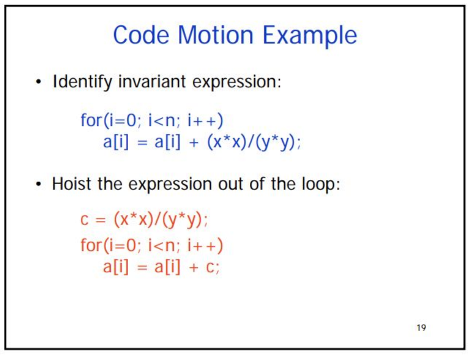
Code Motion:

Reduce frequency with which computation performed

- ▪ If it will always produce same result
- ▪ Especially moving code out of loop



Code Motion Example

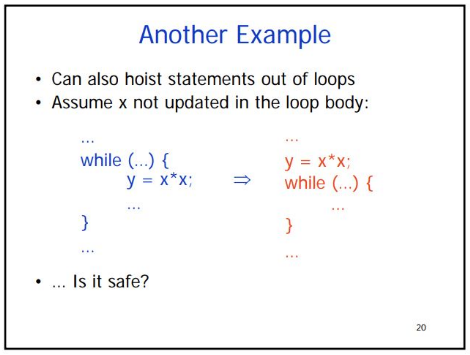
- Identify invariant expression:

```
for(i=0; i<n; i++)
   a[i] = a[i] + (x*x)/(y*y);
```

- Hoist the expression out of the loop:

```
c = (x*x)/(y*y);
for(i=0; i<n; i++)
   a[i] = a[i] + c;
```

19



Another Example

- Can also hoist statements out of loops
- Assume x not updated in the loop body:

```
...                          ...
while (...) {                y = x*x;
      y = x*x;       ⇒       while (...) {
      ...                          ...
}                            }
...                          ...
```

- ... Is it safe?

20

Compiler-Generated Code Motion (-O1)

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 Intel Nehalem, integer multiply requires 3 CPU cycles

Share Common Subexpressions

▪ Reuse portions of expressions ▪ GCC will do this with –O1

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

Loop unrolling:

-2x1

-2x1a

-2x2

Loop unrolling ex:

```
for(i=0; i<n;i=+)
        X[i] = x[i]+s;

for(i=0;i+k-1<n;i+=k)
        X[i] = x[i]+s;
        X[i+1] = x[i+1]+s;
        …;
        X[1+k-1] = x[i+k-1] +s;
```

Open MP:
-It’s an API(an **application programming interface** (**API**) is a set of subroutine definitions, communication protocols, and tools for building software. In general terms, it is a set of clearly defined methods of communication among various components. A good API makes it easier to develop a computer program by providing all the building blocks, which are then put together by the programmer.)
It is a library which lets you run the loops in parallel
OpenMP is a way to program on shared memory devices. This means that the parallelism occurs where every parallel thread has access to all of your data.

You can think of it as: parallelism can happen during execution of a specific for loop by splitting up the loop among the different threads.

Main comments:

```
//Switched order of variables from Stride-N to Stride-1
  //Removed outer loop and processed each plane within the current structure
  //Step two rows and columns at a time to reduce iterations
    //Changed order of for loop, for easier access for cache
    //We are accessing the elements of the 3d matrix in the order they are layed out in memory
    //by accessing each elements in the order it's layed out in memory we are making use of spatial locality because the cache will bring in each of the memory location neighbors and therefore have a cache hit on each successive iteration of the loop
```

Use nested conditionals instead of if statements:

- ? is a ternary operator(it needs 3 operands) it can be used to replace if-else statement
- condition ? expression1 : expression2 (it'll pick the first one(exp1))
- /https://www.tutorialspoint.com/cplusplus/cpp_conditional_operator.html

Stride 1 has good spatial locality because the array is accessed in the same row-major order in which it is stored in memory

Stride N has poor spatial locality because it scans memory with a stride

Use short instead of Int because it uses less memory

-O0 - Reduce [compilation time] and make debugging produce the expected results. This is the default.

-O1 - Optimize. Optimizing compilation takes somewhat more time and a lot more [memory] for a large function.

-O2 - Optimize even more. GCC performs nearly all supported optimizations that do not involve a [space‑speed tradeoff]. As compared to -O, this option increases both compilation time and the performance of the generated code.

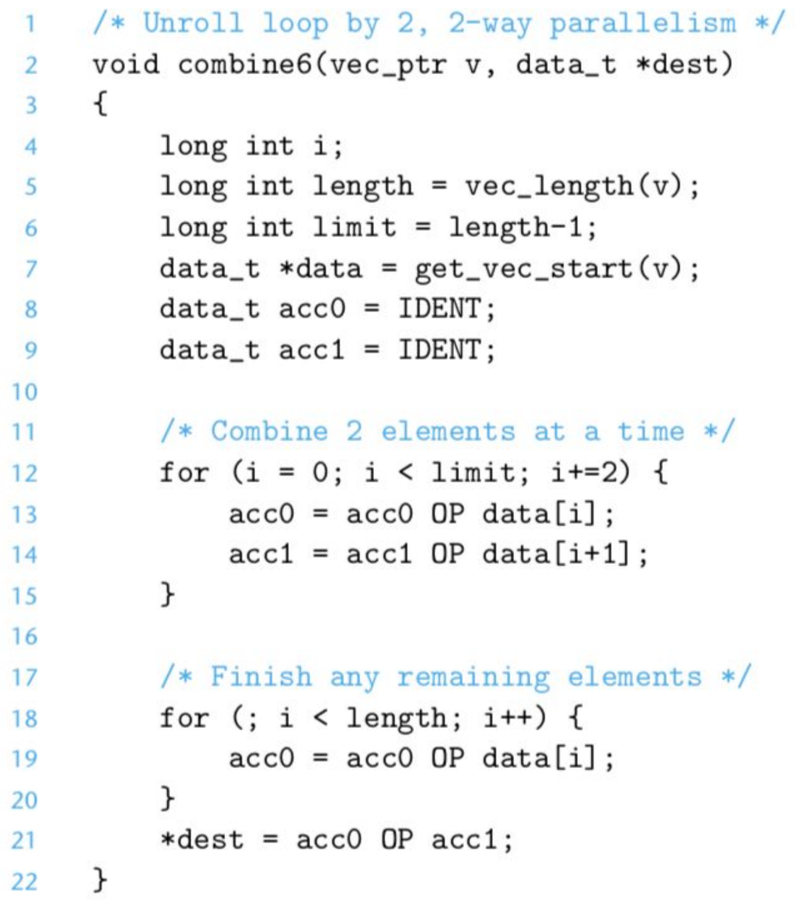
-O3 - Optimize yet more. Turn on all optimizations specified by [‑O2] and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-vectorize, -fvect-cost-model, -ftree-partial-pre and -fipa-cp-clone options.

-Os - Optimize for [size].

-Ofast - Disregard strict standards compliance. -Ofast enables all [‑O3] optimizations. It also enables optimizations that are not valid for all standard-compliant programs. It turns on -ffast-math and the Fortran-specific -fno-protect-parens and -fstack-arrays.

Inline functions have the same semantics as regular functions., Inline functions can allow the compiler to perform additional optimizations, Inline functions save time by eliminating call / return.

Parallel accumulator



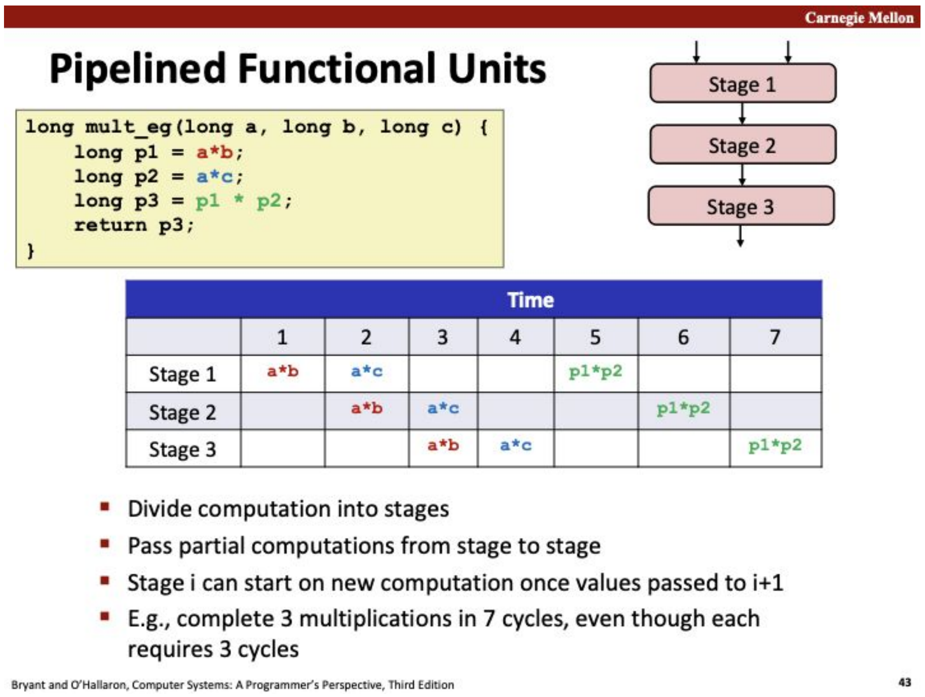
```
/* Unroll loop by 2, 2-way parallelism */
void combine6(vec_ptr v, data_t *dest)
{
    long int i;
    long int length = vec_length(v);
    long int limit = length-1;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;

    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        acc0 = acc0 OP data[i];
        acc1 = acc1 OP data[i+1];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 OP data[i];
    }
    *dest = acc0 OP acc1;
}
```

Figure 5.21 **Unrolling loop by 2 and using two-way parallelism.** This approach makes use of the pipelining capability of the functional units.

Pipelining

- A key feature of pipelining is that it increases the *throughput* of the system, that is, the number of customers served per unit time, but it may also slightly increase the *latency*, that is, the time required to service an individual customer.

- Pipelining improves the throughput performance of a system by letting the different stages operate concurrently.
- https://www.quora.com/Computer-Architecture-Whats-the-difference-between-pipelining-and-parallelism



Carnegie Mellon

Pipelined Functional Units

```
long mult_eg(long a, long b, long c) {
    long p1 = a*b;
    long p2 = a*c;
    long p3 = p1 * p2;
    return p3;
}
```

Stage 1

Stage 2

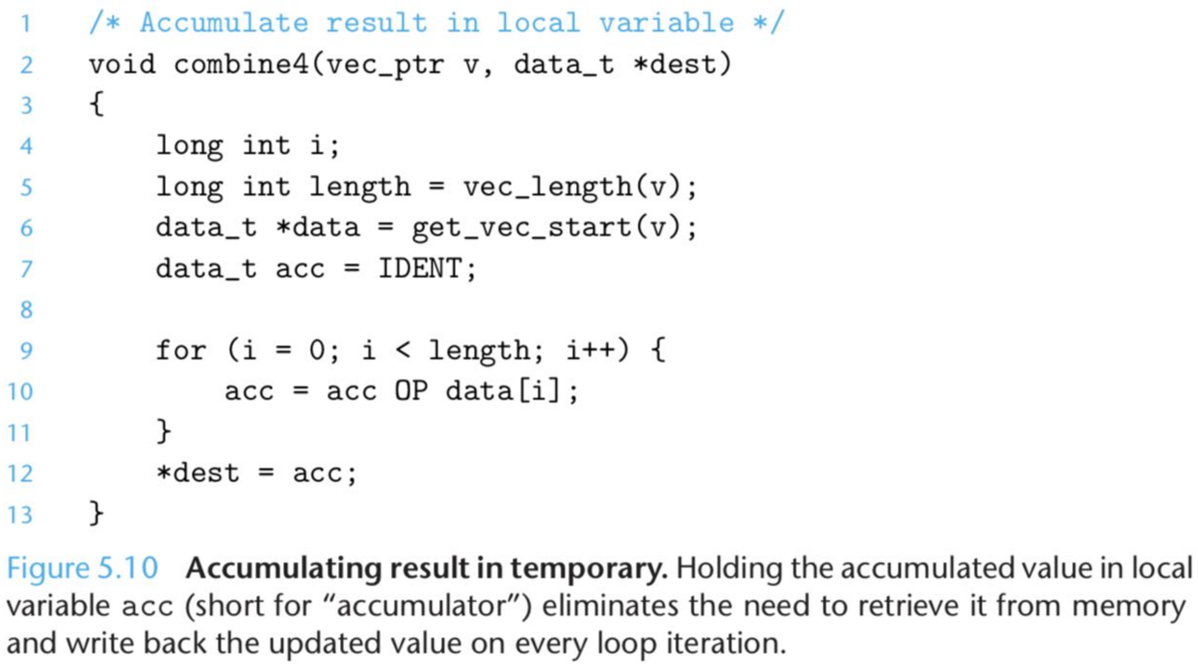
Stage 3

	Time						
	1	2	3	4	5	6	7
Stage 1	a*b	a*c			p1*p2		
Stage 2		a*b	a*c			p1*p2	
Stage 3			a*b	a*c			p1*p2

- Divide computation into stages
- Pass partial computations from stage to stage
- Stage i can start on new computation once values passed to i+1
- E.g., complete 3 multiplications in 7 cycles, even though each requires 3 cycles

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 43

Accumulating in temporary



```
/* Accumulate result in local variable */
void combine4(vec_ptr v, data_t *dest)
{
    long int i;
    long int 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;
}
```

Figure 5.10 **Accumulating result in temporary.** Holding the accumulated value in local variable acc (short for "accumulator") eliminates the need to retrieve it from memory and write back the updated value on every loop iteration.

Re-association transformation (type of parallelism)

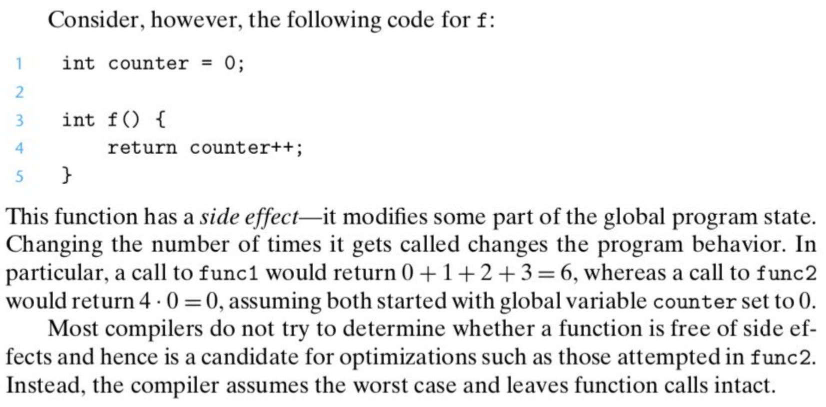
- acc = (acc OP data[i]) OP data[i+1]; while in combine it is performed by the statement
- acc = acc OP (data[i] OP data[i+1]);

- differing only in how two parentheses are placed. We call this a *reassociation trans- formation*, because the parentheses shift the order in which the vector elements are combined with the accumulated value acc.

Inline:

‘

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

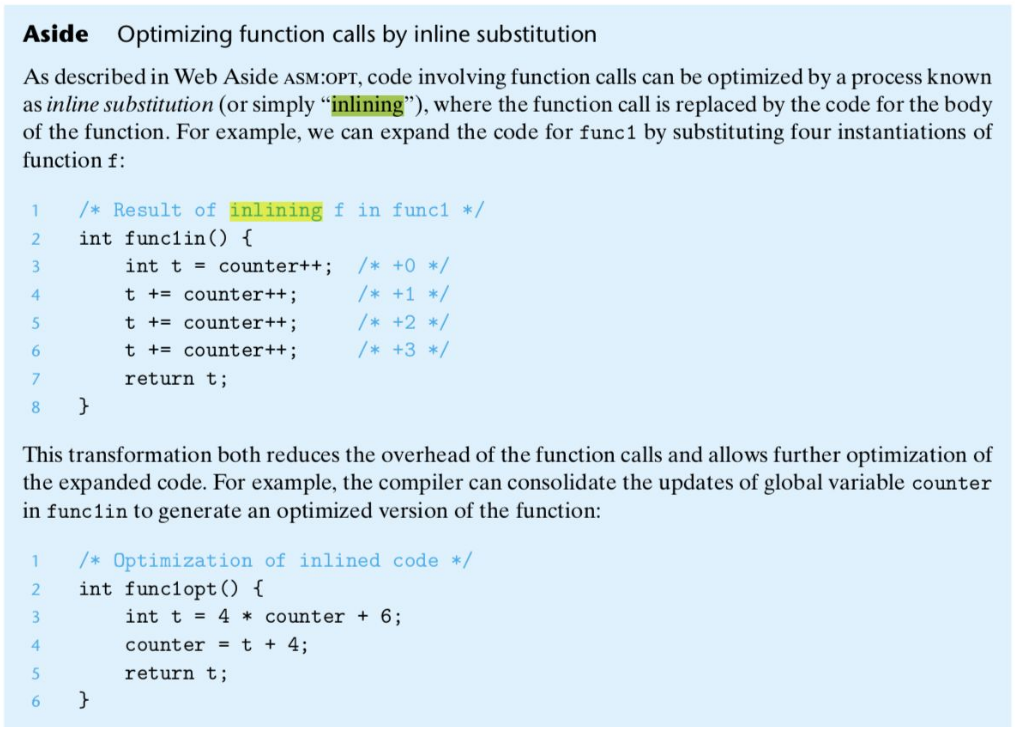


Consider, however, the following code for `f`:

```
int counter = 0;

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

This function has a *side effect*—it modifies some part of the global program state. Changing the number of times it gets called changes the program behavior. In particular, a call to `func1` would return $0+1+2+3=6$, whereas a call to `func2` would return $4 \cdot 0 = 0$, assuming both started with global variable `counter` set to 0.

Most compilers do not try to determine whether a function is free of side effects and hence is a candidate for optimizations such as those attempted in `func2`. Instead, the compiler assumes the worst case and leaves function calls intact.



Aside Optimizing function calls by inline substitution

As described in Web Aside ASM:OPT, code involving function calls can be optimized by a process known as *inline substitution* (or simply "inlining"), where the function call is replaced by the code for the body of the function. For example, we can expand the code for `func1` by substituting four instantiations of function `f`:

```
/* Result of inlining f in func1 */
int func1in() {
    int t = counter++;   /* +0 */
    t += counter++;      /* +1 */
    t += counter++;      /* +2 */
    t += counter++;      /* +3 */
    return t;
}
```

This transformation both reduces the overhead of the function calls and allows further optimization of the expanded code. For example, the compiler can consolidate the updates of global variable `counter` in `func1in` to generate an optimized version of the function:

```
/* Optimization of inlined code */
int func1opt() {
    int t = 4 * counter + 6;
    counter = t + 4;
    return t;
}
```

If two instructions of the same type are executed one immediately after another, for a single functional unit, they will be pipelined faster.

The correct answer is: leaq 8(%rbx), %rdi
leaq 16(%ebc), %rbi
movq (%rdi), %rsi
movq (%rbi),%rax
sar $4,%rsi
sal $2,%rax
addq %rsi,%rax

Locality:

•Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

•

•Temporal locality:

•Recently referenced items are likely

to be referenced again in the near future

•

•Spatial locality:

•Items with nearby addresses tend

to be referenced close together in time