Computer Organization Notes

[5] Optimizing Program Performance

By Sandesh Paudel

Table of Contents:

Optimizing Program Performance

[5.1] Capabilities and Limitations of Optimizing Compilers

- Trade off between speed and {correctness, conciseness , clarity}

- Certain tasks demand speed, like processing video frames.

- Stages of optimizing program efficiency:
  - Use appropriate data structures and algorithms.
  - Write code that the compiler can produce efficient code for.
  - Divide tasks into portions that can be computed in parallel [using multiple cores and processes]

- Many low-level optimizations tend to reduce code readability and modularity.

- **Optimization Blockers:** aspects of programs that can severely limit the opportunities for a compiler to generate optimized code.

  1) Optimization Blocker 1: Memory Aliasing: the case where two pointers may designate the same memory location.

     Consider the following code:

     -Naive implementation: [6 memory references {2 reads of *xp, 2 reads of *yp, 2 writes of *xp}]

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

     -Optimized implementation: [3 memory references {1 read of *xp, 1 read of *yp, 1 write of *xp}]

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

     *The compiler cannot take a function like **twiddle1** and optimize it to be
     Like **twiddle2**.

     Consider the case where xp=yp. Then, **twiddle1** will function as:

```
*xp += *xp;  /* Double value at xp */
*xp += *xp;  /* Double value at xp */
```

     This will **return 4 x *xp**. However, under this circumstance, **twiddle2** will function as:

```
*xp += 2* *xp;  /* Triple value at xp */
```

>Explanation: The compiler knows nothing about how **twiddle1** will be called, so it must assume that xp can equal yp. It therefore cannot generate code in the style of **twiddle2** as an optimized version of **twiddle1**.

>Safe optimizations: In performing only safe optimizations, the compiler must assume that different pointers may be aliased. Another example:

```
x = 1000; y = 3000;
*q = y;    /* 3000 */
*p = x;    /* 1000 */
t1 = *q;   /* 1000 or 3000 */
```

** depending on whether q = p, the value of t1 can be either 1000 or 3000.

2) Optimization Blocker 2: Function calls:

```
long f();

long func1() {
    return f() + f() + f() + f();
}

long func2() {
    return 4*f();
}
```

***func2()** calls **f()** once while **func1()** calls **f() 4 times.** Why doesn't the compiler optimize **func1()** in the style of **func2()?**

> Consider the following code for f():

```
long counter = 0;

long f() {
    return counter++;
}
```

* **f()** has what is called a **side effect**, it **modifies** some part of **the global program state.** The number of time **f()** gets called changes the progra behavior. Most compilers do not try to determine whether a function is free of side effects.

[Among compilers, GCC is considered adequate, but not exceptional, in terms of its optimization capabilities. It performs basic optimizations, but it does not perform the radical transformations on programs that more "aggressive" compilers do. As a consequence, programmers using GCC must put more effort into writing programs in a way that simplifies the compiler's task of generating efficient code.]

> Optimizing function calls by inline substitution: replace function call by the code of the body of the function.

```c
/* Result of inlining f in func1 */

long funclin() {
    long t = counter++; // +0
    t += counter++;     // +1
    t += counter++;     // +2
    t += counter++;     // +3

    return t;

}


/* further optimization of inlined code */

long function1opt(){
    long t = 4 * counter + 6;
    counter += 4;

    return t;
}
```

[5.2] Expressing Program Performance **remember to look at the original program code from the textbook to understand optimization**

- **CPE: cycles per element.** It helps us understand the loop performance of an iterative program at a detailed level. Appropriate for programs that perform a repetitive computation, such as processing pixels in an image or computing the elements in a matrix product.

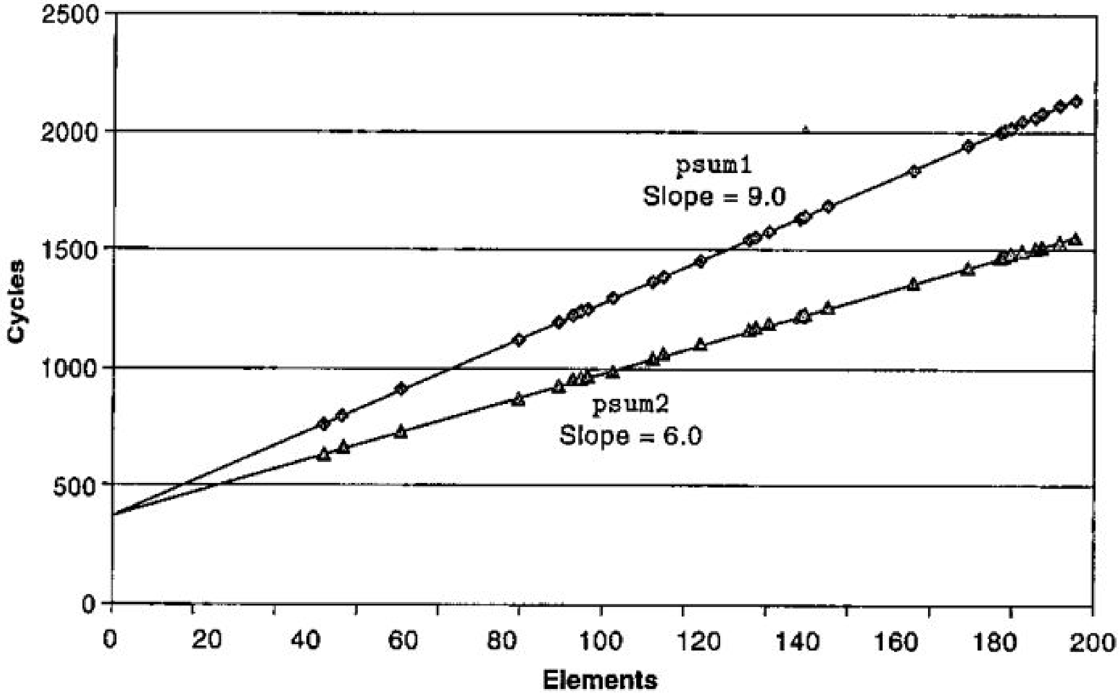
- **Clock:** controls the sequence of activities processed by the processor. Expressed in **Cycles per second.** *Gigahertz*(GHZ) = **billions of cycles per second.**
  - **a "4 GHz"** processor means that the processor clock runs at 4.0 x 10^9 cycles every second.
  - $Time\ for\ each\ cycle\ =\ \frac{1}{frequency}$
  - For programmers, it is more instructive to express measurements in **clock cycles** rather than nanoseconds or picoseconds. This way, the measurements express **how many instructions are being executed** rather than **how fast the clock runs.**

- **Optimization by loop unrolling [naive -> optimized]:**

```c
/* Compute prefix sum of vector a */
void psum1(float a[], float p[], long n)
{
    long i;
    p[0] = a[0];
    for (i = 1; i < n; i++)
        p[i] = p[i-1] + a[i];
}
```

```c
void psum2(float a[], float p[], long n)
{
    long i;
    p[0] = a[0];
    for (i = 1; i < n-1; i+=2) {
        float mid_val = p[i-1] + a[i];
        p[i]     = mid_val;
        p[i+1]   = mid_val + a[i+1];
    }
    /* For even n, finish remaining element */
    if (i < n)
        p[i] = p[i-1] + a[i];
}
```

> **Example of a CPE graph:**



> **Examples of GCC optimization (-g vs -01, -02, -0g):**

Function	Page	Method	Integer +	Integer *	Floating point +	Floating point *
combine1	507	Abstract unoptimized	22.68	20.02	19.98	20.18
combine1	507	Abstract -01	10.12	10.12	10.17	11.14

> **-01 enables a basic set of optimization. This significantly improves the program performance by about a factor of 2 with no effort. It is generally good to get into the habit of enabling some level of optimization.**

> **Unoptimized:**

```
$ gcc [options] [source files] [object files] -o output file
```

```
$ gcc myfile.c -o myfile
$ ./myfile
```

> **Optimized COmpilation Option:**

```
$ gcc -Olevel [options] [source files] [object files] [-o
output file]
```

```
$ gcc -O myfile.c -o myfile
$ ./myfile
```

gcc -O option flag

Set the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

[5.4 Eliminating Loop Inefficiencies: Code Motion]

```
/* Implementation with maximum use of data abstraction */
void combine1(vec_ptr v, data_t *dest)
{
    long i;

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

ıre 5.5 **Initial implementation of combining operation.** Using different decla-

> **Observe** that **vec_length(v)** is called in every iteration, as it is in the test condition for the machine level code. Since **vec_length(v)** stays constant, we can just declare **int length = vec_length(v)** and use **length** constant for the test condition:

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

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

>Resulting Performance Boost:

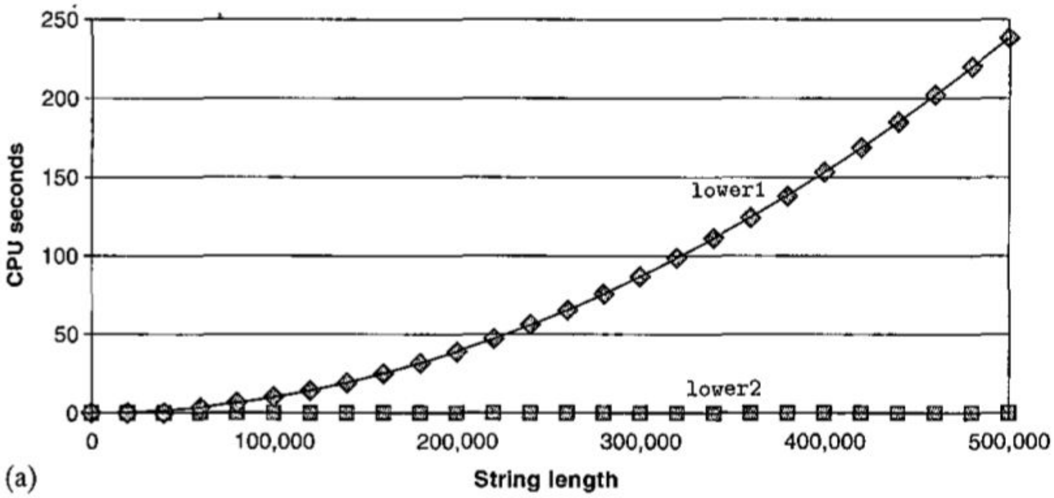
Function	Page	Method	Integer +	Integer *	Floating point +	Floating point *
combine1	507	Abstract -O1	10.12	10.12	10.17	11.14
combine2	509	Move vec_length	7.02	9.03	9.02	11.03

> The optimization above is an instance of general class of optimizations known as Code Motion. They involved identifying a computation that is performed multiple times, but such that the computation will not change. We can therefore move the computation to an earlier section of the code that does not get evaluated as often.

> Another example of Code Motion:

```
/* Convert string to lowercase: slow */
void lower1(char *s)
{
    long i;

    for (i = 0; i < strlen(s); i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

/* Convert string to lowercase: faster */
void lower2(char *s)
{
    long i;
    long len = strlen(s);

    for (i = 0; i < len; i++)
        if (s[i] >= 'A' && s[i] <= 'Z')
            s[i] -= ('A' - 'a');
}

/* Sample implementation of library function strlen */
/* Compute length of string */
size_t strlen(const char *s)
{
    long length = 0;
    while (*s != '\0') {
        s++;
        length++;
    }
    return length;
}
```

> lower1 and lower2 have significant performance disparity:



(a)

> Yet another example of Code Motion:

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

→

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

[5.5 Reducing Procedure Calls]

> Naive:

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

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

> **Optimized:** [move the **get_vec_element()**] out of the loop!**[reduce the number of procedure calls in loops]**

code/opt/vec.c

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

code/opt/vec.c

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

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

Figure 5.9 Eliminating function calls within the loop. The resulting code does not show a performance gain, but it enables additional optimizations.

			Integer		Floating point	
Function	Page	Method	+	*	+	*
combine2	509	Move vec_length	7.02	9.03	9.02	11.03
combine3	513	Direct data access	7.17	9.02	9.02	11.03

-> No apparent performance boost [for now].

[5.6 - Eliminating Unneeded Memory References

> Here is the assembly for the inner loop of combine3:

```
Inner loop of combine3.  data_t = double, OP = *
dest in %rbx, data+i in %rdx, data+length in %rax
.L17:                                   loop:
  vmovsd  (%rbx), %xmm0                   Read product from dest
  vmulsd  (%rdx), %xmm0, %xmm0            Multiply product by data[i]
  vmovsd  %xmm0, (%rbx)                   Store product at dest
  addq    $8, %rdx                        Increment data+i
  cmpq    %rax, %rdx                      Compare to data+length
  jne     .L17                            If !=, goto loop
```

- Couple of observations: (%rbx) = ***dest**, (%rdx) = ***data**

- The accumulated value is read from and written to memory on each iteration. This reading and writing is wasteful [value read from **dest** at the beginning of the iteration should simply be the value written at the end of the previous iteration]

> rewrite the combine3 as combine4 **[use accumulator function instead of pointers]**

```
Inner loop of combine4.  data_t = double, OP = *
acc in %xmm0, data+i in %rdx, data+length in %rax
.L25:                                  loop:
  vmulsd  (%rdx), %xmm0, %xmm0           Multiply acc by data[i]
  addq    $8, %rdx                       Increment data+i
  cmpq    %rax, %rdx                     Compare to data+length
  jne     .L25                           If !=, goto loop
```

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

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

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

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

-Observations: compared to loop in **combine3**, we have reduced the memory operations per iteration from 2 reads and one write to just one single read(**data**). This gives a significant performance boost:

Function	Page	Method	Integer		Floating point	
			+	*	+	*
combine3	513	Direct data access	7.17	9.02	9.02	11.03
combine4	515	Accumulate in temporary	1.27	3.01	3.01	5.01

-Observations: Holding the accumulated value in local variable acc eliminates the need to retrieve it from memory and write back the updated value on every loop iteration.

-Why can't the compiler do this automatically?? -> the functions produce different results if the last element of the vector and the dest are aliased.

[5.7 - Understanding the Microarchitecture]

> the procedures above simply **reduce overhead of procedure calls** and **eliminate “optimization blockers”.**

> **exploiting the processor microarchitecture leads to more performance gains.**

> **Modern microprocessors employ complex and exotic microarchitectures, in which multiple instructions can be executed in parallel, while presenting an operational view of simple sequential instruction execution [instruction level parallelism]**

> **the two different lower bounds:**
- ❖ **Latency bound:** series of operations must be performed in strict sequence, because data dependencies.
- ❖ **Throughput bound:** the raw computing capacity of processor’s functional units.