Optimizing Compilers

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

Don’t (usually) improve asymptotic efficiency
- Up to programmer to select best overall algorithm

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

Machine-Independent Optimizations (Cont.)
Share Common Subexpressions
- Reuse portions of expressions

Only 1 multiplication: i’n

Extract common expression from subexpressions: i’n, (i−1)’n, (i+1)’n

Optimizations you should do regardless of processor / compiler

Code Motion
- Reduce frequency with which computation performed
  - If it will always produce same result
  - Especially moving code out of loop
- Compilers are good at this for simple loop/array structures

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
Reduction in Strength

Strength Reduction
- Costly operation is replaced with an equivalent but less expensive operation
- E.g. 1 Shift, add instead of multiply or divide
  \[ 16 \times x \rightarrow x \ll 4 \]
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
- E.g. 2 Recognize sequence of products

Example

Procedure
- Compute sum of all elements of vector
- Store result at destination location

Move vec_length Call Out of Loop

void combine1(vec * 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;
    }
}

void combine2(vec * 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;
    }
}

- Move call to vec_length out of inner loop
  - vec_length requires only constant time but significant overhead due function call
  - Value does not change from one iteration to next
- Optimization strategy: Code motion

Vector ADT

```c
struct vec_s{
    int len;
    int *data;
};
typedef struct vec_s vec

Procedures
vec * new_vec(int len)
    - Create vector of specified length
    - int get_vec_element(vec * 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 * v)
    - Return pointer to start of vector data

Example

void combine1(vec * 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;
    }
}
```

```
void combine2(vec * 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 Blocker: Procedure Calls

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

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?
- Linker may overload with different version
  - Unless declared static
- Interprocedural optimization is not used extensively due to cost

Reduction in Strength

```c
void combine3(vec * 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++)
        *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

Eliminate Unneeded Memory Refs

```c
void combine4(vec * 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

Limitations:
- However, compiler may not be able to transform code combine3 to combine4 due to memory aliasing

Optimization Blocker: Memory Aliasing

Aliasing
- Two different memory references specify single location

Example
- v: [2, 3, 17]
- Create alias between last element of the vector and destination for storing the result
  - combine3(v, get_vec_start(v)+2) --> ?
  - combine4(v, get_vec_start(v)+2) --> ?

Observations
- Easy to have happen in C
  - Since allowed to do address arithmetic
  - Direct access to storage structures
- Get in habit of introducing local variables
  - Less expensive in accessing as local variable will be a register
- Your way of telling compiler not to check for aliasing
Machine-Independent Opt. Summary

Two common techniques:

Code Motion
- Compilers are good at this for simple loop/array structures
- Don't do well in presence of procedure calls and memory aliasing

Reduction in Strength
- Shift, add instead of multiply or divide
  - Compilers are (generally) good at this
  - Exact trade-offs machine-dependent
- Keep data in registers rather than memory
  - Compilers are not good at this, since concerned with aliasing

Limitations of Optimizing Compilers

Operate Under Fundamental Constraint
- Must not cause any change in program behavior under any possible condition

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

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