *q = 12;

```

because perhaps $p = q$
- Some compilers have pretty fancy aliasing analysis capabilities

Blocker: False Dependencies

- A special case of aliasing

```

a[i] = b[i] + c;
a[i+1] = b[i+1] * d;

```
- The compiler cannot know that $\&(b[i+1])$ is different from $\&(a[i])$
- Therefore, it can't do efficient instruction scheduling
- Instead, one should write code as:

```

float f1 = b[i];
float f2 = b[i+1];
a[i] = f1 + c;
a[i+1] = f2 * d;

```
- Used local variable to expose independent operations
- Some compiler allow users to give them "hints"
 - e.g., declare arrays a and b unaliased via some keyword

Blocker: Function Call

```
sum = 0;
for (i = 0; i <= fact(n); i++)
    sum += i;
```

- A compiler **cannot** optimize this because
 - function `fact` may have *side-effects*
 - e.g., modifies global variables
 - Function May Not Return Same Value for Given Arguments
 - Depends on other parts of global state, which may be modified in the loop
- Why doesn't compiler look at the code for `fact`?
 - Linker may overload with different version
 - Unless declared static
 - Interprocedural optimization is not used extensively due to cost
 - Inlining can achieve the same effect for small procedures
- Again:
 - Compiler treats procedure call as a black box
 - Weakens optimizations in and around them

Other Techniques

- Use more local variables

```
while( ... ) {
    *res++ = filter[0]*signal[0]
           + filter[1]*signal[1]
           + filter[2]*signal[2];
    signal++;
}
```



Helps some compilers

```
register float f0 = filter[0];
register float f1 = filter[1];
register float f2 = filter[2];
while( ... ) {
    *res++ = f0*signal[0]
           + f1*signal[1]
           + f2*signal[2];
    signal++;
}
```

Other Techniques

- Replace pointer updates for strided memory addressing with constant array offsets

```
f0 = *x8; x8 += 4;
f1 = *x8; x8 += 4;
f2 = *x8; x8 += 4;
```



```
f0 = x8[0];
f1 = x8[4];
f2 = x8[8];
x8 += 12;
```

Some compilers are better at figuring this out than others

Some systems may go faster with option #1, some others with option #2!

Bottom line

- Know your compilers
 - Some are great
 - Some are not so great
 - Some will not do things that you think they should do
 - often because you forget about things like aliasing
- There is not golden rule because there are some system-dependent behaviors
 - Although the general principles typically holds
- Doing all optimization by hand is a bad idea in general
 - But we're doing it in the class for some of the programming assignment to truly understand about code, hardware, and performance.

By-hand Optimization of Matrix Multiplication

```
for(i = 0; i < SIZE; i++) {
    for(j = 0; j < SIZE; j++) {
        for(k = 0; k < SIZE; k++) {
            c[i][j] += a[i][k]*b[k][j];
        }
    }
}
```

- Turned array accesses into pointer dereferences
- Assign to each element of `c` just once

```
for(i = 0; i < SIZE; i++) {
    int *orig_pa = &a[i][0];
    for(j = 0; j < SIZE; j++) {
        int *pa = orig_pa;
        int *pb = &a[0][j];
        int sum = 0;
        for(k = 0; k < SIZE; k++) {
            sum += *pa * *pb;
            pa++;
            pb += SIZE;
        }
        c[i][j] = sum;
    }
}
```

Results (Courtesy of CMU)

| RS/6000 | Simple | Optimized |
|---------|--------|-----------|
| xLC-O3 | 63.9s | 65.3s |

| R10000 | Simple | Optimized |
|---------|--------|-----------|
| cc-O0 | 34.7s | 27.4s |
| cc-O3 | 5.3s | 8.0s |
| egcc-O9 | 10.1s | 8.3s |

| Pentium II | Simple | Optimized |
|------------|--------|-----------|
| egcc-O9 | 28.4s | 25.3s |

| 21164 | Simple | Optimized |
|---------|--------|-----------|
| cc-O0 | 40.5s | 12.2s |
| cc-O5 | 16.7s | 18.6s |
| egcc-O0 | 27.2s | 19.5s |
| egcc-O9 | 12.3s | 14.7s |

Why is Simple Sometimes Better?

- Easier for humans *and* the compiler to understand
 - The more the compiler knows the more it can do
- Pointers are hard to analyze, arrays are easier
- You *never* know how fast code will run until you time it on a dedicated system
- The transformations done by hand good optimizers will often do for us
 - *And they will often do a better job than we can do, but not always*
- Pointers may cause aliases and data dependences where the array code had none

Bottom Line

How should I write my programs, given that I have a good, optimizing compiler?

- Don't: Smash Code into Oblivion
 - Hard to read, maintain & ensure correctness
- Do:
 - Select best algorithm
 - Write code that's readable & maintainable
 - Procedures, recursion, without built-in constant limits
 - Even though these factors can slow down code
 - Eliminate optimization blockers
 - Allows compiler to do its job

Conclusions

- Programming for performance means working with the compiler
 - know its limitations
 - know its capabilities if it is unhindered
- This remains a "guessing" game to extract the best possible performance
- Remember that performance is not portable
 - At least the last inch performance is very hardware-dependent