H2 V. Optimizing Program Performance

- How to write efficient programs?
  1. appropriate set of algorithms and data structures
  2. write source code that the compiler can effectively optimize to turn into efficient executable code (CH.12)
     - need to understand the capabilities and limitations of optimizing compilers (pointer arithmetic, casting...)
  3. divide a task into portions that can be computed in parallel (multiple cores and multiple processors)

- To optimize written programs...
  1. eliminate unnecessary work (function calls, conditional tests, memory references)
  2. **instruction-level parallelism** (executing multiple instructions simultaneously)
  3. Use code ***profilers*** -- tools that measure the performance of different parts of a program

H3 §5.1 Capabilities and Limitations of Optimizing Compilers

- Most compilers provide users with some control over **which optimizations** they apply
  - ***Optimization level***
    - -Og : basic set of optimizations
    - -O1 or higher (-O2, -O3) causes more extensive optimizations
      - may expand the program size
      - may make the program harder to debug with standard debugging tools
- In CSAPP, will mostly cosider code compiled with -O1

  c.f. -O2 is standard for most software projects

- Compilers must be careful to apply only **safe** optimizations to a program
  - resulting porgram will have **the exact same behavior** as would an **unoptimized version** for all possible cases the program my encounter
- **Programmer** must make more of an effort to write program in a way that the **compiler can then transform into efficient machine-level code**

H6 *Optimization Blockers*

1. ***Memory aliasing***

```
1  void twiddle1(long *xp, long *yp) {
2    *xp += *yp;
3    *xp += *yp;
4  }
5
6  void twiddle2(long *xp, long *yp) {
7    *xp += 2* *yp;
8  }
```

- both procedures seem to have identical behavior
- If yes, ***twiddel2*** is more efficient
  - twiddle2 - **3 memory references** (read *xp, read *yp, write *xp)
  - twiddle1 - **6 memory references** (2x read *xp, 2x read *yp, 2x write *xp)
- **However,** when **xp and yp are equal**
  - twiddle2 **triples value at xp**
  - twiddle1 **quadruples value at xp**
- Compiler **cannot generate** code in the style of **twiddle2** as an optimized version of twiddle1

- Memory aliasing - two pointers may designate the same memory location
  - Compiler **must assume** that **different pointers may be aliased**

```
1  x = 1000;   y = 3000;
2  *q = y;   // 3000
3  *p = xl;  // 1000
4  t1 = *q;  // 1000 or 3000
5              // 1000 if aliased
```

  - Severely limit the opportunities for a compiler to generate optimized code

2. ***Function calls***

```
long f();

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

long func2() {
  return 4*f();
}
// seems func1 and func2 compute the same result

/* What if f() does... */
long counter = 0;

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

- Function ***f*** has a ***side effect*** -- modifies some part of the **global program state**
- Most copmilers do not try to determine whether a function is **free of side effects**

  --> leave function calls **intact**

c.f. **Optimizing function calls by inline substitution**

- function call is replaced by the code for the body of the function

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

// after inlining, function can be optimized further
long func1opt() {
  long t = 4 * counter + 6;
  counter += 4;
  return t;
} // reproduces the exact save behavior of func1
```

- recent versions of **GCC** attempts this form of optimization (when -finline or -O1 or higher optimization level selected)
  - attempts inlining for functions defined within a single file
- When the code will be evaluated using a **symbolic debugger (GDB),** better **not** to use **inline substitution**
  - tracing, setting breakpoints do not work properly

H3 §5.2 Expressing Program Performance

- ***cycles per element (CPE)*** will be used to express program performance in a way that can guide us in improving the code
  - helps us understand the **loop performance** of an iterative program at a detailed level
  - work well with programs that perform a **repetitive computation** (processing pixels or computing matrix products)

- but why **CPE**?
  - sequencing of activities in controlled by a clock providing a **regular signal of some frequency** (in GHz)
    - when the system uses "4GHz" processor, processor clock runs at 4.0×10^9 cycles per second
  - it is instructive for us programmers to use clock cycles rather than nanoseconds or picoseconds
    - focus more on how many instructions are being executed rather than how fast the clock runs

- Example code
  - functions **psum1** and **psum2** both compute the ***prefix sum*** of a vector of length **n**

For a vector $\vec{a} = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, the prefix sum $\vec{p} = \langle p_0, p_1, \ldots, p_{n-1} \rangle$ is defined as

$$p_0 = a_0$$

$$p_i = p_{i-1} + a_i, \ 1 \leq i < n$$

```
1  /* Compute prefix sum of vector a */
```

```
2  void psum1(float a[], float p[], long n) {
3    long i;
4    p[0] = a[0];
5    for (i = 1; i < n; i++)
6      p[i] = p[i-1] + a[i];
7  }
8
9  void psum2(float a[], float p[], long n) {
10   long i;
11   p[0] = a[0];
12   for (i = 1; i < n-1; i+=2) {
13     float mid_val = p[i-1] + a[i];
14     p[i] = mid_val;
15     p[i+1] = mid_val + a[i+1];
16   }
17   // For even n, finish remaining elem
18   if (i < n)
19     p[i] = p[i-1] + a[i];
20 }
```

- **psum2** uses an optimization technique called ***loop unrolling*** --> computing two elements per iteration
- time required by such a procedure can be characterized as a **Constant + a factor proportional to the number of elements**
  - e.g. **psum1** = $368 + 9.0n$, **psum2** = $368 + 6.0n$

    --> 368 cycles (timing code + initiating procedure, setting up the loop, completing the procedures) + linear factor of **6.0 or 9.0** cycles per element

    --> for large values of n, the run times will be dominated by the linear factors
  - **psum1** : CPE of 9.0, **psum2** : CPE of 6.0

H3 §5.3 Program Example

- To demonstrate how an abstract program can be systematically transformed into more efficient code, will use the **vector data structure**

- Header for defining vectors

```c
// vec.h
/* Create abstract data type for vector */
typedf struct {
  long len;
  data_t *data;
} vec_rec, *vec_ptr;
```

- Generating vectors, accessing vector elements, and determing vector length

```c
// vec.c
/* Create vector of specified length */
vec_ptr new_vec(long len) {
  // Allocate header structure
  vec_ptr result = (vec_ptr) malloc(sizeof(vec_rec));
  data_t *data = NULL;
  if(!result)
    return NULL;  // couldn't allocate storage
  result->len = len;

  // Allocate array
  if(len > 0) {
    data = (data_t *)calloc(len, sizeof(data_t));
    if(!data) {
      free((void *) result);
      return NULL; // couldn't allocate storage
    }
  }
  // Data will either be NUll or allocated arr
  result->data = data;
  return result;
}

/*
 * Retrieve vector element and store at dest
 * Return 0 (out of bounds) or 1 (successful)
 */
int get_vec_element(vec_ptr v, long index, data_t *dest) {
  if(index < 0 || index >= v->len)
    return 0;
  *dest = v->data[index];
  return 1;
}
```

```
34
35  /* Return length of vector */
36  long vec_length(vec_ptr v) {
37    return v->len;
38  }
```

- Note that get_vec_element performs **bounds checking** for every vector reference
  - reduces the chance of program error but **can slow down** program execution

- Initial implementation of combining operation

```
1   /* Implementation with maximum use of data abstraction */
2
3   // Use one of the two definitions below
4     // For addition
5   #define IDENT 0
6   #define OP +
7
8     // For multiplication
9   #define IDENT 1
10  #define OP *
11
12  void combine1(vec_ptr v, data_t *dest) {
13    long i;
14
15    *dest = IDENT;
16    for(i = 0; i < vec_length(v); i++) {
17      data_t val;
18      get_vec_element(v, i, &val);
19      *dest = *dest OP val;
20    }
21  }
```

- Combines all of the elements in a vector into a single value according to some operation (addition or multiplication)

- **CPE** performance of the functions to be measured with an **Intel Core i7 Haswell processor** (CSAPP reference machine)

- CPE measurements for **combine1**

```
                        Integer       Floating pt
                        ---------------------------
Function  Method        +     *       +      *
-------------------------------------------------
combine1  unoptimized  22.68 20.02    19.98  20.18
combine1  Abstract -O1 10.12 10.12    10.17  11.14
```

--> with command-line option -O1, program performance significantly improved (more than 2x)

H3 §5.4 Eliminating Loop Inefficiencies

- Test condition **must be** evaluated on every iteration of the loop
  - in ***combine1***, length of the vector does not change as the loop proceeds
    - compute the vector length only once and use this in test condition!

```c
/* 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;
  }
}
```

--> calls **vec_length** at the beginning and assigns the result to a local variable **length**

```
                        Integer       Floating pt
                        ---------------------------
Function  Method        +     *       +      *
-------------------------------------------------
combine1  Abstract -O1 10.12 10.12    10.17  11.14
combine2  move vec_len  7.02  9.03     9.02  11.03
```

H6 Code Motion

- identifying a computation that is performed multiple times, **but such that the result of the computation will not change**
  - move the computation to an earlier section that does not get evaluated as often
- Optimizing compilers attempt to perform code motion
  - but they assume that function has side effects

    --> programmer must aid compiler by explicitly performing code motion

[Extreme 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;
}
```

- for a string of length n, **strlen** takes time proportional to **n**
  - since **strlen** is called in each of the **n** iterations of **lower1** , the overall run time of

lower1 is **quadratic** in the string length ==> proportional to n^2

- **Code motion can become a major performance bottleneck**

H3 §5.5 Reducing Procedure Calls

- **procedure calls** can incur overhead and block most forms of program optimizations
- in **combine2**, ***get_vec_element*** is called on every loop iteration to retrieve the next vector element
  - ***get_vec_element*** checks the vector index **i** against the loop bounds with every vector referecnce --> inefficiency

- Hence, suppose instead we add a function **get_vec_start** to our abstract data type

```
1  data_t *get_vec_start(vec_ptr v) {
2    return v->data;
3  }
4
5  /* Direct access to vector data */
6  void combine3(vec_ptr v, data_t *dest) {
7    long i;
8    long length = vec_length(v);
9    data_t *data = get_vec_start(v);
10
11   *dest = IDENT;
12   for(i = 0; i < length; i++) {
13     *dest = *dest OP data[i];
14   }
15 }
```

--> No function calls in the inner loop

--> Rather than making a function call to retrieve each vector element, it **accesses the array directly**

- However, in this case, there is **no apparent performance improvement**
  - but it is one of a series of steps that will **ultimately lead** to greatly improved performance

H3 §5.6 Eliminating Unneeded Memory References

- **combine3** accumulates the value being computed by the combining operation at the location designated by the pointer **dest**

[X86-64 code]

```
1  # Inner loop of combine3.   data_t = double, OP = *
2  # dest in %rbx, data+i in %rdx, data+length in %rax
3  .L17:                          #loop:
4    vmovsd  (%rbx), %xmm0          # Read pd from dest
5    vmulsd  (%rdx), %xmm0, %xmm0   # Multip pd by data[i]
6    vmovsd  %xmm0, (%rbx)          # Store pd at dest
7    addq    $8, %rdx               # Increment data+i
8    cmpq    %rax, %rdx             # Compare to data+ln
9    jne     .L17                   # If !=, goto loop
```

- accumulated value is read from and written to memory on each iteration --> wasteful! (since the value read from dest at the beginning of each iteration should simply be the value written at the end of the previous iteration)

- We can optimize the code by using a **temporary variable acc** that is used in the loop to accumulate the computed value
- [**Combine 4** - use of temp var]

```
1  /* Accumulate result in local variable */
2  void combine4(vec_ptr v, data_t *dest) {
3    long i;
4    long length = vec_length(v);
5    data_t *data = get_vec_start(v);
6    data_t acc = IDENT;
7    /* Instead of using *dest, use temp var acc in loop */
8
9    for (i = 0; i < length; i++) {
10     acc = acc OP data[i];
11   }
12   *dest = acc;
13 }
```

- [Generated assembly code (**Inner loop**)]

```
# 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  # Multip acc by data[i[]]
  addq    $8, %rdx              # Increment data+i
  cmpq    %rax, %rdx            # Compare to data+length
  jne     .L25                  # If !=, goto loop
```

```
                        Integer       Floating pt
                        ---------------------------
Function  Method        +     *         +      *
-----------------------------------------------------
combine3  Direct access 7.17  9.02     9.02  11.03
combine4  acc in temp   1.27  3.01     3.01   5.01
```

H3 §5.7 Understanding Modern Processors

- to push the performance further, we must consider optimizations that exploit the ***microarchitecture*** of the processor
  - underlying system design by which a processor executes instructions
- general principles of operation and optimization apply to a wide variety of machines (yet, detailed performance results may not hold for other machines)

- at the **code level**, it appears as if instructions are **executed one at a time** (fetching values from registers or memory, performing an operation, and storing results back to a register or memory location)
- **BUT, in the ACTUAL PROCESSOR, a number of instructions are evaluated *simulatneously*** --> **Instruction-level parallelism**
  - can be 100 or more instructions run in parallel

- **Two different lower bounds** to maximum performance
  1. ***Latency bound*** : when a series of operations **must be** performed in **strict sequence**
     - when the result of one operation is required before the next one can begin

2. ***Throughput bound***: raw computing capacity of the processor's functional units **(ULTIMATE limit on program performance)**

H5 ¶5.7.1 Overall Operation

- recent **Intel processors** = **Superscalar**
  - can perform multiple operations on every clock cycle & out of order (don't need to correspond o their ordering in the machine-level program)
- **Overall Design** = **ICU** (*Instruction control unit*) + **EU** (execution unit)
  - **ICU** - reads a sequence of instructions from memory and generates a set of primitive operations to perform on program data
    - reads the instructions from an **instruction cache** (special high speed memory containing the most recently accessed instructions)
    - fetches **well ahead** of the currently executing instructions
  - **EU** - executes these operations

--> **out-of-order** processors require far greater & more complex hardware but they are better at achieving higher degrees of **instruction-level parallelism**

- When a program **hits a branch** -- two possible directions the program might go
  - Modern processors emply ***Branch prediction***
    - guess whether or not a branch will be taken and predict the target address for the branch
    - begins executing these operations before it has been determined
    - If the **branch was predicted incorrectly, resets the state to that at the branch point** and begins fetching and executing instructions in the **other direction**
    - ***Fetch control*** (in ICU) incorportates branch prediction to perform the task of determining which instructions to fetch

- ***Instruction decoding logic*** takes the actual program instructions and converts them into a **set of primitive operations** (micro-operations)
  - each of these operations performs some simple computational task (adding two numbers, reading data from memory, writing data to memory)
  - for machines with **complex instructions** (x86 processors), instruction can be **decoded into multiple operations**
    - how instructions are decoded varies between machines
    - can **optimize** our programs without knowing the low-level details of a particular machine implementation

```
addq  %rax, %rdx
```

--> converted into a single operation

```
addq  %rax, 8(%rdx)
```

--> yields multiple oprations, **separating the memory references** from the arithmetic operations

1. ***load*** a value from memory into the processor
2. ***add*** the loaded value to the value in register %eax
3. ***store*** the result back to memory

--> allow a division of labor among a set of dedicated hardware units (**multiple instructions in parallel**)

- **EU** (Execution Unit) then receives operations from the instruction fetch unit
  - receives a number of operations on each clock cycle
  - operations are dispatched to a set of ***functional units*** (specialized to handle different types of operations)
    1. **Load unit** - **reads** data from the memory into the processor
       - has an **adder** to perform **address computations**
    2. **Store unit** - **writes** data from the processor to the memory
       - has an **adder** to perform **address computations**

    --> Load unit & Store unit access memory via ***data cache*** (high-speed memory containing the most recently accessed data values)

    3. **Arithmetic operations unit**
       - typically **specialized to perform** different combinations of **integer & floating-point operations**
       - as # of transistors ↑↑

         ==> total # of functional units & combinations of operations each unit can perform & performatnce of each of these units ↑↑

       --> Designed to be able to perform a variety of different operations (If one functional unit were specialized to perform specific operation, would not get the full benefit of having multiple functional units)

    4. **Branch unit**

- **final results are not stored in the program registers or data memory until the processor can be certain that these instructions should actually have been executed**
- **determines** whether or not **branch operations were predicted correctly**
  - If the prediction was **incorrect** , **EU discards** the results that have been computed **beyond the branch point** & **signals the branch unit** that hte prediction was incorrect and **indicate the correct branch destination**

- **Intel Core i7 Haswell** has eight functional units (numbered 0-7) -- partial list of each one's capabilities --
  0. Integer arithmetic, floating-point multiplication, integer and floating-point division, branches
  1. Integer arithmetic, floating-point addition, integer multiplication, floating-point multiplication
  2. Load, address computation
  3. Load, address computation
  4. Store
  5. Integer arithmetic
  6. Integer arithmetic, branches
  7. Store address computation

--> Combination of functional units has the potential to perform **multiple operations of the same type simultaneously** (·.· 4 units for integer operations, 2 units for load operations, 2 for floating-point multiplication)

- ***Retirement unit*** keeps track of the ongoing processing and makes sure that it obeys the sequential semantics of the machine-level program
  - controls the updating of **register files**
  - as an instruction is decoded, information about instruction is placed into a **FIFO queue**, and remains until...
    1. operations for the instructions have completed and any branch points leading to this instruction are confirmed as having been **correctly predicted** --> instruction can be **retired** , with any updates to the program registers
    2. If **mispredicted,** instruction will be **flushed** , discarding any results that may have been computed --> **will not alter the program state**

- to **expedite the communication of results** from one instruction to another, information is **exchanged among the execution units** (operation results)

- **execution units can send results directly to each other**
- ***Register renaming***
  - when an instruction that updates **register r** is decoded, a **tag t** is generated, giving a unique identifier to the result of the operation
    - **(r, t)** is added to a **table** maintaining the **association between program register r and tag t** for an operation that will update this register
  - when some execution unit **completes** the first operation, it generates a result **(v, t)**, indicating that the operation with **tag t produced value v**
    - any operation waiting for **t** as a source will then use **v as the source value** (***data forwarding***)
      - values can be forwarded directly from one operation to another (without registers), enabling the second operation to begin as soon as the first has completed
  - with **register renaming**, an entire sequence of operations can be performed **speculatively,** even though the registers are updated only after the processor is certain of the branch outcomes (∵ information is passed directly within execution unit)

H5 ¶5.7.2 Functional Unit Performance

[Performance of arithmetic operations for **i7 Haswell**]

	Integer			FP		
Op	Latency	Issue	Capac	Latency	Issue	Capac
+	1	1	4	3	1	1
*	3	1	1	5	1	2
/	3-30	3-30	1	3-15	3-15	1

- **Latency**: total time required to perform the operation
  - latencies increase in goring from integer to floating-point operations

- **Issue time**: minimum number of clock cycles between two independent operations of the same type
  - addition & multiplication operations all have issue time of 1

- on each clock cycle, the processor can start a new one of these operations
- achieved through the use of ***pipelining***
    - **pipelined function unit** is implemented as a series of **stages** (performs part of the operation)

      e.g. floating-point adder contains 3 stages (3-cycle latency)

      1. process the exponent values
      2. add the fractions
      3. round the result
    - arithmetic operations can proceed through the stages in close succession (rather than waiting for one operation to complete before the next begins)
        - only if there are successive, logically independent operations to be performed
- funtional units with issue times of 1 cycle are said to be ***fully pipelined*** --> can start a new operation every clock cycle
- **divider is not pipelined** --> issue time = latency
    - divider **must perform** a complete division before it can begin a new one
    - comparatively costly operation

- **Capacity**: number of functional units capable of performing the operation
    - operations with capacity greater than 1 = due to the capabilities of the **multiple functional units**

- **Throughput** of the unit?
    - more common way of expressing issue time = specifying the maximum **throughput** of the unit
        - **reciprocal of the issue time**
        - **fully pipelined functional** unit has a maximum throughput of **1 operation per clock cycle** , while **units with higher issue times** have **lower maximum throughput**
        - having **multiple functional units can increase throughput**

      operation with capacity C and issue time I can potentially be achieved a throughput of C/I operations per clock cycle

- CPU designers must carefully balance the number of functional units and their individual performance to achieve optimal overall performance (∵ limited amount of space on the microprocessor chip)

- in the design of the Core i7 Haswell processor...
  - integer multiplication and floating-point multiplication and addition were considered important operations

    --> most resources dedicated
  - division is relatively infrequent and difficult to implement with either short latency or full pipelining

- ***latency bound*** -- minimum value for the CPE for any function that must perform the combining operation in a strict sequence
- ***throughput bound*** -- minimum bound for the CPE based on the maximum rate at which the functional units can produce results

H5 ¶5.7.3 An Abstract Model of Processor Operation

- to analyze the performance of a machine-level program executing on a modern processor, we use **data-flow** representation of programs, showing **data dependencies** between the different operations **constrain the order** in which they are executed
  - constraints lead to ***critical path*** in the graph, putting a lower bound on the number of clock cycles required to execute a set of machine instructions

c.f. computing the product or sum of n elements requires

$\approx L \cdot n + k$ clock cycles where L is the latency of the combining operation and K represents the overhead of calling the function and initiating and terminating the loop

==> CPE is equal to the latency bound L

- For a code segment forming a **loop**, we can classify the registers that are accessed into **four categories**
  1. **Read-only**
     - used as source values (as data or to compute memory addresses)
     - **not modified within the loop**
  2. **Write-only**
     - used as the **destinations** of data-movement operations
  3. **Local**

- updated and used within the loop, but there is **no dependency from one iteration to another**
- e.g. **Condition code registers**

4. **Loop**
   - used both as **source values** and as **destinations** for the loop
   - value generated in one iteration being used in another

- **For all of the cases where the operation has a latency L greater than 1, we see that the measured CPE is simply L**, indicating that **this chain forms the performance-limiting critical path**

- critical paths in a **data-flow representation** provide ***lower bound*** on how many cycles a program will require
  - **other factors can also limit performance** (total number of functional units available & number of data vaues that can be passed among the functional units on any given step)
  - e.g. **Integer addition** -- data operation is sufficiently fast that the rest of the operations cannot supply data fast enough

H3 §5.8 Loop Unrolling

- program transformation that reduces the number of iterations for a loop by **increasing the number of elements computed on each iteration**
- loop unrolling can improve performance in two ways...
  1. **reduces** the **number of operations that do not contribute directly** to the program result
     - loop indexing / conditional branching
  2. **exposes** ways in which we can further transform the code to **reduce the number of operations in the critical paths** of the overall computation

[Example C code, **2 x 1 loop unrolling**]

```
1  /* 2 x 1 loop unrolling */
2  void combine5(vec_ptr v, data_t *dest) {
3    long i;
4    long length = vec_length(v);
5    long limit = length-1;
```

```
6     data_t *data = get_vec_start(v);
7     data_t acc = IDENT;
8
9     /* Combine 2 elements at a time */
10    for (i = 0; i < limit; i+=2) {
11      acc = (acc OP data[i]) OP data[i+1];
12    }
13
14    /* Finish any remaining elements */
15    for (; i < length; i++) {
16      acc = acc OP data[i];
17    }
18    *dest = acc;
19  }
```

--> first loop steps through the array **two elements at a time** , combining operation is applied to array elements i and $i+1$ in a single iteration

--> since the vector length will not always be a multiple of 2, second loop operates on the remaining element(s)

--> can generalize the idea to **unroll a loop by any factor** k --> yielding $k \times 1$ ***loop unrolling*** (unroll by a factor of k but accumulate values in a single variable acc)

1) set the upper limit to be $n-k+1$, within the loop apply the combining operation to elements i through $i+k-1$

2) loop index i is incremented by k in each iteration

3) maximum array index $i+k-1$ will be less than n

4) include the second loop to step through the final few elements of the vector one at a time

[Performance of unrolled code for unrolling factors $k=2$ and $k=3$]

```
                        Integer       Floating pt
                        ---------------------------
Function  Method        +     *       +      *
-------------------------------------------------
combine4  No unrolling  1.27  3.01    3.01   5.01
combine5  2x1 unrolling 1.01  3.01    3.01   5.01
          3x1 unrolling 1.01  3.01    3.01   5.01

Latency bound           1.00  3.00    3.00   5.00
Throughput bound        0.50  1.00    1.00   0.50
```

- CPE for **integer addition improves**, achieving the **latency bound of 1.00**
  - benefits of **reducing loop overhead operations**
    - reducing the number of overhead operations relative to the number of additions required to compute the vector sum, **can reach the point where 1-cycle latency of integer addition becomes the performance-limiting factor**
  - **none go below their latency bounds** (even for bigger unrolling factors)
    - ($\because 1/k$ iterations of nk operations per iteration --> critical path of n operations is the **limiting factor** for the performance)

[**Assembly code for the inner loop of combine5** ($k = 2$, data_t = double)]

```
# Inner loop of combine5. data_t = double, OP = *
# i in %rdx, data %rax, limit in %rbx, acc in %xmm0
.L35:                                  # loop:
  vmulsd  (%rax,%rdx,8), %xmm0, %xmm0   # Multiply acc by data[i]
  vmulsd  8(%rax,%rdx,8), %xmm0, %xmm0  # Multiply acc by data[i+1]
  addq    $2, %rdx                      # Increment i by 2
  cmpq    %rdx, %rbp                    # Compare to limit:i
  jg      .L35                          # If >, goto loop
```

H3 §5.9 Enhancing Parallelism

- **functional units** performing **addition** and **multiplication** are fully pipelined --> can start **new operations every clock cycle**
- some operations can be performed by **multiple functional units**

--> has the potential to perform additions and multiplications at a higher rate

--> but in **combine5**, we used a single variable **acc** to accumulate the values. If we use more variables...?

H5 ¶5.9.1 Multiple Accumulators

- improve performance by **splitting the set of combining operations into two or more parts and combining the results at the end**

  Let P_n denote the product of elements $a_0, a_1, \ldots, a_{n-1}$:

  $$P_n = \prod_{i=0}^{n-1} a_i$$

  Assuming n is even, we can write this as $P_n = PE_n \times PO_n$, where PE_n is the product of the elements with even indices, and PO_n is the product of the elements with odd indices:

  $$PE_n = \prod_{i=0}^{n/2-1} a_{2i}$$
  $$PO_n = \prod_{i=0}^{n/2-1} a_{2i+1}$$

[Example C code, **2 x 2 loop unrolling**]

```c
/* 2 x 2 loop unrolling */
void combine6(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  long 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;
}
```

- uses **two-way loop unrolling** and **two-way parallelism**

[Performance comparison]

```
                          Integer       Floating pt
                          ---------------------------
Function  Method          +     *        +      *
-------------------------------------------------
combine4  No unrolling    1.27  3.01     3.01   5.01
combine5  2x1 unrolling   1.01  3.01     3.01   5.01
combine6  2x2 unrolling   0.81  1.51     1.51   2.51

Latency bound             1.00  3.00     3.00   5.00
Throughput bound          0.50  1.00     1.00   0.50
```

- improved the performance for all cases by a **factor of around 2**
- **broken through the barrier** imposed by the **latency bound**
  - **two critical paths** - one for even-numbered elements, and one for the odd-numbered elements
    - each contains only $n/2$ operations --> CPE $/2$
  - only **exception** = **integer addition**
    - too much loop overhead to achieve the theoretical limit of 0.50

- in general, a program can **achieve the throughput bound** for an operation only when it can **keep the pipelines filled for all of the functional units capable of performing that operation**
  - for an operation with latency L and capacity C, this requires an unrolling factor $k \geq C \cdot L$

- **MUST consider** whether it **preserves the functionality** of the original function
  - **two's-complement arithmetic** is **commutative** and **associative** even when **overflow occurs**
    - optimizing compiler could convert the code by loop unrolling & parallelism
  - **floating-point multiplication and addition** are **not associative**
    - loop unrolling & parallelism could produce different results due to **rounding** or **overflow**
    - however, **achieving a performance gain of 2x outweighs the risk of generating different results** for strange data patterns

- in real-life applications it is unlikely that multiplying the elements in strict order gives fundamentally better accuracy than does multiplying two groups independently and then multiplying those products together

- most compilers **do not attempt** such transformations since they have no way to judge the risks of introducing transformations can change the program behavior (no matter how small it is)

H5 ¶5.9.2 Reassociation Transformation

[Example C code, **2 x 1a loop unrolling**]

```
/* 2 x 1a loop unrolling */
void combine7(vec_ptr v, data_t *dest) {
  long i;
  long length = vec_length(v);
  long limit = length-1;
  data_t *data = get_vec_start(v);
  data_t acc = IDENT;

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

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

[Difference between combine5 - combine7]

```
/* combine5 */
acc = (acc OP data[i]) OP data[i+1];

/* combine7 */
acc = acc OP (data[i] OP data[i+1]);
```

[Performance Comparison]

		Integer		Floating pt	
Function	Method	+	*	+	*
combine4	No unrolling	1.27	3.01	3.01	5.01
combine5	2x1 unrolling	1.01	3.01	3.01	5.01
combine6	2x2 unrolling	0.81	1.51	1.51	2.51
combine7	2x1a unrollin	1.01	1.51	1.51	2.51
Latency bound		1.00	3.00	3.00	5.00
Throughput bound		0.50	1.00	1.00	0.50

- except for integer addition, other **three cases double the performance** relative to $k \times 1$ unrolling, **breaking** through the barrier imposed by the **latency bound**
  - **single critical path with only n/2 operations**

- **Reassociation transformation** can **reduce the number of operations along the critical path in a computation**, resulting in better performance by **better utilizing the multiple functional units and their pipelining capabilities**
  - However, in general, **loop unrolling + accumulating multiple values in parallel** is more reliable way to achieve improved program performance

H3 §5.10 Summary of Results for Optimizing Combining Code

		Integer		Floating pt	
Function	Method	+	*	+	*
combine1	Abstract -O1	10.12	10.12	10.17	11.14
combine6	2x2 unrolling	0.81	1.51	1.51	2.51
	10x10 unrolling	0.55	1.00	1.01	0.52
Latency bound		1.00	3.00	3.00	5.00
Throughput bound		0.50	1.00	1.00	0.50

- all been done using **ordinary C code and a standard compiler**
- by rewriting the code to take advantage of the newer **SIMD** instructions yields **additional performance gains of ≈4x or 8x**
  - for floating point multiplication, CPE drops to 0.06 (180x performance)

H3 §5.11 Some Limiting Factors

- **Major limiting factors so far**
  1. **critical path** in a data flow
     - if there is **some chain of data dependencies** (sum of all of the latencies = T), then the program will require at least T cycles to execute
  2. **throughput bounds** of the functional units
     - assuming that a program requires a total of N computations of some operation, and the microprocessor has C functional units capable of performing that operation, and have an issue time of I, then the program will require at least $N \cdot I/C$ cycles to execute
- **Two Additional** limiting factors...

H5 ¶5.11.1 Register Spilling

- benefits of loop parallelism are limited by the **ability to express the computation in assembly code**
- If a program has a degree of parallelism P that **exceeds the number of available registers**, then the compiler will resort to ***spilling***
  - **storing** some of the **temporary values in memory** by allocating space on the **run-time stack**

```
                        Integer       Floating pt
                        ---------------------------
Function  Method        +     *       +      *
-------------------------------------------------
combine6
        10x10 unrolling 0.55  1.00    1.01   0.52
        20x20 unrolling 0.83  1.03    1.02   0.68

Throughput bound        0.50  1.00    1.00   0.50
```

- modern **x86-64** processors have **16 integer registers** and can make use of the **16 YMM registers** to store floating-point data
    - once the number of loop variables **exceeds** the number of available registers --> **program must allocate some on the stack**

```
# Updating of accumulator acc0 in 10x10 unrolling
vmulsd  (%rdx), %xmm0, %xmm0    # acc0 *= data[i]

# Updating of accumulator acc0 in 20x20 unrolling
vmovsd  40(%rsp), %xmm0
vmulsd  (%rdx), %xmm0, %xmm0
vmovsd  %xmm0, 40(%rsp)
```

- the **accumulator** is kept as a **local variable on the stack**
    - program **MUST** **read** both its value and the value of data[i] from memory, **multiply** them and **store** the result back to memory

H5 ¶5.11.2 Branch Prediction and Misprediction Penalties

- conditional branch can incur a **significant** **misprediction penalty** when the branch prediction logic does not correctly anticipate whether or not a branch will be taken
- modern processors work well ahead of the currently executing instructions (***instruction pipelining***)
    - when a **branch is encountered,** the processor **must guess** which way the branch will go
- with ***speculative execution***, the processor **begins executing** the instructions at the **predicted branch target**
    - **while not modifying any actual register or memory locations** until the actual outcome has been determined
    - if the prediction is **correct** --> processor **"commits"** the results of the speculatively executed instructions by **storing** them in registers or memory
    - if the prediction is **incorrect** --> processor **"must discard"** all of the speculatively executed results and **restart** the instruction fetch process at the correct location

- **Conditional move** instructions (rather than conditional transfers of control)
    - compute the values along both branches of a conditional expression or statement and then **use conditional moves** to **select the desired value**
    - no need to guess whether or not the condition will hold --> **no penalty for guessing incorrectly**

H6 Do Not Be Overly Concerned about Predictable Branches

- **branch prediction logic** found in modern processors is very good at **discerning regular patterns and long-term trends** for the different branch instructions
- performance of predicted branches is **limited by the latencies of operations**
  - processor is able to **predict** the outcomes of branches, so **none of this evaluation has much effect** on the fetching and processing of the **instructions that form the critical path** in the program execution

H6 Write Code Suitable for Implementation with Conditional Moves

- **branch prediction is only reliable for** <u>regular patterns</u>
  - many tests are completely **unpredictable!**
    - program performance can be **greatly enhanced i** f the compiler is able to generate code **using conditional data transfers** rather than conditional control transfers

[Example 'Naive' C code]

```
1  /* Rearrange two vectors, for each i, b[i] >= a[i] */
2  void minmax1(long a[], long b[], long n) {
3    long i;
4    for (i = 0; i < n; i++) {
5      if (a[i] > b[i]) {
6        long t = a[i];
7        a[i] = b[i];
8        b[i] = t;
9      }
10   }
11 }
```

--> CPE of ≈13.5 for random data & ≈2.5-3.5 for predictable data

[Example 'Better' C code - conditional data transfers]

```c
/* Rearrange two vectors, for each i, b[i] >= a[i] */
void minmax2(long a[], long b[], long n) {
  long i;
  for (i = 0; i < n; i++) {
    long min = a[i] < b[i] ? a[i] : b[i];
    long max = a[i] < b[i] ? b[i] : a[i];
    a[i] = min;
    b[i] = max;
  }
}
```

--> CPE of ≈4.0 regardless of whether the data are arbitrary or predictable

- not all conditional behavior can be implemented with conditional data transfers
  - ∃ cases where programmers cannot avoid writing code that will lead to conditional branches for which the processor will do poorly with its branch predictions
    - try best to use **conditional data transfers**

H3 §5.12 Understanding Memory Performance

- all modern processors contain one or more **cache** memories to provide **fast access** to such small amounts of memory
  - modern processors have dedicated ***functional units*** to perform load and store operations
    - these units have **internal buffers** to hold sets of outstanding requests for memory operations
    - i7 Haswell has... (can initiate 1 operation every clock cycle)
      - **2x load units** - each can hold up to **72** pending read requests
      - **1x store unit** - can hold up to **42** write requests

H5 ¶5.12.1 Load Performance

- performance of a program containing load operations depends on **1) pipelining capability** and **2) latency of the load unit**

- 1) one factor limiting the CPE is the fact that they all require **reading one value from memory** for each element computed
  - with two load units (each able to initiate at most 1 load operation every clock cycle), CPE cannot be less than 0.50
  - for applications loading k values for every element computed, CPE cannot be lower than $k/2$

- 2) performance effects due to the **latency of load operations**

  [Example C code]

```c
typedef struct ELE {
  struct ELE *next;
  long data;
} list_ele, *list_ptr;

long list_len(list_ptr ls) {
  long len = 0;
  while (ls) {
    len++;
    ls = ls->next;
  }
  return len;
}
```

  --> sequence of load operations (outcome of one determines the address for the next)

  --> **list_len** has a CPE of 4.00

  [Generated **Assembly code**]

```asm
# Inner loop of list_len
# ls in %rdi, len in %rax
.L3:                    # loop:
  addq    $1, %rax        # Increment len
  movq    (%rdi), %rdi    # ls = ls->next
  testq   %rdi, %rdi      # Test ls
  jne     .L3             # If nonnull, goto loop
```

  --> **movq** instruction on line 5 forms the **critical bottleneck** in this loop

--> each successive value of register %rdi depends on the result of a load operation having the value in %rdi as its address

--> load operation for one iteration cannot begin until the one for the previous iteration has completed

H5 ¶5.12.2 Store Performance

- ***store*** operation -- writes a register value to memory
  - can operate in a **fully pipelined mode**, beginning a new store on every cycle (CPE of 1.0, with a single store funcitonal unit)
  - **DOES NOT** affect any register values
    - series of store operations **CANNOT** create a **data dependency**
- **Store unit** includes a ***store buffer*** containing the **addresses and data of the store operations** that have been issued to the store unit, **but**, **have not yet been completed** (completion = updating the data cache)
  - series of store operations can be executed without having to wait for each one to update the cache
  - when a **load operation occurs**, it **must check the entries in the store buffer** for matching addresses
    - If it finds a match (any of the bytes being written have the same address as any of the bytes being read), it **retrieves the corresponding data entry** as the result of the load operation

[Example C code]

```
1  /* Set elements of array to 0 */
2  void clear_array(long *dest, long n) {
3    long i;
4    for (i = 0; i < n; i++)
5      dest[i] = 0;
6  }
7
8  /* Write to dest, read from src */
9  void write_read(long *src, long *dst, long n) {
10   long cnt = n;
11   long val = 0;
12
13   while (cnt) {
14     *dst = val;
15     val = (*src)+1;
```

```
16      cnt--;
17    }
18  }
```

--> if function **write_read** is called with arguments **src** and **dest** pointing to the **same memory location**, ***write/read dependency*** causes a slowdown in the processing

[Generated **Assembly code**]

```
1  # Inner loop of write_read
2  # src in %rdi, dst in %rsi, val in %rax
3  .L3:                    # loop:
4    movq    %rax, (%rsi)    # Write val to dst
5    movq    (%rdi), %rax    # t = *src
6    addq    $1, %rax        # val = t+1
7    subq    $1, %rdx        # cnt--
8    jne     .L3             # If != 0, goto loop
```

- With matching source and destination addresses, **data dependency** between the **s_data** (movq instruction in L4 is translated into two operations: 1) s_addr, 2) s_data where s_addr computes the address for the store operation, creates an entry in the store buffer, and sets the address field for that entry, and s_data sets the data field for the entry) causes a **critical path** to form involving data being stored, loaded, and incremented

- with operations on **registers** -- **processor can determine** which instructions will affect which others as they are being decoded into operations
- with operations on **memories** -- **processor cannot predict** which will affect which others until the load and store addresses have been computed

H3 §5.13 Life in the Real World: Performance Improvement Techniques

Basic strategies for optimizing program performance:

1. ***High-level design*** -- choose appropriate algorithms and data structures for the problem at hand

2. ***Basic coding principles*** -- avoid optimization blockers so that a compiler can generate efficient code
    - Eliminate excessive function calls
        - move computations out of loops when possible
    - Eliminate unnecessary memory references
        - introduce temporary variables to hold intermediate results
        - store a result in an array or global variable only when the final value has been computed
3. ***Low-level optimizations*** -- structure code to take advantage of the hardware capabilities
    - **unroll loops** to reduce overhead and to enable further optimizations
    - find ways to increase **instruction-level parallelism** by techniques such as **multiple accumulators** and **reassociations**
    - rewrite conditional operations in a functional style to enable compilation via **conditional data transfers**

H3 §5.14 Idenfitying and Eliminating Performance Bottlenecks

- when working with large programs, it may be difficult to find where to focus more for oprimization
- ***code profilers*** ; analysis tools that collect **performance data** about a program as it executes

H5 ¶5.14.1 Program Profiling

- to determine how much time the different parts of the program require

- Unix systems provide the profiling program **GPROF**
    1. determines how much **CPU time** was spent for each of the functions in the program
    2. computes a **count of how many times each function gets called** , categorized by which function performs the call

- Profiling with **GPROF** in three steps

Suppose we have a C program **prog.c** which runs with command-line argument **file.txt**

1. program must be compiled and linked for profiling
   - include the run-time flag **-pg** on the command line

```
linux> gcc -Og -pg prog.c -o prog
```

2. program is then executed as usual

```
linux> ./prog file.txt
```

   - runs slightly slower than normal, generating a file **gmon.out**
3. GPROF is invoked to analyze the data in **gmon.out**

```
linux> gprof prog
```

- the **first part** of the profile report **lists the times spent executing the different functions** , sorted in descending order
- the **second part** of the profile report shows the **calling history of the functions**

- Properties of **GPROF**
  1. timing is **not very precise**
  2. calling information is **quite reliable**
  3. by default, **the timings for library functions are not shown** , these times are **incorporated into the times for the calling functions**

H3 §5.15 Summary

- No compiler can replace an inefficient algorithm or data structure by a good one

  --> **better program design** should be a primary concern for programmers

1. **Eliminate optimization blockers**
   - memory aliasing
   - procedure calls

2. **Know processor's microarchitecture** (OOO processors)
    - capabilities, latencies, issue times of the functional units
3. **Exploit the instruction-level parallelism** provided by modern processors
    - loop unrolling
    - creaing multiple accumulators
    - reassociation
4. **Analyze data dependencies**
    - critical paths between the different iterations of a loop
5. **Use Conditional data transfers**
    - make branches more predictable
    - keep values in local variables, allowing them to be stored in registers
6. **Use Code profilers**
    - to analyze the program