Code Optimization: Machine Independent Optimizations

Topics
- Machine-Independent Optimizations
  - Common subexpression sharing
  - Code motion (e.g., move computation with constant result outside loop)
  - Reduction in strength (e.g., reduce procedure calls)
  - Eliminating unneeded memory references
Optimizing Compilers

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

Have difficulty overcoming “optimization blockers”
- potential memory aliasing
  - Two different memory references specify single location
- potential procedure side-effects
  - Procedure may alter global state each time called

When in doubt, the compiler must be conservative
- must not cause any change in program behavior under any possible condition
Memory Aliasing

Avoid compiler optimizations of aliased pointers

```c
void add_memory_vals (long *xp, long *yp) {
    *xp += *yp;
    *xp += *yp;
    *xp += *yp;
}

void add_memory_vals2 (long *xp, long *yp) {
    *xp += 2 * *yp;
    *xp += 2 * *yp;
}
```
Machine-Independent Opts.

Share Common Subexpressions

- Reuse portions of expressions

Compilers often not very sophisticated in exploiting arithmetic properties

```c
/* Sum neighbors of i,j */
up = val[(i-1)*n + j];
down = val[(i+1)*n + j];
left = val[i*n  + j-1];
right = val[i*n  + j+1];
sum = up + down + left + right;
```

```c
long inj = i*n + j;
up = val[inj - n];
down = val[inj + n];
left = val[inj - n];
right = val[inj + n];
sum = up + down + left + right;
```

3 multiplications: i*n, (i-1)*n, (i+1)*n
1 multiplication: i*n

```assembly
leaq 1(%rsi), %rax  # i+1
leaq -1(%rsi), %r8  # i-1
imulq %rcx, %rsi  # i*n
imulq %rcx, %rax  # (i+1)*n
imulq %rcx, %r8  # (i-1)*n
addq %rdx, %rsi  # i*n+j
addq %rdx, %rax  # (i+1)*n+j
addq %rdx, %r8  # (i-1)*n+j
```
Vector ADT

Procedures

vec_ptr new_vec(int len)
- Create vector of specified length

vec_length(vec_ptr v)
- Return length of vector

int get_vec_element(vec_ptr v, int index, int *dest)
- Retrieve vector element, store at *dest
- Return 0 if out of bounds, 1 if successful

int *get_vec_start(vec_ptr v)
- Return pointer to start of vector data
Optimization Example

```c
void combine1(vec_ptr v, int *dest) {
    int i;
    *dest = 0;
    for (i = 0; i < vec_length(v); i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```

**Procedure**

- Compute sum of all elements of integer vector
- Store result at destination location
- Vector data structure and operations defined via abstract data type
Understanding Loop

void combine1-goto(vec_ptr v, int *dest)
{
    int i = 0;
    int val;
    *dest = 0;
    goto Test;
    loop:
        get_vec_element(v, i, &val);
        *dest += val;
        i++;
    Test:
        if (i < vec_length(v))
            goto loop
    done:
}

Inefficiency

- Procedure vec_length called every iteration
- Even though result always the same
Move vec_length Call Out of Loop

```c
void combine2(vec_ptr v, int *dest) {
    int i;
    int length = vec_length(v);
    *dest = 0;
    for (i = 0; i < length; i++) {
        int val;
        get_vec_element(v, i, &val);
        *dest += val;
    }
}
```

**Optimization**

- Eliminate loop inefficiency
- Move call to vec_length out of inner loop
  - Value does not change from one iteration to next
Optimization Blocker: Procedure Calls

Why couldn’t the compiler move vec_length out of the inner loop?

- Procedure may have side effects
  - Alters global state each time called

Why doesn’t compiler look at code for vec_length?

- Interprocedural optimization is not used extensively due to cost

Warning:

- Compiler treats procedure call as a black box
- Weak optimizations in and around them
Reducing Procedure Calls

```
void combine3(vec_ptr v, int *dest) {
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    *dest = 0;
    for (i = 0; i < length; i++)
        *dest += data[i];
}
```

**Optimization**

- Avoid procedure call to retrieve each vector element
  - Get pointer to start of array before loop
  - Within loop just do pointer reference
  - Not as clean in terms of data abstraction !!!!!! (Impacts modularity)

- Why more efficient?
  - Procedure calls are expensive!
  - Bounds checking is expensive
Eliminate Unneeded Memory Refs

void combine4(vec_ptr v, int *dest)
{
    int i;
    int length = vec_length(v);
    int *data = get_vec_start(v);
    int sum = 0;
    for (i = 0; i < length; i++)
        sum += data[i];
    *dest = sum;
}

Optimization

- Don’t need to store in destination until end
- Local variable sum held in register
- Avoids 1 memory read, 1 memory write per iteration
- Why more efficient?
  - Memory references are expensive!
Detecting Unneeded Memory Refs.

**Performance**

- **Combine3**
  - 6 instructions in more time
  - `addl` must read and write memory

- **Combine4**
  - 5 instructions in less time
Optimization Blocker: Memory Aliasing

Aliasing
- Two different memory references specify single location

Example
- v: [3, 2, 17]
- combine3(v, get_vec_start(v)+2) --> ?
- combine4(v, get_vec_start(v)+2) --> ?

Answer: *dest = v[2] = 10 vs. 22
Loop Unrolling

Example

do { /* count > 0 assumed */
    *to = *from++;
} while (--count > 0);

- Can use techniques such as Duff’s Device to unroll loops
- Avoids repeated testing of conditions and (conditional) jumps
Duff’s Device

Example

```c
switch (count % 8) {
    case 0:        do {  *to = *from++;  
    case 7:              *to = *from++;  
    case 6:              *to = *from++;  
    case 5:              *to = *from++;  
    case 4:              *to = *from++;  
    case 3:              *to = *from++;  
    case 2:              *to = *from++;  
    case 1:              *to = *from++;  
    } while ((count -= 8) > 0);
}
```

- C allows for case statements within nested sub-blocks of a matching switch statement
- The absence of break statements allows fallthrough
- The above can sometimes optimize away branches due to loop-unrolling
- Duff’s device unwinds blocks of statements (here, 8) to be called in one branch