CS 4400
Computer Systems

LECTURE 10

Capabilities and limitations of compilers

Optimization blockers

Machine-independent optimizations

Optimization

- Writing efficient programs requires
  - selecting the best data structures and algorithms
  - writing source code that the compiler can optimize
- Often there is a trade-off between readability and speed.
  - one can program a simple insertion sort in minutes
  - a highly-efficient sorting routine can take days to code, debug
- When should program performance be traded for ease of implementation and maintenance?
- Optimizations are machine independent or dependent.

Capabilities of Compilers

- By determining what values are computed and how they are used, optimizing compilers can often generate faster code than a compiler doing a direct translation.
- Optimizing compilers exploit opportunities
  - to simplify expressions
  - to use a single computation in several places
  - to reduce the number of times a given computation is performed
- But, all of this must be done in addition to maintaining the exact semantics of the original program.

Limitations of Compilers

1. The correct program behavior must be maintained.

2. Understanding of the program's behavior and the environment in which it will be used is limited.

3. Optimizations must be performed quickly.

```
void twiddle1(int* xp, int* yp) {
  *xp += *yp;
  *xp += *yp;
}

void twiddle2(int* xp, int* yp) {
  *xp += 2 * *yp;
}
```

Which function is more efficient?

CLICK: 1 or 2

Is the behavior of each identical?

CLICK: 1-yes or 2-no

Optimization Blocker: Aliasing

- An *optimization blocker* is a feature of the program's behavior that depends strongly on execution environment.
- Memory aliasing is when a single memory location can be referenced with multiple identifiers.
  - The compiler must assume that different pointers may designate the same place in memory.

```
void swap(int* xp, int* yp) {
  *xp = *xp + *yp;  /* x+y */
  *yp = *xp - *yp;  /* x+y-y=x */
  *xp = *xp - *yp;  /* x+y-x=y */
}
```

What if `xp` and `yp` are equal?

Optimization Blocker: Function Calls

```
int counter = 0;

int f(int x) { return counter += x; }

int func1(int x) {
  return f(x) + f(x) + f(x) + f(x);
}

int func2(int x) { return 4 * f(x); }
```

How are `func1` and `func2` different?

- Function `f` has a *side effect*—modifying part of the global program state.
- Most compilers do not try to determine whether a function is free of side effects.
  - They simply assume the worst case.

```c
typedef int data_t;          /* change as needed for float, ... */

typedef struct {
  int len;
  data_t* data;
} vec_rec, * vec_ptr;        /* typedefs struct, pointer to struct */

vec_ptr new_vec(int len) {   /* create vector of specified length */
  vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
  if(!result) return NULL;   /* cannot allocate storage */
  result->len = len;
  if(len > 0) {
    data_t* data = (data_t*) calloc(len, sizeof(data_t));
    if(!data) {              /* cannot allocate storage */
      free((void*) result);
      return NULL;
    }
    result->data = data;
  }
  else result->data = NULL;
  return result;
}

/* retrieve vector element and store at dest */
int get_vec_element(vec_ptr v, int index, data_t* dest) {
  if(index < 0 || index >= v->len)   /* bounds checking */
    return 0;
  *dest = v->data[index];
  return 1;
}

int vec_length(vec_ptr v) { return v->len; };
```

Example: Vector ADT

```
#define IDENT 0     /* 0,+ sums elements of vector */
#define OPER +      /* change to 1,* for product */

void combine1(vec_ptr v, data_t* dest) {
  int i;

  *dest = IDENT;
  for(i = 0; i < vec_length(v); i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OPER val;
  }
}
```

CPEs	int		float-pt		
	+	*	+	F *	D *
gcc	29.01	29.21	27.40	27.90	27.36
gcc -O1	12.00	12.00	12.00	12.01	13.00

Performance Measurements

- Among compilers, the optimization capabilities of gcc are considered adequate (not exceptional).
- *Unoptimized*—code suitable for stepping through with debugger, closely matches source code.
  - `-O1`—enables basic optimizations
- CPE measures the number of clock cycles per element.
  - appropriate for programs that perform a repetitive computation (e.g., processing pixels, computing elts of matrix product)
  - not necessarily *cycles per iteration*
  - Why is CPE better than measuring actual running time?

Loop Inefficiency

- Observe that `combine1` calls `vec_length` as the test condition on *every iteration of the loop*.
- However, the vector length does not change.
  - As we know, the compiler will not move the function call.
  - The programmer must explicitly perform this optimization.
- *Code motion* optimization:
  - Identify a computation that is performed repeatedly, but whose result does not change.
  - Move the computation so that it does not get executed as often.

Example: Loop Inefficiency

```
/* move call to vec_length out of loop */
void combine2(vec_ptr v, data_t* dest) {
  int i;
  int length = vec_length(v);

  *dest = IDENT;
  for(i = 0; i < length; i++) {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OPER val;
  }
}
```

CPEs	int		float-pt		
	+	*	+	F *	D *
`combine1` (-O1)	12.00	12.00	12.00	12.01	13.00
`combine2` (-O1)	8.03	8.09	10.09	11.09	12.08

Clicker Question

What is the total number of function calls in this loop?

(Assume the `x` is 10 and `y` is 100.)

```
for(i = min(x, y); i < max(x, y); incr(&i, 1))
  t += square(i);
```

A. 4

B. between 50 and 100

C. between 101 and 200

D. more than 200

Clicker Question

We express relative performance as a ratio of the form:

$$\frac{T_{\text{old}} = \text{time of the original version}}{T_{\text{new}} = \text{time of the modified version}}$$

Which of the following is true?

A. A ratio of 0 means no improvement, 1 means slight improvement, 2 means significant improvement.

B. The ratio will never be less than 1.

C. The CPEs is 12.00 for `combine1` and 8.03 for `combine2`. Thus, the performance ratio is about 1.5.

D. None are true.

E. More than one of A-C is true.

Reducing Procedure Calls

- Procedure calls incur overhead and block optimizations.
- `get_vec_element` is called on every loop iteration.
  - especially costly procedure call because of bounds checking
  - simple analysis shows all array references to be valid

```
data_t* get_vec_start(vec_ptr v) { return v->data; }

/* direct access to vector data */
void combine3(vec_ptr v, data_t* dest) {
  int i;
  int length = vec_length(v);
  data_t* data = get_vec_start(v);

  *dest = IDENT;
  for(i = 0; i < length; i++)
    *dest = *dest OPER data[i];
}
```

Reducing Procedure Calls

CPEs	int		float-pt		
	+	*	+	F *	D *
`combine2` (-O1)	8.03	8.09	10.09	11.09	12.08
`combine3` (-O1)	6.01	8.01	10.01	11.01	12.02

- How does this transformation affect the modularity?
- The CPE improvement is up to a factor of 1.3X.
  - ratio $T_{old} / T_{new} = 8.03 / 6.01 = 1.34$
  - what is the factor if there is no improvement?
- Modest improvement, but call is bottleneck for future opts.

Compromise modularity and abstraction for speed, if performance is a significant issue.

Reducing Memory References

```
# (x86-64 floating-pt code)
# combine3, data_t is float, OPER is *
# dest in %rbp, data in %rax, i in %rdx, length in %r12
.L498:
  movss (%rbp),%xmmo               # read *dest
  mulss (%rax,%rdx,4),%xmm0        # multiply by data[i]
  movss %xmm0,(%rbp)               # write *dest
  addq  $1, %rdx                   # i++
  cmpq  %rdx,%r12                  # compare i to length
  jg    .L498                      # if i<length, goto loop
```

- The value being computed is accumulated in the location designated by pointer `dest`, memory read/write required.
- Possible to avoid so many reads and writes of memory?
  - value written is read on next iteration

Reducing Memory References

```
/* accumulate result in local variable */
void combine4(vec_ptr v, data_t* dest) {
  int i;
  int length = vec_length(v);
  data_t* data = get_vec_start(v);
  data_t acc = IDENT;

  for(i = 0; i < length; i++)
    acc = acc OPER data[i];
  *dest = acc;
}
```

```
# combine4
# data in %rax, acc in %xmm0,
# i in %rdx, length in %rbp
.L488:
  mulss (%rax,%rdx,4),%xmm0
  addq  $1, %rdx
  cmpq  %rdx,%rdp
  jg .L488
```

(AKA "*scalar replacement*")

CPEs	int		float		
	+	*	+	F *	D *
combine3 (-O1)	6.01	8.01	10.01	11.01	12.02
combine4 (-O1)	2.00	3.00	3.00	4.00	5.00

Will Compiler Reduce Refs?

- Is scalar replacement an optimization the compiler will perform automatically?
  - not in this case (why not? because of potential memory aliasing)
- Consider vector `v = [2,3,5]`, `OPER` is `*`, and calls
  - `combine3(v, get_vec_start(v)+2)` results in `[2,3,36]`
  - `combine4(v, get_vec_start(v)+2)` results in `[2,3,30]`
- An optimizing compiler cannot make a judgment about the conditions under which a function might be used. Thus, it is obliged to preserve its exact functionality.

Loop Unrolling

- Some loops have such a small body that most of the execution time is spent updating the loop-counter variable and testing the loop-exit condition.
- It is more efficient to *unroll* such loops, putting two or more copies of the loop body in a row.
- Then, avoid setting and testing the loop counter in every loop body, reducing "loop overhead".
- How should the new loop update/exit compare to original?

Example: Loop Unrolling

BEFORE

L_1: x ← M[i]
s ← s + x
i ← i + 4
if i < n goto L_1
L_2:

AFTER

L_1: x ← M[i]
s ← s + x
x ← M[i+4]
s ← s + x
i ← i + 8
if i < n goto L_1
L_2:

- Will this work if the original loop iterated an odd number of times?
- How can we accommodate an odd number of iterations?
- How can we modify our strategy to unroll by a factor of *K*?
- Will the optimizing compiler perform loop unrolling automatically?

Summary

- To effectively use optimizing compilers, programmers must know the capabilities and limitations.
- Machine-independent optimizations:
  - code motion
  - reducing procedure calls
  - reducing memory references
  - loop unrolling (its machine dependence to be revisited)
- The programmer does have to help the optimizing compiler by dealing with optimization blockers.

Exercise: Loop Unrolling

```
void inner_prod(vec_ptr u, vec_ptr v, data_t *dest) {
  int i;
  int length = vec_length(u);
  data_t *udata = get_vec_start(u);
  data_t *vdata = get_vec_start(v);
  data_t sum = (data_t) 0;

  for (i = 0; i < length; i++) {
    sum += udata[i] * vdata[i];
  }

  *dest = sum;
}
```

Perform 4-way loop unrolling.