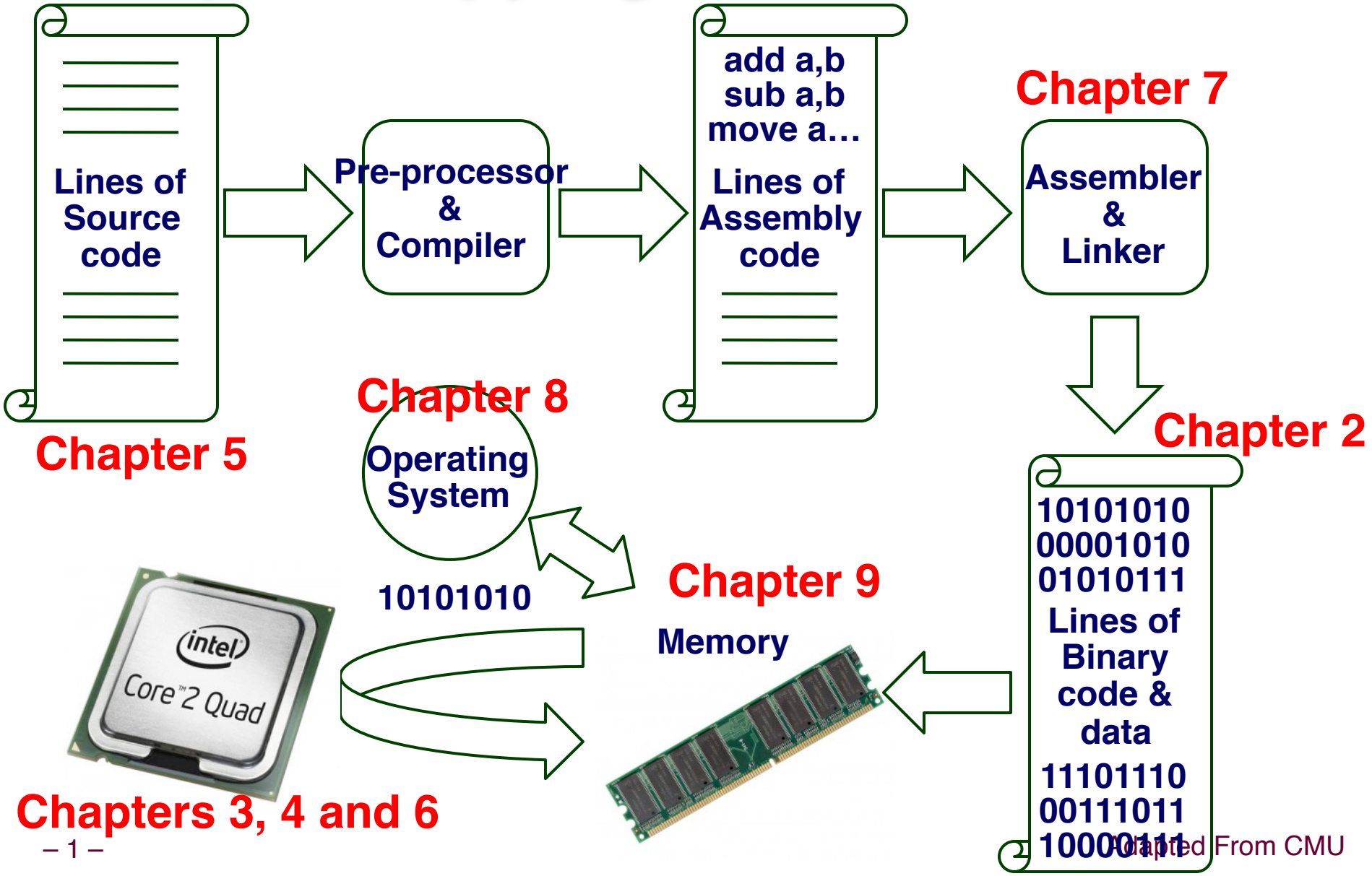
Chapter Mapping



Chapter 5: Improving the Performance of Code Execution

Topics

- Machine-Independent Optimizations
  - Code motion
  - Reduction in strength
  - Common subexpression sharing

Optimizing Compilers

Everyone wants their code to run faster

- Web site servers and Web browsers
- Scientists running massive atmospheric simulations
- Mobile clients – rendering a pinch-and-zoom UI

Can compile code with different optimization levels

- -O1, -O2, -O3
- Compiler applies different transformations to source code to try and make the code run faster
  - Code motion
  - Reduction in strength
  - Common subexpression sharing

Limitations of Optimizing Compilers

Operate Under Fundamental Constraint

- Must not cause any change in program behavior under any possible condition
- Often prevents it from making optimizations when would only affect behavior under pathological conditions.

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

- potential memory aliasing
- potential procedure side-effects

Optimization Blocker: Memory Aliasing

Memory Aliasing – two pointers point to or alias the same memory location

```
void  fun1(int *xp, int *yp)
{
  *xp += *yp;   // line 1
  *xp += *yp;   // line 2
}
```


```
void  fun1(int *xp, int *yp)
{
  *xp += 2* *yp;
}
```

If xp==yp,
then *xp = 2* *xp after line 1
and *xp = 4* *xp after line 2

If xp==yp,
then *xp = 3* *xp

- **Compiler must be conservative and not blindly apply optimizations, in this case withholding optimization in case the pointers are aliased**

Optimization Blocker: Procedure Side-Effects

Function modifies some part of the global program state

```
int f();

void  fun1(int *xp, int *yp)
{
  return f()+f()+f()+f()
}
```


```
int f();

void  fun1(int *xp, int *yp)
{
  return 4*f();
}
```

Suppose f() increments a global counter

then above fun1() will increment counter 4 times

But above "optimized" fun1() will increment counter only once!

- **Compiler must be conservative and not blindly apply optimizations, in this case because of functional side effects**
- **Most compilers don't try to determine whether a function is free from side effects**

Machine-Independent Optimizations

- Optimizations you should do regardless of processor / compiler
- Usually handled by compiler

Code Motion

- Reduce frequency with which computation performed
  - If it will always produce same result
  - Especially moving code out of loop

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

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Compiler-Generated Code Motion

- Most compilers do a good job with array code + simple loop structures

Code Generated by GCC

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

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  int *p = a+ni;
  for (j = 0; j < n; j++)
    *p++ = b[j];
}
```

```
  imull %ebx,%eax         # i*n outside of inner j loop
  movl 8(%ebp),%edi       # a
  leal (%edi,%eax,4),%edx # p = a+i*n (scaled by 4)
# Inner Loop
.L40:
  movl 12(%ebp),%edi      # b
  movl (%edi,%ecx,4),%eax # b+j  (scaled by 4)
  movl %eax,(%edx)        # *p = b[j]
  addl $4,%edx            # p++  (scaled by 4)
  incl %ecx               # j++
  jl .L40                 # loop if j<n
```

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x  -->  x << 4`

  - Utility is machine dependent
  - Depends on cost of multiply or divide instruction
  - On Pentium II or III, integer multiply only requires 4 CPU cycles
- Recognize sequence of products

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

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Make Use of Registers

- Reading and writing registers much faster than reading/ writing memory

Limitation

- Compiler not always able to determine whether variable can be held in register
- Possibility of *Aliasing*
- See example later

Machine-Independent Opts. (Cont.)

Share Common Subexpressions

- Reuse portions of expressions
- Compilers often not very sophisticated in exploiting arithmetic properties

```
/* 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;
```

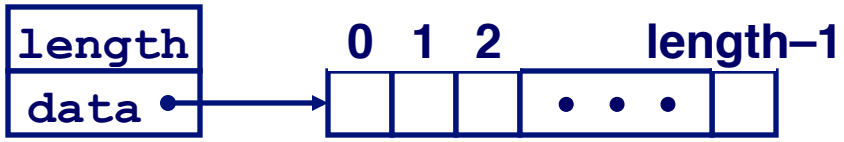
3 multiplications: i*n, (i–1)*n, (i+1)*n

```
int inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
  leal -1(%edx),%ecx  # i-1
  imull %ebx,%ecx     # (i-1)*n
  leal 1(%edx),%eax   # i+1
  imull %ebx,%eax     # (i+1)*n
  imull %ebx,%edx     # i*n
```

Vector ADT



Procedures

`vec_ptr new_vec(int len)`

- Create vector of specified length

`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 vec_length(vec_ptr v)`

- Return length of vector

`int *get_vec_start(vec_ptr v)`

- Return pointer to start of vector data

- Similar to array implementations in Pascal, Java
  - e.g., always do bounds checking

Optimization Example

Procedure

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

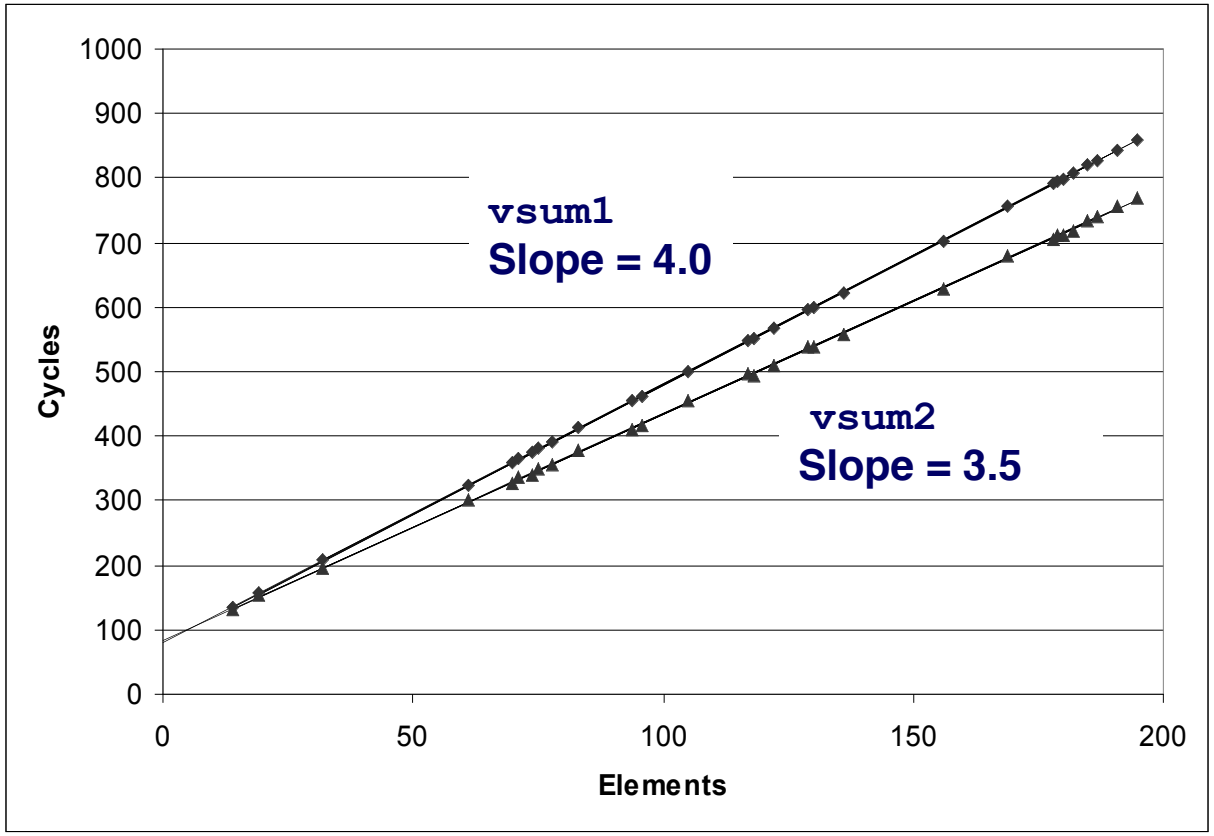
```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

Pentium II/III Performance: Clock Cycles / Element (CPE)

- 42.06 (Compiled -g) 31.25 (Compiled -O2)

Cycles Per Element (CPE)

- Convenient way to express performance of program that operators on vectors or lists
- Length = n
- T = CPE*n + Overhead



Understanding Loop

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

1 iteration

Inefficiency

- Procedure vec_length called every iteration
- Even though result always the same

Move `vec_length` Call Out of Loop

Optimization

- Move call to `vec_length` out of inner loop
  - Value does not change from one iteration to next
  - Code motion

```
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;
  }
}
```

- CPE: 20.66 (Compiled -O2)
  - `vec_length` requires only constant time, but significant overhead

Code Motion Example #2

Procedure to Convert String to Lower Case

```
void lower1(char *s)
{
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

For a string of length n, strlen() takes n steps to search for '\0', and the loop calls strlen() n times, so overall complexity is n^2 --- Quadratic!

```
void lower2(char *s)
{
  int i;
  int len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

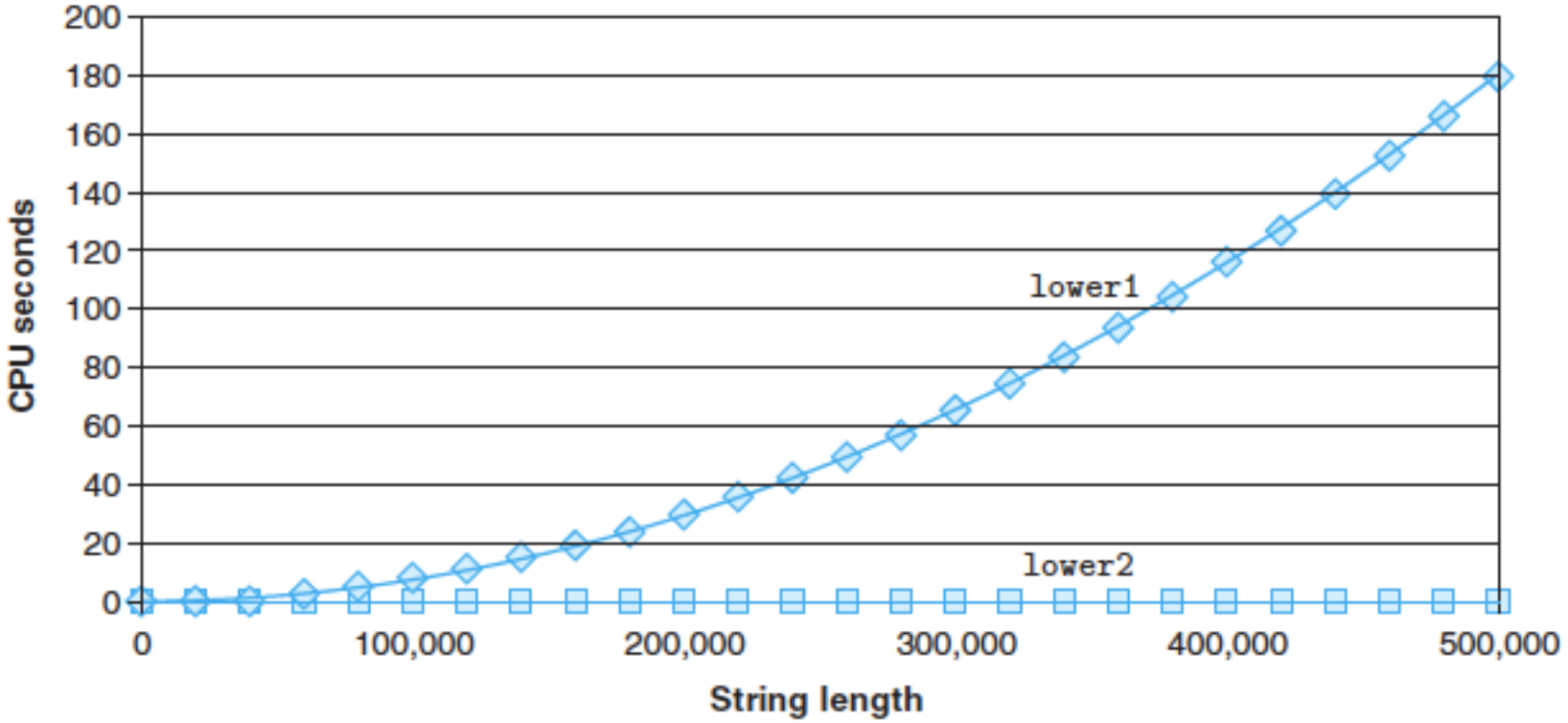
Execute strlen() only once outside of loop, since result does not change at each Iteration.

Much more scalable!

Form of code motion

Lower Case Conversion Performance

- Time quadruples when double string length
- Quadratic performance



Optimization Blocker: Procedure Calls

Why couldn't the compiler move `vec_length` or `strlen` out of the inner loop?

- Procedure may have side effects
  - Alters global state each time called
- Function may not return same value for given arguments
  - Depends on other parts of global state
  - Procedure `lower` could interact with `strlen`

Why doesn't compiler look at code for `vec_len` or `strlen`?

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

Warning:

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

Reduction in Strength

```
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
- CPE: 6.00 (Compiled -O2)
  - 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 element
- CPE: 2.00 (Compiled -O2)
  - Memory references are expensive!

Detecting Unneeded Memory Refs.

Combine3

```
.L18:
        movl  (%ecx,%edx,4),%eax
        addl  %eax,(%edi)
        incl  %edx
        cmpl  %esi,%edx
        jl  .L18
```

Combine4

```
.L24:
        addl  (%eax,%edx,4),%ecx

        incl  %edx
        cmpl  %esi,%edx
        jl  .L24
```

Performance

- Combine3
  - 5 instructions in 6 clock cycles
  - `addl` must read and write memory
- Combine4
  - 4 instructions in 2 clock cycles (we'll see an example of how this is derived later in the class)

Optimization Blocker: Memory Aliasing

```
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];
}
```

```
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;
}
```

Example

- `combine(v, dest)`
  - Normally expect `dest` to be any pointer unrelated to v, but what if aliased?
- `v: [3, 2, 17]`
- `combine3(v, get_vec_start(v)+2)` → ?
  - `[3,2,0]` before loop
  - `[3,2,3]` i=0
  - `[3,2,5]` i=1
  - `[3,2,10]` i=2, final
- `combine4(v, get_vec_start(v)+2)` → ?
  - `[3,2,17]` before loop
  - Sum updated to 22
  - `[3,2,22]` final

Optimization Blocker: Memory Aliasing

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, e.g. combine4()
  - Accumulating within loops
  - Your way of telling compiler not to check for aliasing

Machine-Independent Opt. Summary

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***

Share Common Subexpressions

- *compilers have limited algebraic reasoning capabilities*

General Forms of Combining

```
void abstract_combine4(vec_ptr v, data_t *dest)
{
  int i;
  int length = vec_length(v);
  data_t *data = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP data[i];
  *dest = t;
}
```

Data Types

- Use different declarations for `data_t`
- `int`
- `float`
- `double`

Operations

- Use different definitions of `OP` and `IDENT`
- `+` / `0`
- `*` / `1`

Machine Independent Opt. Results

Optimizations

- Reduce function calls and memory references within loop

Method	Integer		Floating Point	
	+	*	+	*
Abstract -g	42.06	41.86	41.44	160.00
Abstract -O2	31.25	33.25	31.25	143.00
Move vec_length	20.66	21.25	21.15	135.00
data access	6.00	9.00	8.00	117.00
Accum. in temp	2.00	4.00	3.00	5.00

Performance Anomaly

- Computing FP product of all elements exceptionally slow.
- Very large speedup when accumulate in temporary
- Caused by quirk of IA32 floating point
  - Memory uses 64-bit format, register use 80
  - Benchmark data caused overflow of 64 bits, but not 80

Pointer Code

```
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;
}
```

```
void combine4p(vec_ptr v, int *dest)
{
  int length = vec_length(v);
  int *data = get_vec_start(v);
  int *dend = data+length;
  int sum = 0;
  while (data < dend) {
    sum += *data;
    data++;
  }
  *dest = sum;
}
```

Optimization

- Use pointers rather than array references
- CPE: 3.00 (Compiled -O2)
  - Oops! We're not making progress here!

Warning: Some compilers do better job optimizing array code

Pointer vs. Array Code Inner Loops

Array Code

```
.L24:                           # Loop:
  addl (%eax,%edx,4),%ecx       # sum += data[i]
  incl %edx                     # i++
  cmpl %esi,%edx                # i:length
  jl .L24                       # if < goto Loop
```

Pointer Code

```
.L30:                   # Loop:
  addl (%eax),%ecx      #  sum += *data
  addl $4,%eax          #  data ++
  cmpl %edx,%eax        #  data:dend
  jb .L30               #  if < goto Loop
```

Performance

- Array Code: 4 instructions in 2 clock cycles
- Pointer Code: Almost same 4 instructions in 3 clock cycles

Supplementary Slides

Time Scales

Absolute Time

- Typically use nanoseconds
  - 10^{-9} seconds
- Time scale of computer instructions

Clock Cycles

- Most computers controlled by high frequency clock signal
- Typical Range
  - 100 MHz
    - » 10^{8} cycles per second
    - » Clock period = 10ns
  - 2 GHz
    - » 2 X 10^{9} cycles per second
    - » Clock period = 0.5ns

Lower Case Conversion Performance

- Time doubles when double string length
- Linear performance