Topics

- How architecture impacts your programs
- How (and how not) to tune your code
- Statically scheduled processors

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 realistic 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
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++) {
    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

Results

Is the “optimized” code optimal?

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>R10000</td>
<td>34.7s</td>
<td>27.4s</td>
</tr>
<tr>
<td>cc -O0</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>21164</td>
<td>40.5s</td>
<td>12.2s</td>
</tr>
<tr>
<td>cc -O0</td>
<td>16.7s</td>
<td>18.6s</td>
</tr>
<tr>
<td>egcc -O0</td>
<td>27.2s</td>
<td>19.5s</td>
</tr>
<tr>
<td>egcc -O5</td>
<td>12.3s</td>
<td>14.7s</td>
</tr>
<tr>
<td>Pentium II</td>
<td>28.4s</td>
<td>25.3s</td>
</tr>
<tr>
<td>xlc -O3</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**

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
- ikj: sequential += constant * sequential
- jik: constant += sequential * striding
- jki: striding += striding * constant
- kij: sequential += constant * sequential
- kji: striding += striding * constant
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

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

Dataflow graph:

```
load b_r[j] 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];
```

Pipeline:

```
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
load b_r[j] a_r_c
load c_r[j] *
store c_r[j]
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
sum = 0;
fn = fact(n);
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
- Weakens 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!

Other than Superscalar

- All of the above applies to all processors.
- So far, we have focused on superscalar
  - Dynamic issue
  - Out-of-order
  - Resolve dependences at runtime

- What about VLIW?
  - Requires static scheduling
  - Move work to compiler
  - More scalable

VLIW

- Issue multiple instructions at once - BUT
  - Each instruction is assigned to a pre-determined FU
  - Thus, VERY LONG instruction word

- Originally, no interlocks
- Requires deep understanding of architecture
- Why is this problematic?

Finding ILP in VLIW

- Increase number of instructions that can be scheduled.
- Loop unrolling
- Global Scheduling
  - Trace scheduling
  - Software pipelining
Example from Text

For (i=1000; i>0; i--)
\[ x[i] = x[i] + s \]

L:
LD F0 <- [R0] ; R0 points to X[i]
ADD F0 <- F0 + F1 ; F1 holds s
STR [R0] <- F0
SUB R0 <- R0 - 8
BNE R0, R1, L ; R1 holds x[-1]

Going to VLIW

L:
LD F0 <- [R0] ; R0 points to X[i]
ADD F0 <- F0 + F1 ; F1 holds s
STR [R0] <- F0
SUB R0 <- R0 - 8
BNE R0, R1, L ; R1 holds x[-1]

Assume, 2 Mem, 2 FP, 1 I, 1 B
L: LD, -, -, -, -
    -, -, Add, -, -
    STR, -, -, -, -
    ... |

Unroll

L:
LD F0 <- [R0] ; R0 points to X[i]
ADD F0 <- F0 + F1 ; F1 holds s
STR [R0] <- F0
SUB R0 <- R0 - 8
BNE R0, R1, L ; R1 holds x[-1]

Assume, 2 Mem, 2 FP, 1 I, 1 B
L: LD F0, [R0],
    LD F1, -8[R0],
    LD F2, -16[R0],
    LD F3, -32[R0],
    LD F4, -40[R0],
    LD F5, -48[R0],
    ADD, Add,
    LD, LD, Add, Add,
    STR, STR, Add, Add,
    STR, STR, Add, Add, R2 <- R0 - 54
    STR, STR, -, -, -, BNE R2, F1, L
    str, str, -, -, -, R0 <- R2

Other Ways to increase ILP

Predication
Speculation

E.g., IA-64
All instructions predicated
LD.S
LD.A

But, what about exceptions?
<table>
<thead>
<tr>
<th>Speculation &amp; Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Ignore exceptions for speculative instructions</td>
</tr>
<tr>
<td>- Checks inserted</td>
</tr>
<tr>
<td>- Poison bits</td>
</tr>
<tr>
<td>- Commit buffers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IA-64 approach to VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Instruction bundles (not one long word)</td>
</tr>
<tr>
<td>- Predication on all instructions</td>
</tr>
<tr>
<td>- Speculative loads (and Checks)</td>
</tr>
<tr>
<td>- Poison bits (or words)</td>
</tr>
<tr>
<td>- Rotating registers</td>
</tr>
</tbody>
</table>