Programming for Performance  
CS 740  
Oct. 3, 2001

Topics  
- How architecture impacts your programs  
- How (and how not) to tune your code

Performance Matters  

Constant factors count!  
- easily see 10:1 performance range depending on how code is written  
- must optimize at multiple levels:  
  - algorithm, data representations, procedures, and loops

Must understand system to optimize performance  
- how programs are compiled and executed  
- how to measure program performance and identify bottlenecks  
- how to improve performance without destroying code modularity and generality

Optimizing Compilers  

Provide efficient mapping of program to machine  
- register allocation  
- code selection and ordering  
- eliminating minor inefficiencies

Don’t (usually) improve asymptotic efficiency  
- up to programmer to select best overall algorithm  
- big-O savings are (often) more important than constant factors  
  - but constant factors also matter

Have difficulty overcoming “optimization blockers”  
- potential memory aliasing  
- potential procedure side-effects

Limitations of Optimizing Compilers  

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
What do compilers try to do?

- Reduce the number of instructions
  - Dynamic
  - Static
- Take advantage of parallelism
- Eliminate useless work
- Optimize memory access patterns
- Use special hardware when available

Matrix Multiply – Simple Version

```c
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];
        }
    }
}
```

Heavy use of memory operations, addition and multiplication
Contains redundant operations

Matrix Multiply – Hand Optimized

```c
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;
    }
}
```

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

Results

Is the “optimized” code optimal?

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc -O0</td>
<td>34.7s</td>
<td>27.4s</td>
</tr>
<tr>
<td>cc -O3</td>
<td>5.3s</td>
<td>8.0s</td>
</tr>
<tr>
<td>egcc -O9</td>
<td>10.1s</td>
<td>8.3s</td>
</tr>
<tr>
<td>cc -O0</td>
<td>40.5s</td>
<td>12.2s</td>
</tr>
<tr>
<td>cc -O5</td>
<td>16.7s</td>
<td>18.6s</td>
</tr>
<tr>
<td>egcc -O9</td>
<td>12.3s</td>
<td>14.7s</td>
</tr>
<tr>
<td>Pentium II</td>
<td>27.2s</td>
<td>19.5s</td>
</tr>
<tr>
<td>egcc -O9</td>
<td>28.4s</td>
<td>25.3s</td>
</tr>
<tr>
<td>RS/6000</td>
<td>63.9s</td>
<td>65.3s</td>
</tr>
</tbody>
</table>
Why is Simple 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
The transformations we did by hand good optimizers will do for us
- And they will often do a better job than we can do
Pointers may cause aliases and data dependences where the array code had none

Optimization blocker: pointers

Aliasing: if a compiler can’t tell what a pointer points at, it must be conservative and assume it can point at almost anything
Eg:
```c
void strcpy(char *dst, char *src)
{
    while(*(src++) != '\0')
        *(dst++) = *src;
    *dst = '\0';
}
```

Could optimize to a much better loop if only we knew that our strings do not alias each other

SGI’s Superior Compiler

Loop unrolling
- Central loop is unrolled 2X
Code scheduling
- Loads are moved up in the schedule to hide their latency
Loop interchange
- Inner two loops are interchanged giving us ikj rather than ijk
  - Better cache performance - gives us a huge benefit
Software pipelining
- Do loads for next iteration while doing multiply for current iteration
Strength reduction
- Add 4 to current array location to get next one rather than multiplying by index
Loop invariant code motion
- Values which are constants are not re-computed for each loop iteration

Loop Interchange

```
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];
```

Does any loop iteration read a value produced by any other iteration?
What do the memory access patterns look like in the inner loop?
- ijk: constant += sequential * striding
- ijk: sequential += constant * sequential
- jki: constant += sequential * striding
- kji: striding += constant * sequential
- kji: striding += constant * striding
Software Pipelining

- Now must optimize inner loop
- Want to do as much work as possible in each iteration
- Keep all of the functional units busy in the processor

```
for(j = 0; j < SIZE; j++)
c_r[j] += a_r_c * b_r[j];
```

Dataflow graph:

- Load $b_r[j]$
- Load $a_r_c$
- Load $c_r[j]$
- *
- +
- Store $c_r[j]$

Software Pipelining cont.

```
for(j = 0; j < SIZE; j++)
c_r[j] += a_r_c * b_r[j];
```

Not pipelined:

```
load b_r[j] a_r_c
load c_r[j] *
+
store c_r[j]
```

Pipelined:

```
load b_r[j] a_r_c
load c_r[j] *
+
store c_r[j]
```

Code Motion Examples

- Sum Integers from 1 to $n!$

**Bad**

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

**Better**

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

**Best**

```c
fn = fact(n);
sum = fn * (fn + 1) / 2;
```

Optimization Blocker: Procedure Calls

Why couldn’t the compiler move fact(n) out of the inner loop?

- **Procedure May Have Side Effects**
  - i.e., alters global state each time called
- **Function May Not Return Same Value for Given Arguments**
  - Depends on other parts of global state

Why doesn’t compiler look at code for fact(n)?

- **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

Warning:

- **Compiler treats procedure call as a black box**
- **Weaken optimizations in and around them**
### Role of Programmer

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
  - Account for cache behavior

**Focus on Inner Loops**
- Use a profiler to find important ones!